

Dissertação apresentada à Pró-Reitoria de Pós-Graduação do Instituto Tecnológico de Aeronáutica, como parte dos requisitos para obtenção do título de Mestre em Engenharia do Curso de Mestrado Profissional em Engenharia Aeronáutica no Programa de Pós-Graduação em Engenharia Aeronáutica e Mecânica.

João Paulo Monteiro Cruvinel da Costa

**DESENVOLVIMENTO DE UMA FERRAMENTA
EM PYTHON PARA A ANÁLISE DE
AEROELASTICIDADE ESTÁTICA EM
AERONAVES FLEXÍVEIS**

Dissertação aprovada em sua versão final pelos abaixo assinados:

Prof. Dr. Roberto Gil Annes da Silva
Orientador

Me. Marcos Paulo Halal Lombardi
Coorientador

Prof. Dr. Pedro Teixeira Lacava
Pró-Reitor de Pós-Graduação

Campo Montenegro
São José dos Campos, SP – Brasil
2019

Dados Internacionais de Catalogação-na-Publicação (CIP)
Divisão de Informação e Documentação

Costa, João Paulo Monteiro Cruvinel da

Desenvolvimento de uma ferramenta em Python para a análise de aeroelasticidade estática em aeronaves flexíveis / João Paulo Monteiro Cruvinel da Costa.

São José dos Campos, 2019.

221f.

Dissertação de Mestrado – Curso de Engenharia Aeronáutica e Mecânica, Área de Engenharia Aeronáutica – Instituto Tecnológico de Aeronáutica, 2019. Orientador: Prof. Dr. Roberto Gil Annes da Silva. Coorientador: Me. Marcos Paulo Halal Lombardi

1. Método de Malha Turbilhonar. 2. Aeroelasticidade. 3. Estabilidade de Aeronaves. 4. Método de elementos finitos. 5. Engenharia Aeronáutica. I. Instituto Tecnológico de Aeronáutica. II. Desenvolvimento de uma ferramenta em Python para a análise de aeroelasticidade estática em aeronaves flexíveis

REFERÊNCIA BIBLIOGRÁFICA

COSTA, João Paulo Monteiro Cruvinel da. **Desenvolvimento de uma ferramenta em Python para a análise de aeroelasticidade estática em aeronaves flexíveis**. 2019. 221f. Dissertação de mestrado em Engenharia Aeronáutica – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSÃO DE DIREITOS

NOME DO AUTOR: João Paulo Monteiro Cruvinel da Costa

TÍTULO DO TRABALHO: Desenvolvimento de uma ferramenta em Python para a análise de aeroelasticidade estática em aeronaves flexíveis

TIPO DO TRABALHO/ANO: Dissertação / 2019

É concedida ao Instituto Tecnológico de Aeronáutica permissão para reproduzir cópias desta dissertação e para emprestar ou vender cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte desta dissertação ou tese pode ser reproduzida sem a sua autorização (do autor).


João Paulo Monteiro Cruvinel da Costa
Av. Quinze de Novembro, 850, Apartamento 126
CEP: 14801-030, Araraquara - SP

**DESENVOLVIMENTO DE UMA FERRAMENTA
EM PYTHON PARA A ANÁLISE DE
AEROELASTICIDADE ESTÁTICA EM
AERONAVES FLEXÍVEIS**

João Paulo Monteiro Cruvinel da Costa

Composição da Banca Examinadora:

Prof. Dr. Roberto Gil Annes da Silva
Me. Marcos Paulo Halal Lombardi
Prof. Dr. Flávio Luiz Cardoso Ribeiro
Dr. Leandro Afonso da Silva

Presidente/Orientador - ITA
Coorientador - EMBRAER
Membro - ITA
Membro - EMBRAER

Dedico esse trabalho ao
meu avô Filogônio Luiz Cruvinel

Agradecimentos

Agradeço primeiramente a minha família, em especial ao meus pais, cujo apoio e dedicação são o principal motivo por trás das minhas conquistas.

Ao Prof. Gil, meu orientador, pela orientação, interesse e disponibilidade durante a execução desse trabalho, e principalmente por estar disposto a me ajudar mesmo à distância e lidar com as complicações que isso gerou.

Ao meus colegas de trabalho, em especial meu coorientador Marcos Paulo Halal Lombardi, por estarem sempre dispostos a partilharem um pouco do seu tempo para tirar minhas dúvidas e por me manterem motivado.

Aos cientistas e engenheiros em cuja obra minha educação e esse trabalho são baseados.

A Deus, por ter me dado a saúde e a curiosidade necessárias para completar essa dissertação.

"I have not the smallest molecule of faith in aerial navigation other than ballooning"
— LORD KELVIN, 1896

Resumo

Este trabalho apresenta o desenvolvimento de uma ferramenta computacional, chamada *Flying Circus* usando a linguagem de programação *Python* e bibliotecas de código aberto para a análise de aeroelasticidade estática de aeronaves em fase de projeto conceitual.

A ferramenta usa um método de *vortex lattice* para os cálculos aerodinâmicos, um método de elementos finitos baseado em elementos de viga de Euler-Bernoulli para o cálculo estrutural, um critério de proximidade para o acoplamento aerodinâmico/estrutural e um método iterativo para o cálculo aeroelástico.

Os resultados obtidos por essa ferramenta foram validados contra outros resultados disponíveis na literatura. A ferramenta foi utilizada para avaliar uma aeronave UAV (*Unmanned Aerial Vehicle*) da categoria HALE (*High Altitude Long Endurance*).

O código fonte da ferramenta desenvolvida foi incluído nos anexos, juntamente com o código utilizado para gerar os resultados apresentados.

Abstract

This work presents the development of a computational tool, called Flying Circus using the Python programming language and open source libraries for static aeroelasticity analysis of aircraft in the conceptual development stage.

The tool uses a vortex lattice method for aerodynamic calculations, a finite element method using Euler-Bernoulli beam elements for structural calculation, a proximity criterion for aerodynamic/structural coupling, and an iterative method for aeroelastic computation.

The results obtained were validated against other results available in the literature. The tool was also used to evaluate a UAV (Unmanned Aerial Vehicle) of the HALE (High Altitude Long Endurance) category.

The source code of the developed tool was included in the annexes, along with the code used to generate the results presented.

Listas de Figuras

FIGURA 1.1 – Alongamento vs. ano do primeiro voo, de diversas aeronaves do tipo <i>long-haul</i>	19
FIGURA 1.2 – Ponta de asa dobrável da aeronave <i>Boeing 777-9X</i>	19
FIGURA 1.3 – Global Hawk no centro de pesquisa de Dryden da NASA	20
FIGURA 1.4 – <i>Boeing 787 Dreamliner</i>	20
FIGURA 1.5 – <i>General Atomics Altair</i>	21
FIGURA 2.1 – Discretização de uma asa em <i>vortex lattice</i> (DRELA, 2014)	28
FIGURA 2.2 – Representação vetorial da velocidade induzida em um ponto por um vórtice ferradura (DRELA, 2014)	30
FIGURA 2.3 – Força aerodinâmica agindo em um painel (DRELA, 2014)	31
FIGURA 2.4 – Discretização de uma viga em elementos finitos	33
FIGURA 2.5 – Construção da matriz de rigidez global do sistema	34
FIGURA 2.6 – Elemento finito de viga utilizado	35
FIGURA 2.7 – Cossenos diretores (LOGAN, 2011)	37
FIGURA 2.8 – Diagrama mostrando a interação entre as disciplinas envolvidas no estudo de aeroelasticidade	38
FIGURA 2.9 – Diagrama de um aerofólio 2D	40
FIGURA 3.1 – Fluxo de Informação entre os <i>subpackages</i> da Ferramenta	44
FIGURA 3.2 – Sistema de coordenadas utilizado pelo pacote <i>geometry</i> juntamente com o modelo de uma aeronave comercial	46
FIGURA 4.1 – Evolução das deflexões calculadas com o número de Painéis	53
FIGURA 4.2 – Evolução do C_l calculado com o número de painéis	53

FIGURA 4.3 – Evolução do tempo de simulação com o número de painéis	54
FIGURA 4.4 – Malhas original e deformadas, em azul a malha original, em laranja a deformada para $\alpha = 2$ e em verde a deformada para $\alpha = 4$	55
FIGURA 4.5 – Delta de pressão entre intradorso e extradorso para $\alpha = 2$ graus . .	56
FIGURA 4.6 – Delta de pressão entre intradorso e extradorso para $\alpha = 4$ graus . .	57
FIGURA 4.7 – Evolução na deformação da ponta da asa durante as iterações para $\alpha = 2$ graus	58
FIGURA 4.8 – Evolução na deformação da ponta da asa durante as iterações para $\alpha = 4$ graus	59
FIGURA 4.9 – C_l vs. α para a Asa Smith, altitude: 20000m, TAS: 25 m/s	60
FIGURA 4.10 – Distribuição de sustentação ao longo da envergadura, para a asa rígida e flexível	61
FIGURA 4.11 – Deflexão em Z ao longo da envergadura	62
FIGURA 4.12 – Torção em Y ao longo da envergadura	62
FIGURA 4.13 – Comparação entre os resultados de flexão para $\alpha = 2$ graus	63
FIGURA 4.14 – Comparação entre os resultados de torção para $\alpha = 2$ graus	64
FIGURA 4.15 – Comparação entre os resultados de flexão para $\alpha = 4$ graus	64
FIGURA 4.16 – Comparação entre os resultados de torção para $\alpha = 4$ graus	65
FIGURA 4.17 – Ilustração fornecida por Patil, Hodges e Cesnik da aeronave estudada	67
FIGURA 4.18 – Vista superior do modelo utilizado	68
FIGURA 4.19 – Vista em 3/4 do modelo utilizado	70
FIGURA 4.20 – Vista em 3/4 das malhas aerodinâmica e estrutural deformadas . .	72
FIGURA 4.21 – Flexão da asa	73
FIGURA 4.22 – Torção da asa	74
FIGURA 4.23 – Vista em 3/4 da deformação e delta de pressão para $\alpha = 2$ graus . .	75
FIGURA 4.24 – Curva de C_l da aeronave para os ângulos de ataque simulados . . .	76
FIGURA 4.25 – Curva de C_m da aeronave para os ângulos de ataque simulados . .	77
FIGURA 4.26 – Curva de C_d da aeronave para os ângulos de ataque simulados . .	78
FIGURA 4.27 – Distribuição de sustentação ao longo da asa	79
FIGURA 4.28 – Distribuição de sustentação ao longo da empenagem horizontal . .	80

Lista de Tabelas

TABELA 4.1 – Propriedades da Asa Smith	52
TABELA 4.2 – Propriedades da aeronave HALE	69
TABELA 4.3 – Propriedades da Malha Aerodinâmica e Estrutural da Aeronave Hale	71

Lista de Abreviaturas e Siglas

2D	Duas dimensões ou bidimensional
3D	Três dimensões ou tridimensional
IATA	International Air Transport Association
CFD	Computational Fluid Dynamics
C.G.	Centro de Gravidade
HALE	High Altitude Long Endurance
HPC	High Performance Computing
TAS	True Airspeed
UAV	Unmanned Aerial Vehicle

Listas de Símbolos

.	Produto escalar
\times	Produto vetorial
A	Área de uma seção
A_{ij}	Matrix dos coeficientes de influência aerodinâmica
a	Vetor $r - r_a$ apresentado na figura 2.2
a_1	Inclinação da curva C_l vs α de um aerofólio
b	Vetor $r - r_b$ apresentado na figura 2.2
C_d	Coeficiente de arrasto
C_l	Coeficiente de sustentação
C_m	Coeficiente de momento de arfagem
$C_{pq'}$	Cosseno entre os vetores p e q'
c	Corda de um aerofólio ou seção de asa
E	Módulo de elasticidade de um material
ec	Distância entre o centro aerodinâmico e o eixo elástico de um aerofólio
F	Vetor de cargas nodais no sistema de coordenadas global
G	Módulo de elasticidade torcional de um material
I_{yy}	Momento de Inércia de uma seção em relação ao eixo Y
I_{zz}	Momento de Inércia de uma seção em relação ao eixo Z
J	Momento Polar de Inércia de uma seção em relação ao eixo X
K	Matrix de rigidez global do sistema
K_θ	Rigidez de uma mola torsional
k	Matrix de rigidez global de um elemento finito de viga
k'	Matrix de rigidez local de um elemento finito de viga
ℓ_i	Vetor que define o segmento de vórtice
$Mach$	Número de Mach, razão entre a velocidade do escoamento e a velocidade do som
n_{0i}	Vetor normal ao painel aerodinâmico i
q	Pressão dinâmica, $0.5\rho^2$
R	ec^2a_1/K_θ
Re	Número de Reynolds, razão entre as forças inerciais e viscósas dentro de um fluido
r	Vetor que define a posição de um ponto no espaço

r_i^c	Ponto de controle do painel aerodinâmico i
T	Matrix de rotação de um elemento de viga
T^T	Matrix de rotação de um elemento de viga transposta
U	Vetor de velocidade translacional da aeronave em relação ao escoamento
u', v', w'	Deformações translacionais em X' , Y' e Z' do elemento finito de viga
V	Velocidade do escoamento
$V(r)$	Vetor de velocidade do escoamento em um ponto r
$\hat{V}_i(r)$	Integral de Biot-Savart, expressa na equação 2.2
X, Y, Z	Eixos X , Y e Z do sistema de coordenadas global da aeronave
X', Y', Z'	Eixos X , Y e Z do sistema de coordenadas local do elemento finito de viga
x	Vetor de deformações no sistema de coordenadas global
\hat{x}	Vetor unitário do eixo X
α	Ângulo de ataque
Γ_i	Intensidade de um filamento de vórtice i
ϵ	Parâmetro de distância mínima a um segmento de vórtice
θ	Ângulo de incidência devido uma carga aerodinâmica
θ_0	Ângulo de incidência inicial de um aerofólio
ρ	Densidade do ar
ϕ_x, ϕ_y, ϕ_z	Deformações angulares em X' , Y' e Z' do elemento finito
Ω	Vetor de velocidade rotacional da aeronave em relação ao escoamento

Sumário

1	INTRODUÇÃO	18
1.1	Motivação	18
1.2	Objetivos	21
1.3	Revisão Bibliográfica	22
1.4	Estrutura do Trabalho	24
2	FUNDAMENTAÇÃO TEÓRICA	26
2.1	Aerodinâmica	26
2.1.1	Método de <i>Vortex Lattice</i>	27
2.2	Estruturas	32
2.2.1	Método dos Elementos Finitos	32
2.3	Aeroelasticidade	37
2.3.1	Aeroelasticidade Estática	39
3	METODOLOGIA	42
3.1	Subpacote <i>geometry</i>	44
3.2	Subpacote <i>aerodynamics</i>	45
3.2.1	Módulo <i>functions</i>	46
3.2.2	Módulo <i>objects</i>	47
3.2.3	Módulo <i>vlm</i>	47
3.3	Subpacote <i>structures</i>	47
3.4	Subpacote <i>aeroelasticity</i>	48
3.4.1	Passo 1: Cálculo das cargas aerodinâmicas	48
3.4.2	Passo 2: Aplicação das cargas aerodinâmicas na estrutura	49

3.4.3	Passo 3: Cálculo da deformação da estrutura	49
3.4.4	Passo 4: Deformação da geometria da aeronave	49
3.4.5	Passo 5: Iteração	50
3.5	Subpacote <i>loads</i>	50
3.6	Subpacote <i>visualization</i>	50
4	RESULTADOS	51
4.1	Validação	51
4.1.1	Asa Smith	51
4.2	Estudo de Caso - Aeronave Hale	66
4.2.1	Descrição do Modelo e da Simulação	66
4.2.2	Modelo	67
4.2.3	Simulação	70
4.2.4	Resultados	71
5	CONCLUSÃO	81
5.1	Comentários Finais	81
5.2	Sugestões para Trabalhos Futuros	81
	REFERÊNCIAS	83
	APÊNDICE A – CÓDIGO FONTE	86
A.1	__init__.py	86
A.2	mathematics.py	87
A.3	aerodynamics	90
A.3.1	__init__.py	90
A.3.2	functions.py	90
A.3.3	objects.py	96
A.3.4	vlm.py	96
A.4	aeroelasticity	102
A.4.1	__init__.py	102
A.4.2	functions.py	102

A.5 <i>geometry</i>	117
A.5.1 __init__.py	117
A.5.2 functions.py	117
A.5.3 objects.py	130
A.6 <i>loads</i>	145
A.6.1 __init__.py	145
A.6.2 functions.py	145
A.7 <i>structures</i>	151
A.7.1 __init__.py	151
A.7.2 fem.py	151
A.7.3 functions.py	161
A.7.4 objects.py	162
A.8 <i>visualization</i>	168
A.8.1 __init__.py	168
A.8.2 plot_2D.py	168
A.8.3 plot_3D.py	169
APÊNDICE B – EXEMPLOS DE APLICAÇÃO	178
B.1 Asa Smith	178
B.1.1 smith_wing_data.py	178
B.1.2 smith_wing_simulation.py	180
B.1.3 smith_wing_results.py	186
B.1.4 smith_wing_mesh_sensitivity.py	196
B.2 Hale Aircraft	198
B.2.1 hale_aircraft_data.py	198
B.2.2 hale_aircraft_simulation.py	204
B.2.3 hale_aircraft_results.py	210
B.2.4 hale_aircraft_results_processing.py	214

1 Introdução

1.1 Motivação

Um dos principais objetivos durante o projeto da maioria das aeronaves é a redução do arrasto por ela sofrido durante o voo, especialmente durante o cruzeiro. Essa preocupação vem de fatores operacionais, econômicos e, nas últimas décadas, ecológicos.

Do ponto de vista operacional o alcance e autonomia de uma aeronave é proporcional à sua razão de sustentação e arrasto, portanto, para o projeto de uma aeronave de grande autonomia uma razão de *sustentação/arrasto* (L/D) favorável é um requisito.

Economicamente, o gasto de combustível de uma aeronave é parte considerável do seu custo operacional total (IATA, 2018), portanto uma aeronave com baixo consumo tende a ter custos operacionais mais baixos. Isso faz com que exista uma grande pressão dos operadores sobre os fabricantes de aeronaves para que projetos novos sejam cada vez mais eficientes.

Por fim preocupações relacionadas às emissões de CO_2 (dióxido de carbono), entre outros poluentes, de aeronaves comerciais e sua contribuição para a acentuação do efeito estufa tem gerado grande incentivo para que novos projetos sejam o mais eficientes possíveis. A comunidade aeronáutica tem respondido a essas preocupações como pode ser visto por iniciativas como o projeto *Clean Sky* (CLEANSKYJU, 2018), da União Europeia, que visa ao desenvolvimento de novas tecnologias, e a criação de metas como a proposta criada pela IATA (IATA, 2019) de aumentar a eficiência no consumo de combustível de aeronaves em 1,5% ao ano no período de 2009 a 2020, e diminuir pela metade as emissões de CO_2 relacionadas à aviação até 2050, em comparação com os níveis de 2005.

Dentre os tipos de arrasto sofridos por uma aeronave se destaca o arrasto induzido, o qual chega a ser responsável por até 43% do total de arrasto sofrido por um avião comercial de grande porte (ABBAS *et al.*, 2013). Uma das decisões de projeto que levam à diminuição do arrasto induzido é o aumento do alongamento da asa da aeronave (RAYMER, 1992). Essa abordagem tem sido utilizada pela indústria como apresentado na figura 1.1 (AFONSO *et al.*, 2017) que mostra o aumento no alongamento das asas das aero-

naves com seu ano de lançamento, e pode ser exemplificado pelo novo *Boeing 777X*, figura 1.2, no qual o aumento da envergadura da asa gerou a necessidade de um mecanismo que permite que as pontas das asas se dobrarem em solo para permitir o acesso da aeronave a *gates* de aeroportos já existentes. Asas de alto alongamento também são frequentemente encontradas em UAVs (*Unmanned Aerial Vehicles*) de longo alcance como por exemplo o *RQ-4 Global Hawk* fabricado pela *Northrop Grumman* mostrado na figura 1.3, cuja asa tem um alongamento de 25.

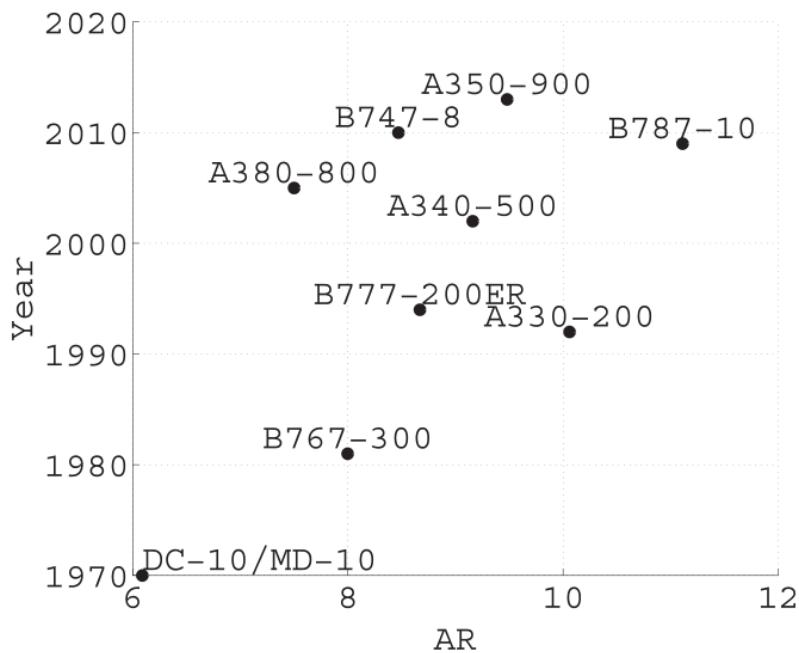


FIGURA 1.1 – Alongamento vs. ano do primeiro voo, de diversas aeronaves do tipo *long-haul*

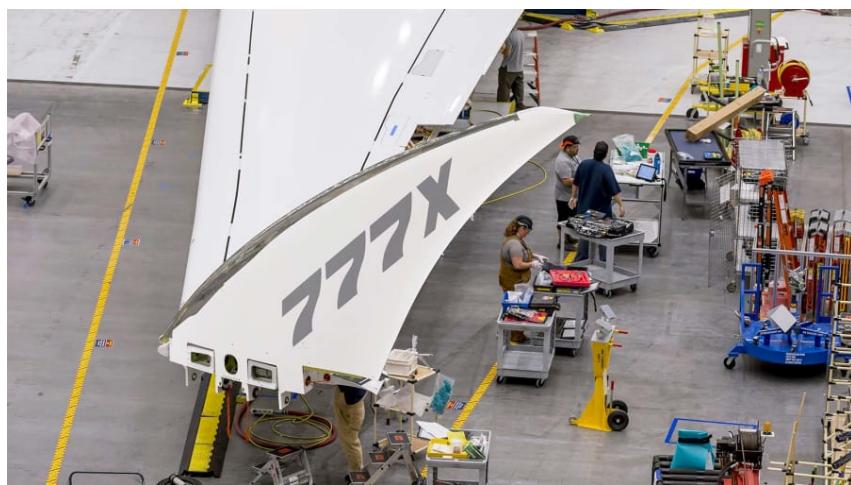


FIGURA 1.2 – Ponta de asa dobrável da aeronave *Boeing 777-9X*

O aumento do alongamento da asa de uma aeronave é benéfico para a diminuição de arrasto mas traz consigo algumas complicações. Dadas as limitações de peso às quais a



FIGURA 1.3 – Global Hawk no centro de pesquisa de Dryden da NASA

aeronave está sujeita e os materiais de construção disponíveis, uma asa muito alongada acaba por ter uma flexibilidade considerável. O uso de materiais compósitos, mais leves porém mais flexíveis, acaba por exacerbar essa característica. A grande flexibilidade faz com que, mesmo em condições normais de voo, as asas dessas aeronaves sofram deformações consideráveis, claramente perceptíveis em aeronaves como o *Boeing 787 Dreamliner* (figura 1.4) e o *General Dynamics Altair*, uma variante do *General Dynamics MQ-9 Reaper* (figura 1.5).



FIGURA 1.4 – *Boeing 787 Dreamliner*



FIGURA 1.5 – *General Atomics Altair*

As grandes deflexões sofridas por esse tipo de asa fazem com que várias simplificações geralmente usadas no projeto conceitual de aeronaves deixem de ser válidas, já que a geometria da aeronave real submetida às cargas de voo passa a divergir consideravelmente da sua geometria de projeto. Metodologias mais robustas, usadas em fases mais avançadas do projeto, são capazes de levar em consideração esses efeitos de flexibilidade, entretanto elas demandam um grande detalhamento da geometria da aeronave e de sua estrutura além de possuir um custo computacional considerável. Tendo em vista que durante a fase conceitual de projeto informações detalhadas não estão disponíveis e o grande número de iterações necessário o uso desses métodos se torna inviável.

Caso essa tendência de aeronaves cada vez mais flexíveis continue, faz-se necessário o desenvolvimento de ferramentas que facilitem o projeto conceitual de aeronaves com essas características. Esse trabalho trata do desenvolvimento de uma dessas ferramentas.

1.2 Objetivos

O objetivo desse trabalho é desenvolver uma ferramenta que permita realizar a análise de cargas aerodinâmicas em aeronaves flexíveis, cujo custo computacional, detalhamento e precisão sejam compatíveis com a etapa de projeto conceitual de aeronaves.

São requisitos que essa ferramenta deve cumprir:

1. Permitir ao usuário a análise de uma gama diversa de geometrias, parametrizadas apenas com características tipicamente conhecidas nessa etapa de projeto como área, alongamento, afilamento, diedro e torção geométrica.

2. Permitir ao usuário a reprodução da geometria de aeronaves já existentes com um grau adequado de fidelidade.
3. Fornecer resultados cujo grau de precisão seja adequado à etapa de projeto conceitual de aeronaves.
4. Ter um custo computacional baixo, de forma que um usuário possa realizar um grande número de análises em um tempo adequado, utilizando apenas um computador do tipo *workstation*, ou seja, as análises não devem depender de acesso a instalações de *High Performance Computing* - HPC como *clusters*.
5. Ser desenvolvida em uma linguagem de código aberto e ter seu código fonte disponibilizado, de forma que seja facilmente expansível e customizável para se adequar às necessidades de diversos usuários.

1.3 Revisão Bibliográfica

Foi feita uma pesquisa na literatura disponível de forma à identificar trabalhos com objetivos similares que pudessem acrescentar e orientar o trabalho aqui apresentado. A pesquisa focou tanto as ferramentas projetadas para projeto conceitual de aeronaves como aquelas mais específicas para a análise de aeroelasticidade. Segue a seguir um breve resumo das referências mais relevantes.

Rentema (2004) desenvolveu uma ferramenta computacional de suporte ao projeto conceitual, chamada AIDA (*Artificial Intelligence supported conceptual Design of Aircraft*) que usa inteligência artificial. Baseada nos critérios definidos pelo usuário a ferramenta sugere uma configuração inicial, avalia as características de desempenho da configuração sugerida e propõe modificações.

Raymer desenvolveu a ferramenta RDS-win (2019) para desenvolvimento de aeronaves. Inicialmente uma implementação do método descrito em seu livro (RAYMER, 1992) com objetivos didáticos a ferramenta foi expandida e em sua versão de uso profissional permite ao usuário realizar análises de desempenho, alcance, custos, gerar modelos 3D da aeronave projetada e usar rotinas de otimização.

Roskam, juntamente com a DARcorporation desenvolveu a ferramenta AAA - *Advanced Aircraft Analysis* (2019). Desenvolvida a partir dos livros do mesmo autor (ROSKAM, 1987-1990) a ferramenta foi expandida e atualmente possui consideráveis capacidades permitindo a estimativa das propriedades de peso, aerodinâmica, desempenho, estabilidade e controle.

CEASION - Conceptual Aircraft Design Tool (CEASION, 2019) é um conjunto de mó-

dulos desenvolvidos em MATLAB que permite o projeto conceitual de aeronaves usando uma combinação de metodologias de baixa, alta e média fidelidade. Ele é composto pelos módulos *AcBuilder* responsável pela definição da geometria da aeronave, *AMB - Aerodynamic Model Builder* capaz de calcular as características aerodinâmicas da aeronave, *SDSA - Simulation and Dynamic Stability Analyser* para simulação dinâmica e análise de estabilidade, *FCSDT (Flight Control System Design Toolkit)* para projeto de sistemas de controle e análise de confiabilidade, *NeoCASS (Next generation Conceptual Aero-Structural Sizing Suite)* para dimensionamento estrutural e análises de aeroelasticidade estática e dinâmica. Uma versão em *python*, chamada de *CEASIONpy* está em desenvolvimento.

SUAVE ((STANFORD-AEROSPACE-DESIGN-LAB, 2019)) é uma ferramenta escrita em *python* que fornece uma estrutura para analizar e otimizar aeronaves convencionais ou não-convencionais. O código desenvolvido permite ao usuário agrupar informações de várias fontes diferentes de forma a permitir o projeto de aeronaves com tecnologias avançadas usando métodos baseados em cálculos físicos juntamente com correlações históricas.

Travaglini (2016) desenvolveu a ferramenta *PyPAD - A Python Package for Preliminary Aircraft Design*, dedicada ao projeto preliminar de aeronaves de asa fixa. O código é composto por três principais módulos, o primeiro *PyGFEM* capaz de gerar modelos em elementos finitos da aeronave a partir da sua descrição, *PyAERO* responsável pelo cálculo das cargas aerodinâmicas e resposta aeroelástica da aeronave, *PySIZE* dedicado ao dimensionamento da estrutura baseado em um processo de otimização e critérios como tensões máximas e resposta aeroelástica.

Os trabalhos apresentados a seguir tratam mais especificamente da análise de aeroelasticidade.

Drela (1999) desenvolveu um modelo integrado para simulação de aerodinâmica, estrutura e controle de aeronaves. O modelo usa uma rede de vórtices/fontes com correções de compressibilidade e vigas não-lineares para a simulação de deformações arbitrariamente grandes. O modelo foi implementado no software ASWING (DRELA, 2010) que permite o rápido diagnóstico de estol local, falha estrutural e saturação do sistema de controle.

Ruggeri (2015) desenvolveu um código de *vortex lattice*, chamado de VLM4FW, capaz de levar em consideração a deformação aeroelástica de asas. O código foi acoplado com o programa comercial Abaqus para o cálculo, usando efeitos aeroelásticos com geometria não-linear, de cargas estacionárias. Ruggeri também calculou modos de vibração de asas flexíveis em sua forma deformada usando o software MSC.Nastran e investigou os efeitos relativos a estabilidade aeroelástica dinâmica (*flutter*) da deformação da asa usando o método ZONA6 do software ZAERO.

Ribeiro (2011) desenvolveu uma formulação matemática para a modelagem de aeronaves de grande flexibilidade. Usando um modelo de viga não-linear para a modelagem

estrutural e a teoria de faixas para o cálculo aerodinâmico Ribeiro investigou como a flexibilidade afeta as características de dinâmica de voo usando uma asa voadora como estudo de caso.

Grosdanov (2017) desenvolveu uma ferramenta para o cálculo de aeroelasticidade estática de aeronaves com voo em regime subsônico e transônico usando um método de *vortex lattice* combinado com dados 2.5D obtido de simulações CFD RANS e uma formulação corrotacional de viga de Euler-Bernoulli. Os resultados obtidos foram comparados com aqueles de outros métodos e uma boa concordância entre as abordagens foi encontrada.

Smith, Patil e Hodges (2001) analisaram uma asa retangular de alto alongamento usando três metodologias diferentes: um código de painéis usando um modelo de viga não-linear; um código de CFD usando um modelo de viga linear; e um código de CFD usando um modelo de viga não-linear. Foram estudadas duas condições de voo: uma a 20000 m de altitude, velocidade de 25 m/s e ângulo de ataque de 2 graus; a segunda na mesma altitude e velocidade mas com ângulo de ataque igual a 4 graus.

Patil, Hodges e Cesnik (2001) analisaram as características de voo de uma aeronave HALE (grande altitude, longa autonomia) utilizando uma abordagem não-linear rigorosa, foi concluído que as características de voo de uma aeronave desse tipo podem divergir muito daquelas previstas se efeitos não-lineares de flexibilidade não forem levados em consideração.

1.4 Estrutura do Trabalho

Este trabalho foi dividido em cinco capítulos e dois apêndices.

O primeiro capítulo trata da motivação por trás do trabalho, bem como dos objetivos almejados. Também é apresentada uma revisão sobre trabalhos anteriores presentes na literatura cujos objetivos são correlatos aos dessa dissertação.

O segundo capítulo apresenta um resumo das bases teóricas utilizadas, divididas nas disciplinas de aerodinâmica, estruturas e aeroelasticidade.

O terceiro capítulo se dedica a explicar a metodologia implementada e a estrutura do código desenvolvido.

O quarto capítulo traz comparações entre os resultados obtidos com a ferramenta desenvolvida e resultados presentes na literatura, visando à validação do cálculo apresentado e à estimativa de seu nível de precisão. Também é apresentado um estudo de caso onde a ferramenta é aplicada em uma situação similar àquela encontrada no mundo real. O caso trata de uma aeronave UAV do tipo HALE (*High Altitude Long Endurance*).

O quinto capítulo traz as conclusões finais obtidas com a dissertação e apresenta sugestões para trabalhos futuros.

O apêndice A é composto pelo código fonte da ferramenta desenvolvida, incluindo comentários.

O apêndice B contém os códigos usados especificamente para obter os resultados apresentados, de forma a facilitar a sua reprodução e fornecer exemplos de utilização da ferramenta.

2 Fundamentação Teórica

A ferramenta desenvolvida nesse trabalho utiliza o método de *vortex lattice* para simular o comportamento do escoamento ao redor da aeronave, e um método de elementos finitos baseado em elementos de viga de Euler-Bernoulli para calcular as deformações sofridas pela sua estrutura. Esse capítulo resume as bases teóricas nas quais esses métodos, e consequentemente esse trabalho, são fundamentados.

Também é apresentada uma breve discussão sobre aeroelasticidade e seus principais fenômenos.

2.1 Aerodinâmica

Aerodinâmica é o estudo do movimento e das forças resultantes da interação entre o ar e objetos se movendo através dele. O termo se origina da junção de duas palavras gregas: *aerios*, relativo ao ar, e *dynamis*, força (NASA, 2015). Ela é um subcampo da disciplina de mecânica de fluidos.

O ar, assim como outros fluídos, tem seu comportamento descrito pelas equações diferenciais de mecânica de fluídos, em especial a equação de *Navier-Stokes*. As dificuldades em lidar com essas equações, tanto analiticamente como computacionalmente, somadas às características do ar e à natureza dos problemas encontrados na área aeronáutica, levaram ao desenvolvimento de técnicas e simplificações que permitem a obtenção de resultados com grau de precisão suficiente para uma grande variedade de aplicações.

A primeira dessas simplificações é considerar a viscosidade do ar como desprezível. Para escoamentos com altos números de Reynolds (Re), como aqueles encontrados pela maioria das aeronaves, os efeitos de viscosidade estão confinados a finas camadas limite e finas esteiras, o fluido fora dessa regiões limitadas pode ser considerado não viscoso sem grande perda de acuracidade (KATZ; PLOTKIN, 2001). As equações de mecânica de fluídos considerando a viscosidade do ar igual a zero são conhecidas como equações de Euler.

Caso os elementos de um fluido não possuam velocidade angular, ou seja, seu movi-

mento é apenas de translação pura, o escoamento é considerado irrotacional. Existe um grande número de problemas aerodinâmicos onde o escoamento pode ser considerado praticamente irrotacional; nesses casos, embora o escoamento dentro das regiões de camada limite seja altamente rotacional, o escoamento fora dessas regiões pode ser frequentemente considerado irrotacional (ANDERSON, 2010). Um escoamento que é irrotacional e não viscoso é chamado de escoamento potencial.

Outra simplificação consiste em desprezar a compressibilidade do ar. Para a maioria dos escoamentos encontrados na prática no ramo aeronáutico o ar pode ser considerado incompressível para números de *Mach* menores que 0,3-0,5 (HOUGHTON *et al.*, 2012).

Apesar de útil, a teoria desenvolvida com base nessas simplificações não é suficiente para descrever o fluxo de ar ao redor de uma asa, uma vez que ela leva a conclusões não observáveis na prática. Uma das contradições mais importantes é o chamado paradoxo de d'Alembert's, descoberto pelo matemático francês Jean le Rond d'Alembert, que demonstrou que um corpo se movendo a velocidade constante através de um escoamento potencial não sofre nenhum tipo de arrasto, independentemente de sua forma. Essa previsão vai contra a observação experimental de que qualquer corpo se movendo através de um fluido convencional sofre uma força contrária ao seu movimento.

O paradoxo de d'Alembert foi explicado pelo engenheiro alemão Ludwig Prandtl (HOUGHTON *et al.*, 2012) e sua teoria de camada limite. Prandtl também desenvolveu uma modificacão (baseada em filamentos de vórtices) na teoria de escoamento potencial que permite a previsão do fluxo de ar ao redor de uma asa de alto alongamento: essa teoria é conhecida como linha sustentadora.

O método de *vortex lattice*, termo cunhado por Falkner em 1946 (DEYOUNG, 1976), é baseado na teoria de linha sustentadora e é descrito a seguir.

2.1.1 Método de *Vortex Lattice*

O método de *vortex lattice* é um método numérico baseado na teoria de linha de sustentação e que permite o cálculo 3D do escoamento ao redor de uma configuração qualquer de superfícies sustentadoras. A formulação aqui apresentada é baseada naquela apresentada por Drela (2014) e Katz e Plotkin (2001).

O método é composto de 5 etapas ao final das quais é possível obter as forças e momentos gerados pela interação entre o ar e as superfícies sustentadoras.

Essas etapas são:

1. Discretização da superfície em painéis com "vórtices ferradura"
2. Representação do campo de velocidades

3. Condição de tangência ou não-penetração nos pontos de controle
4. Construção e solução do sistema linear
5. Cálculo das forças aerodinâmicas

2.1.1.1 Passo 1: Discretização da superfície em painéis com vórtices ferradura

Para a aplicação do método de *vortex lattice* é primeiramente necessário discretizar as superfícies sustentadoras em painéis trapezoidais. A cada um desses painéis será atribuído um vórtice ferradura composto por três filamentos de vórtices: um filamento situado a um quarto da corda do painel e em contato com a superfície; e outros dois filamentos de esteira, paralelos ao eixo X se extendendo das extremidades do primeiro filamento até o infinito à jusante do escoamento (ver figura 2.1). Esses três filamentos tem todos a mesma intensidade de circulação.

Para cada um desses painéis são definidos 2 pontos principais: o centro aerodinâmico, situado a um quarto da corda e no centro da envergadura, e um ponto de controle, situado a três quartos da corda e no centro da envergadura.

Na implementação usada na ferramenta *Flying Circus* os painéis são definidos por quatro vértices. Embora esses vértices não precisem ser necessariamente coplanares, cuidado deve ser tomado na definição da malha aerodinâmica para que a superfície definida por eles possa ser aproximada por um plano.

A figura 2.1 mostra a discretização de uma asa em *vortex lattice* e os segmentos de vórtice utilizados.

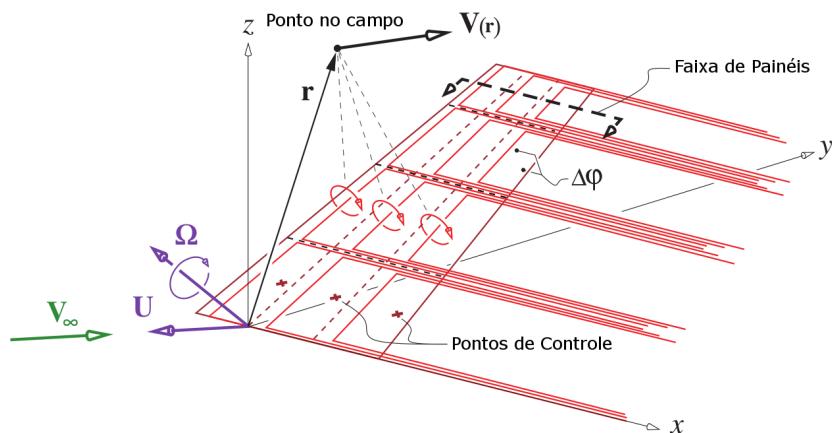


FIGURA 2.1 – Discretização de uma asa em *vortex lattice* (DRELA, 2014)

2.1.1.2 Passo 2: Representação do campo de velocidades

Uma vez que a rede de vórtices foi estabelecido agora é possível descrever o campo de velocidades aos redor da aeronave. A velocidade em qualquer ponto r é dada pela sobreposição da velocidade do ar em relação a aeronave (considerando suas velocidades de translação e de rotação) com a velocidade induzida naquele ponto por todos os vórtices farraduras criados. Drela (2014) apresenta a seguinte abordagem para o cálculo dessas velocidades.

Sendo U o vetor que define a velocidade de translação da aeronave e Ω o vetor que define a sua velocidade de rotação temos que a velocidade do ar em relação a aeronave é dada pela equação 2.1.

$$-(U + \Omega \times r) \quad (2.1)$$

A somatória das velocidades induzidas pelos vórtices farradura pode ser expressa pela somatória das integrais de Biot-Savart de cada um dos vórtices farradura conforme a equação 2.2, onde Γ é a intensidade do filamento de vórtice, e ℓ é o vetor que define o filamento de vórtice.

$$\sum \frac{\Gamma}{4\pi} \int \frac{d\ell' \times (r - r')}{|r - r'|^3} \quad (2.2)$$

Representando a integral por $\hat{V}_i(r)$ temos que a velocidade V em um ponto qualquer r é dada pela equação 2.3:

$$V(r) = \sum_{i=1}^N \Gamma_i \hat{V}_i(r) - (U + \Omega \times r) \quad (2.3)$$

A integral $\hat{V}_i(r)$ pode ser calculada utilizando a equação 2.4 (DRELA, 2014), com os vetores a e b representados na figura 2.2.

$$\hat{V}_i(r) = \frac{1}{4\pi} \left\{ \frac{a \times b}{|a||b| + a} \left(\frac{1}{|a|} + \frac{1}{|b|} \right) + \frac{a \times \hat{x}}{|a| - a \cdot \hat{x}} \frac{1}{|a|} - \frac{b \times \hat{x}}{|b| - b \cdot \hat{x}} \frac{1}{|b|} \right\} \quad (2.4)$$

Se a distância do filamento de vórtice até ponto para o qual se quer calcular a velocidade induzida é muito pequena ou igual a zero, a velocidade induzida calculada tenderá ao infinito, resultado que não condiz com a realidade e que traz problemas numéricos. Katz e Plotkin (KATZ; PLOTKIN, 2001) sugerem que caso a distância calculada seja menor que um parâmetro ϵ escolhido, a velocidade induzida pelo vórtice seja considerada igual a zero. Essa abordagem foi adotada na ferramenta *Flying Circus*.

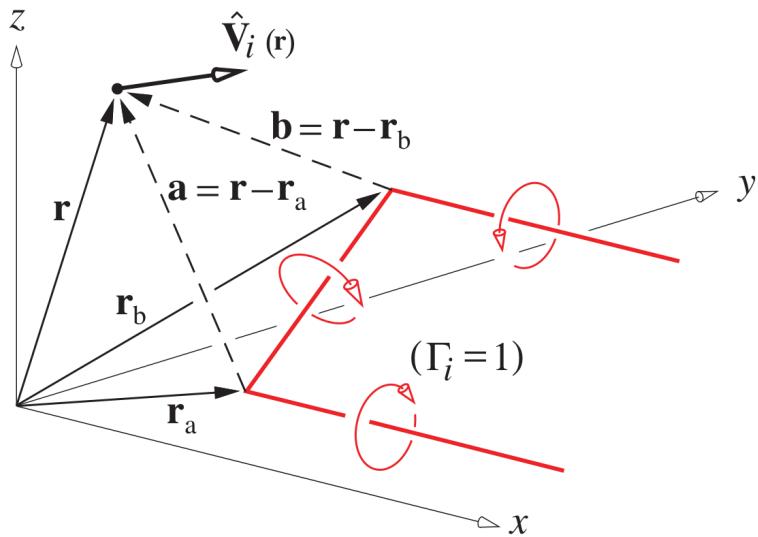


FIGURA 2.2 – Representação vetorial da velocidade induzida em um ponto por um vórtice farradura (DRELA, 2014)

2.1.1.3 Passo 3: Condição de tangência nos pontos de controle

Até agora os painéis foram representados apenas como vórtices farradura, o que significa que nada impede que o escoamento flua através dos painéis. O próximo passo é garantir a condição de tangência na superfície dos painéis, ou seja, garantir que a velocidade do escoamento na superfície dos painéis seja tangente à sua superfície.

Isso é feito a partir dos pontos de controle estabelecidos anteriormente. Sendo r_i^c o ponto de controle de um painel, a velocidade do encaimento nesse ponto é tangente à superfície do painel se o produto vetorial entre a velocidade do escoamento naquele ponto e o vetor normal à superfície do painel (n_{0i}) for igual a zero; essa condição é expressa na equação 2.5.

$$V(r_i^c) \cdot n_i = \left(\sum_{j=1}^N \Gamma_j \hat{V}_j(r_i^c) - (U + \Omega \times r) \right) \cdot n_{0i} = 0 \quad (2.5)$$

2.1.1.4 Passo 4: Construção e solução do sistema linear

Usando a equação 2.5 é possível construir um sistema linear onde as incógnitas são as intensidades da circulação dos vórtices farraduras de cada um dos painéis. A integral $\hat{V}_i(r)$ depende unicamente da geometria da rede de vórtices, assim como o vetor normal à superfície do painel (n_0), sendo assim é possível definir a matrix A_{ij} , chamada de matrix de coeficientes de influência aerodinâmica.

O sistema linear pode ser descrito pela equação 2.6.

$$\begin{bmatrix} A_{ij} \end{bmatrix} \left\{ \Gamma_i \right\} = (U + \Omega \times r) \cdot n_{0i} \quad (2.6)$$

Onde a matrix de coeficientes de influência aerodinâmica, A_{ij} , é dada pela equação 2.7.

$$A_{ij} = \hat{V}_j(r_i^c) \cdot n_{0i} \quad (2.7)$$

2.1.1.5 Passo 5: Cálculo das forças aerodinâmicas

Uma vez calculadas as intensidades de circulação de cada um dos vórtices ferradura é possível calcular a força aerodinâmica agindo em cada um dos painéis usando a equação 2.8, onde ρ é a densidade do ar, ℓ_i é o vetor que define o segmento de vórtice em contato com a superfície, V_i é o vetor que define a velocidade do ar no centro aerodinâmico do painel, e Γ_i é a intensidade de circulação do painel calculada anteriormente. Essas grandezas estão representadas na figura 2.3.

$$F_i = \rho \Gamma_i (V_i \times \ell_i) \quad (2.8)$$

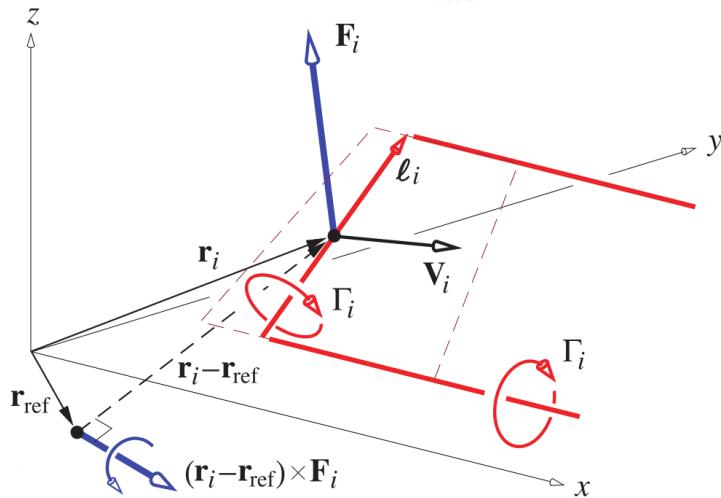


FIGURA 2.3 – Força aerodinâmica agindo em um painel (DRELA, 2014)

Com a força F_i e a área do painel é possível calcular a variação de pressão entre o extradorso e o intradorso do painel. Somando as forças de todos os painéis é possível calcular a somatória de forças e momentos no centro de gravidade da aeronave sendo estudada, ou em qualquer outro ponto.

2.2 Estruturas

A estrutura da aeronave é representada por uma série de vigas conectadas por engastes. Uma viga é um elemento estrutural longo e delgado geralmente submetido a um carregamento transversal que produz efeitos de flexão bem mais significativos que efeitos de torsão ou deformação axial (LOGAN, 2011). Para o cálculo das deformações da estrutura é utilizado o método dos elementos finitos com, elementos de viga baseados na teoria de Euler-Bernoulli.

2.2.1 Método dos Elementos Finitos

Problemas de interesse da engenharia como análise estrutural, escoamento de fluidos, transferência de calor ou potencial eletromagnético geralmente envolvem a solução de equações diferenciais relacionando geometrias complexas, propriedades de materiais, carregamentos complicados e vários outros parâmetros dependentes da natureza do problema a ser resolvido.

A solução analítica dessas equações é muito difícil e por conta disso foram desenvolvidos métodos numéricos que, através do uso de computadores, permitem encontrar soluções aproximadas cuja precisão é suficiente para a maioria das aplicações. O método dos elementos finitos é um desses métodos.

O método dos elementos finitos é baseado na representação de um corpo por um sistema de corpos menores (elementos finitos) equivalentes interconectados por pontos (nós) e restringidos por condições de contorno. As equações do problema são formuladas para cada elemento finito e depois combinadas de forma a obter a solução para o corpo todo. Para os problemas de análise estrutural estática esse processo permite que a solução do problema possa ser obtida pela solução de um sistema de equações algébricas em vez das equações diferenciais iniciais.

Segundo Logan (2011) o método dos elementos finitos geralmente segue 8 passos, aqui resumidos para um caso de análises estrutural.

1. Discretize e selecione os tipos de elementos
2. Selecione uma função de deslocamento
3. Defina as relações de deformação/deslocamento e tensão/deformação
4. Derive a matrix de rigidez e equações dos elementos
5. Combine as equações dos elementos para obter as equações globais e introduza as condições de contorno

6. Resolva o sistema de equações e encontre os valores dos deslocamentos
7. Calcule as tensões e deformações dos elementos
8. Interprete os resultados

No trabalho aqui desenvolvido não foi necessário executar o passo número 7, uma vez que apenas os deslocamentos dos elementos já eram suficientes para atingir os objetivos da ferramenta *Flying Circus*.

2.2.1.1 Formulação Linear de Elementos de Viga Baseado na Teoria de Euler-Bernoulli

O método dos elementos finitos é exemplificado a seguir usando o caso bidimensional de um viga, um processo análogo é utilizado no caso tridimensional.

Primeiramente a viga é discretizada em um número finito de elementos, nesse caso três. Cada um dos elementos é composto por dois nós, cada um deles com dois graus de liberdade: v , para o movimento de translação no sentido transversal; Φ , para o movimento de rotação. Essa discretização é apresentada na figura 2.4.

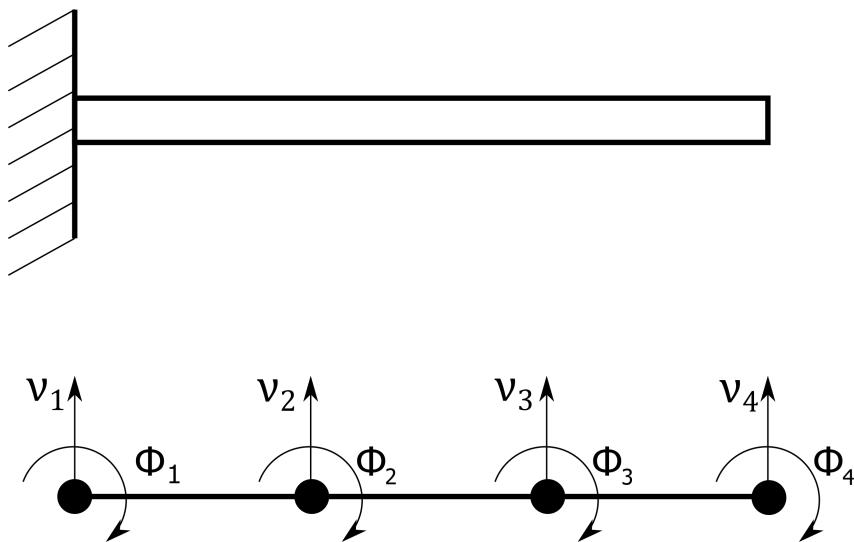


FIGURA 2.4 – Discretização de uma viga em elementos finitos

Utilizando a teoria de Euler-Bernoulli é possível escrever uma matriz de rigidez para cada um dos elementos de viga. A matriz de rigidez (k') para o elemento utilizado nesse

exemplo é dado pela equação 2.9 (LOGAN, 2011).

$$\mathbf{k}' = \frac{EI}{L^3} \begin{bmatrix} 12 & 6L & -12 & 6L \\ 6L & 4L^2 & -6L & 2L^2 \\ -12 & -6L & 12 & -6L \\ 6L & 2L^2 & -6L & 4L^2 \end{bmatrix} \quad (2.9)$$

Utilizando as matrizes de rigidez k'_n de cada um dos elementos e a relação entre esses elemento e os nós que os constituem, é possível construir a matriz de rigidez k global do sistema. Esse processo está representado na figura 2.5. O mesmo processo pode ser utilizado no caso tridimensional.

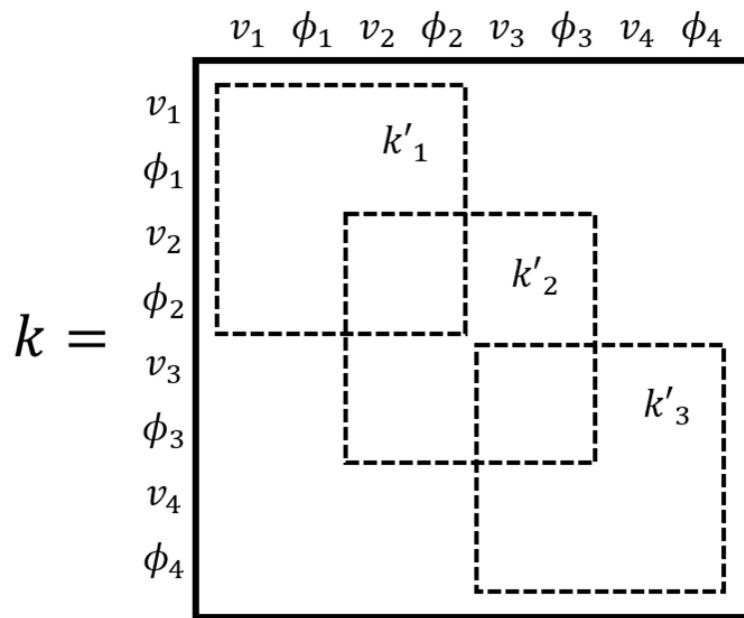


FIGURA 2.5 – Construção da matriz de rigidez global do sistema

Logan (2011) fornece uma formulação do método dos elementos finitos para o caso de vigas orientadas arbitrariamente no espaço, cuja aplicação (implementação e resultados) a este trabalho é abordada nessa seção.

O elemento utilizado é uma viga composta por dois nós, cada um em uma extremidade. Cada nó possui 6 graus de liberdade: 3 de translação nos sentidos X' , Y' e Z' no sistema de coordenadas local do elemento; 3 de rotação nos eixos X' , Y' e Z' , totalizando 12 graus de liberdade por elemento. A figura 2.6 ilustra o elemento utilizado neste trabalho.

É utilizado um polinômio de terceiro grau para aproximar os deslocamentos dentro de cada elemento; as relações de deformação/deslocamento e tensão/deformação são deduzidas a partir da teoria de Euler-Bernoulli para vigas. As matrizes de rigidez são derivadas

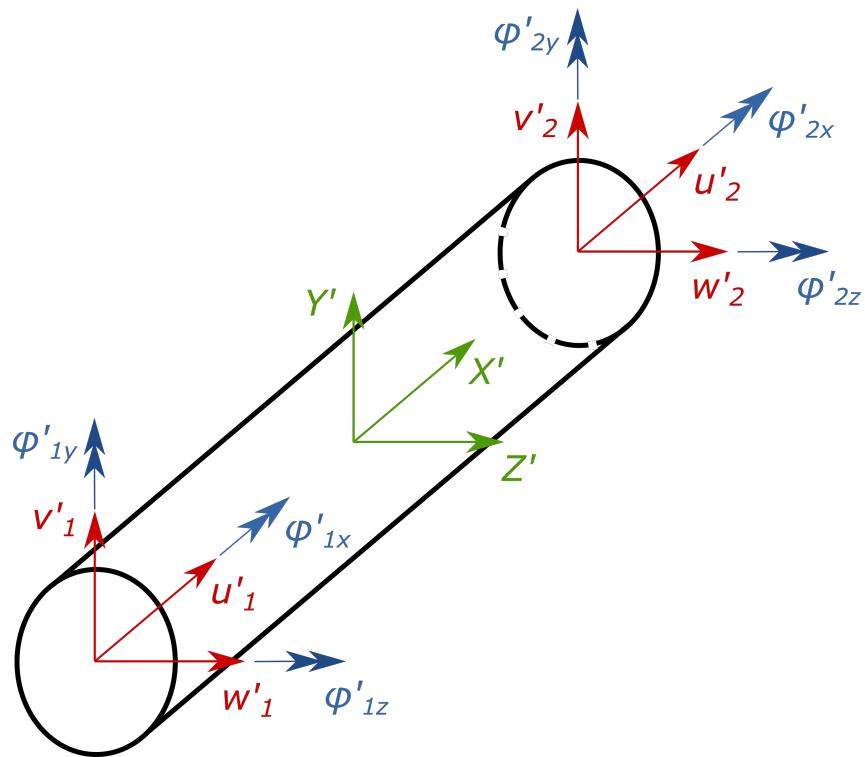


FIGURA 2.6 – Elemento finito de viga utilizado

a partir de um método de equilíbrio direto.

Logan (2011) fornece a equação 2.10 para a matrix de rigidez no sistema de coor-

nadas local do elemento de viga (k').

$$k' = \begin{bmatrix} u'_1 & v'_1 & w'_1 & \phi'_{1x} & \phi'_{1y} & \phi'_{1z} & u'_2 & v'_2 & w'_2 & \phi'_{2x} & \phi'_{2y} & \phi'_{2z} \\ \frac{AE}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{AE}{L} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} & 0 & -\frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} \\ 0 & 0 & \frac{12EI_y}{L^3} & 0 & -\frac{6EI_y}{L^2} & 0 & 0 & 0 & -\frac{12EI_y}{L^3} & 0 & -\frac{6EI_y}{L^2} & 0 \\ 0 & 0 & 0 & \frac{GJ}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{GJ}{L} & 0 & 0 \\ 0 & 0 & -\frac{6EI_y}{L^2} & 0 & \frac{4EI_y}{L} & 0 & 0 & 0 & \frac{6EI_y}{L^2} & 0 & \frac{2EI_y}{L} & 0 \\ 0 & \frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{4EI_z}{L} & 0 & -\frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{2EI_z}{L} \\ \hline -\frac{AE}{L} & 0 & 0 & 0 & 0 & 0 & \frac{AE}{L} & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{12EI_z}{L^3} & 0 & 0 & 0 & -\frac{6EI_z}{L^2} & 0 & \frac{12EI_z}{L^3} & 0 & 0 & 0 & -\frac{6EI_z}{L^2} \\ 0 & 0 & -\frac{12EI_y}{L^3} & 0 & \frac{6EI_y}{L^2} & 0 & 0 & 0 & \frac{12EI_y}{L^3} & 0 & \frac{6EI_y}{L^2} & 0 \\ 0 & 0 & 0 & -\frac{GJ}{L} & 0 & 0 & 0 & 0 & 0 & \frac{GJ}{L} & 0 & 0 \\ 0 & 0 & -\frac{6EI_y}{L^2} & 0 & \frac{2EI_y}{L} & 0 & 0 & 0 & \frac{6EI_y}{L^2} & 0 & \frac{4EI_y}{L} & 0 \\ 0 & \frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{2EI_z}{L} & 0 & -\frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{4EI_z}{L} \end{bmatrix} \quad (2.10)$$

A matriz de rigidez do elemento no sistema de coordenadas global (k) pode ser calculada pela equação 2.11, onde T é a matriz de rotação dada pela equação 2.12 onde $C_{pq'}$ são os cossenos diretores que relacionam o sistema de coordenadas local do elemento ao sistema de coordenadas global do sistema como é mostrado na figura 2.7.

$$k = T^T * k' * T \quad (2.11)$$

$$\begin{bmatrix} C_{xx'} & C_{yx'} & C_{zx'} \\ C_{xy'} & C_{yy'} & C_{zy'} \\ C_{xz'} & C_{yz'} & C_{zz'} \end{bmatrix} \quad (2.12)$$

Uma vez calculadas as matrizes k de cada elemento é possível montar a matriz K

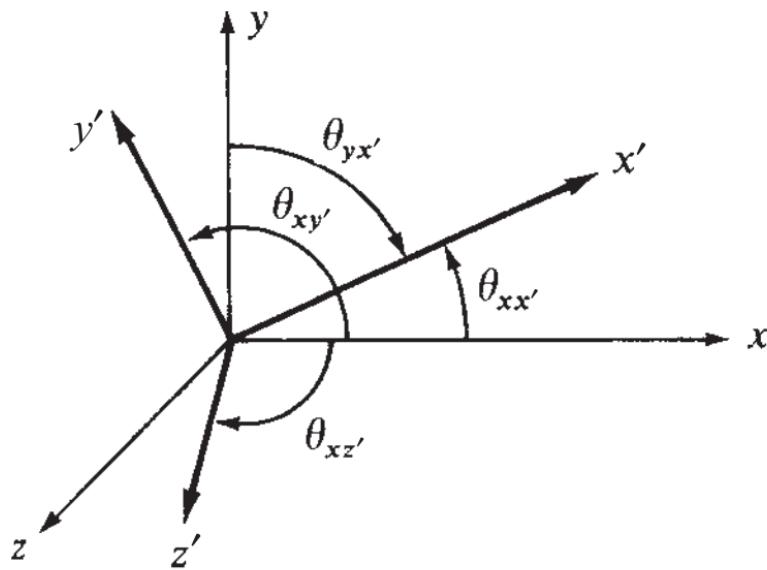


FIGURA 2.7 – Cossenos diretores (LOGAN, 2011)

global do sistema ao relacionar os graus de liberdade de cada elemento com os graus de liberdade globais do sistema e sobrepor os termos das matrizes locais à matrix global.

O sistema de equações global pode então ser construído, ele é dado pela equação 2.13 onde K é a matriz de rigidez global, F é o vetor de carga nodais e x são as deformações nos graus de liberdade.

$$[K][x] = [F] \quad (2.13)$$

A aplicação das condições de contorno permite eliminar linhas e colunas do sistema de equações global; a solução do sistema resultante fornece as deformações nodais de cada um dos nós da estrutura. Essas deformações são a informação necessária para o funcionamento da ferramenta aqui desenvolvida.

2.3 Aeroelasticidade

Aeroelasticidade é a área da engenharia dedicada ao estudo da interação entre as forças aerodinâmicas, elásticas e iniciais resultantes das deformações sofridas por uma estrutura flexível se movendo através de um fluxo de ar.

A figura 2.8, conhecida como *Diagrama de Collar* por ter sido criada por A. R. Collar por volta de 1940 (HODGES; PIERCE, 2009), representa a natureza interdisciplinar da área de aeroelasticidade. A aerodinâmica permite o cálculo das forças agindo sobre o

corpo, a elasticidade prevê as deformações sofridas por esse corpo devido às cargas aplicadas, e a dinâmica inclui os efeitos das forças iniciais. A interação entre aerodinâmica e dinâmica é o foco de estudo da disciplina de mecânica de voo enquanto a disciplina de dinâmica estrutural investiga os efeitos da combinação dos efeitos dinâmicos com a elasticidade. A disciplina de aeroelasticidade estuda os fenômenos frutos da interação entre elasticidade e aerodinâmica: aqueles em que os efeitos dinâmicos são desprezíveis inserem-se no que se denomina aeroelasticidade estática; os demais, classificam-se como fenômenos de aeroelasticidade dinâmica.

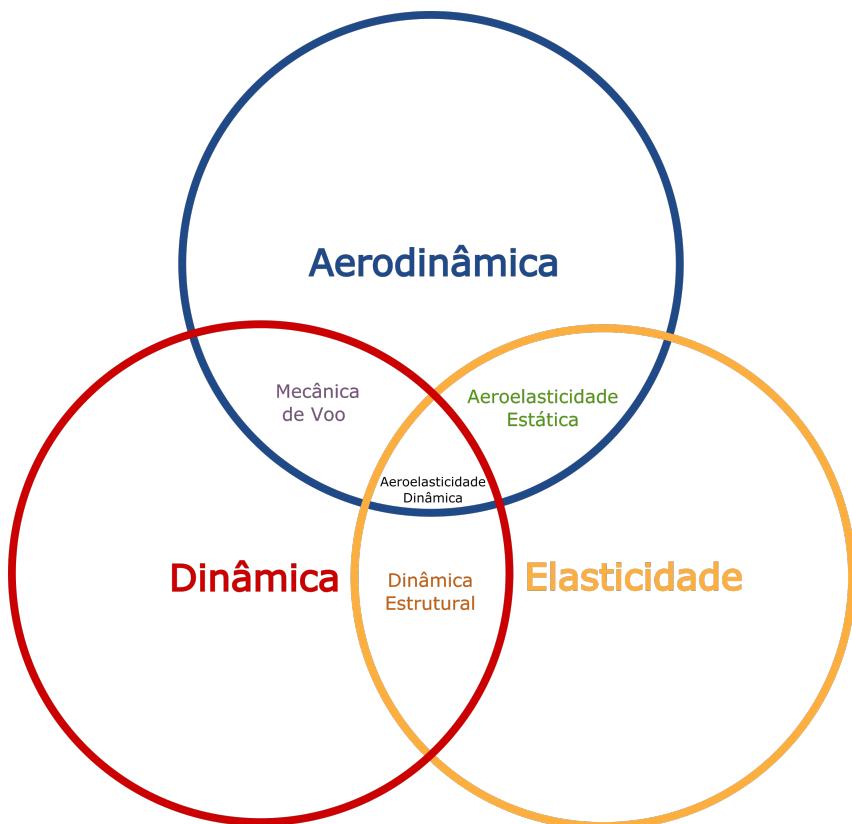


FIGURA 2.8 – Diagrama mostrando a interação entre as disciplinas envolvidas no estudo de aeroelasticidade

A aeroelasticidade estática considera os efeitos não-oscilatórios da influência da flexibilidade de estruturas aeronáutica no carregamento aerodinâmico e nas características de voo da aeronave, seu coeficiente de arrasto por exemplo.

A aeroelasticidade dinâmica é focada em fenômenos de natureza oscilatória, cujo representante mais famoso (devido à sua natureza catastrófica) é conhecido como *flutter*. *Flutter* é uma instabilidade gerada pelo acoplamento de forças aerodinâmicas, elásticas e iniciais que permitem a estrutura da aeronave retirar energia do escoamento (WRIGHT; COOPER, 2014). O estudo desses fenômenos quando efeitos da interação com o sistema de controle são incluídos é conhecido como aeroservoelasticidade.

O presente trabalho trata apenas dos efeitos de aeroelasticidade estática.

2.3.1 Aeroelasticidade Estática

Os fenômenos de aeroelasticidade estática são caracterizados por serem insensíveis às as velocidades e acelerações das deformações estruturais. Em geral pode-se dividir as consequências desses fenômenos para o projeto de aeronaves em duas classes (HODGES; PIERCE, 2009):

A primeira é o efeito das deformações elásticas nas cargas aerodinâmicas aplicadas a aeronave em condições normais de operação. Esses efeitos afetam o desempenho, a estabilidade, a distribuição de cargas estruturais e o controle de qualquer aeronave em diferentes graus de severidade, a depender das características geométricas e estruturais da aeronave em questão.

A segunda classe é conhecida como divergência e envolve a instabilidade estática de superfícies sustentadoras com consequências potencialmente catastróficas. A divergência acontece quando a deformação causada pelas cargas aerodinâmicas resulta em um aumento retroalimentado dessas mesmas cargas, o qual leva ao um aumento ilimitado da deformação até o ponto de falha estrutura. A interação entre forças aerodinâmicas e elásticas leva a uma perda de rigidez efetiva da estrutura (HODGES; PIERCE, 2009), o que pode acarretar em limitações no envelope operacional da aeronave.

Esse trabalho concentra-se em estudar a primeira classe de fenômenos citada, em particular no que diz respeito à redistribuição de cargas estruturais e seu efeito nas características de estabilidade da aeronave. Para tanto, a seção a seguir contempla os fundamentos da formulação aeroelástica estática empregada neste trabalho.

2.3.1.1 Formulação Aeroelástica Estática Bidimensional

Esta seção descreve a formulação de procedimento iterativo para cálculo de deformação aeroelástica estática (WRIGHT; COOPER, 2014); embora apresente um caso bidimensional (2D) os conceitos apresentados são análogos àqueles utilizados para os casos tridimensionais (3D) abordados.

Considere o aerofólio 2D mostrado na figura 2.9 com envergadura unitária e corda c . O aerofólio é simétrico e está acoplado a uma mola torcional de rigidez K_θ a uma distância ec atrás do seu centro aerodinâmico situado a um quarto de corda. O aerofólio tem um ângulo de incidência inicial de θ_0 e é rotacionado por um ângulo desconhecido θ devido à carga aerodinâmica. A curva C_l versus α do aerofólio tem inclinação igual a a_1 .

A força de sustentação agindo no centro aerodinâmico para θ_0 , velocidade do ar V (TAS), e densidade ρ do ar gera um momento no ponto de fixação dado pela expressão

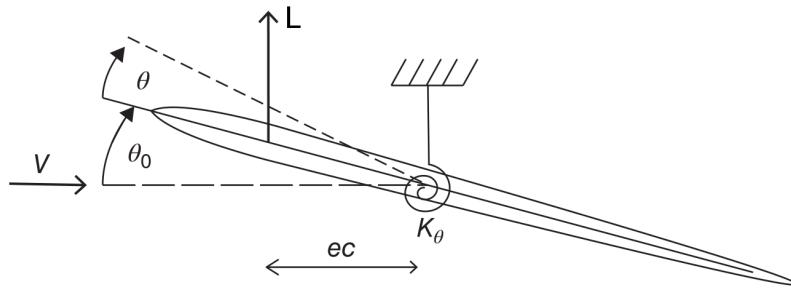


FIGURA 2.9 – Diagrama de um aerofólio 2D

2.14, onde q é a pressão dinâmica.

$$M = \left(\frac{1}{2} \rho V^2 c a_1 \theta_0 \right) ec = \frac{1}{2} \rho V^2 e c^2 a_1 \theta_0 = q e c^2 a_1 \theta_0 \quad (2.14)$$

Podemos então calcular qual a deflexão θ causada pelo momento gerado usando a equação 2.15, onde $R = e c^2 a_1 / K_\theta$

$$M = K_\theta = q e c^2 a_1 \theta_0; \text{ portanto; } \theta = \frac{q e c^2 a_1}{K_\theta} \theta_0 = q R \theta_0 \quad (2.15)$$

O momento aerodinâmico gera uma deflexão θ a qual aumenta o ângulo de ataque do aerofólio o que gera um aumento no momento aerodinâmico. Calculando o novo momento usando o novo ângulo de incidência $\theta_0 + q R \theta_0$ temos a equação 2.16.

$$M = q e c^2 a_1 (\theta_0 + q R \theta_0) \quad (2.16)$$

Temos a expressão 2.17 para o novo θ .

$$\theta = q e c^2 a_1 \left(\frac{1 + q R}{K_\theta} \right) = q R (1 + q R) \theta_0 \quad (2.17)$$

Prosseguindo com as iterações temos que o ângulo θ final é dado pela série infinita expressa em 2.18. O valor de θ pode ser aproximado usando tantos termos quanto for necessário.

$$\theta = q R [1 + q R + (q R)^2 + (q R)^3 + (q R)^4 + \dots] \theta_0 \quad (2.18)$$

Se $|q R| \leq 1$ a expressão para a soma dos infinitos termos é dada pela equação 2.19

(WRIGHT; COOPER, 2014).

$$\theta = \frac{qR}{1 - qR} \theta_0 \quad (2.19)$$

Um método análogo é utilizado nesse trabalho para o cálculo da deformação aeroelástica da estrutura tridimensional.

As forças de sustentação e os momentos (M e L) gerados por ela são calculados usando o método de *vortex lattice*, a rigidez (K_θ) e a deformação θ são calculadas para a estrutura usando o método dos elementos finitos, e são utilizadas para deformar a malha de vórtices. As forças e momentos aerodinâmicos são atualizados iterativamente até que a deformação da estrutura entre uma iteração e outra atenda a um critério de convergência.

3 Metodologia

A ferramenta foi desenvolvida utilizando a linguagem de programação *Python* (PYTHON-SOFTWARE-FOUNDATION, 2019), essa escolha se deve aos seguintes fatores:

- *Python* é uma linguagem de código aberto e disponível gratuitamente, sendo assim seu uso não é limitado pelo alto custo da compra de licenças como acontece com outras linguagens de programação equivalentes como *MATLAB* ou *Mathematica*.
- *Python* é uma linguagem de alto nível desenvolvida visando grande legibilidade, essas características permitem com que um usuário sem um conhecimento aprofundado de programação possa rapidamente entender códigos já desenvolvidos e criar suas próprias ferramentas
- *Python* é amplamente utilizada no meio científico e por conta disso existem diversas bibliotecas que fornecem funções de grande utilidade para aplicações de engenharia. Entre elas destacam-se: *Numpy* (NUMPY-DEVELOPERS, 2019), cálculos e manipulação de variáveis em formato matricial; *Scipy* (SCIPY-DEVELOPERS, 2019), implementações de diversos métodos numéricos como integradores, soluções de sistemas lineares e funções estatísticas; *Matplotlib* (MATPLOTLIB-DEVELOPMENT-TEAM, 2019), criação de gráficos 2D e 3D; *Numba* (ANACONDA, 2019), biblioteca que compila o código *python* em tempo real e possibilita níveis de desempenho próximo aos de linguagens pré-compiladas como C e FORTRAN: *pyquaternion* (WYNN, 2019), para representação de quaternions, usada para representar rotações no espaço 3D.

Para os cálculos aerodinâmicos o método escolhido foi o de *vortex lattice*, uma vez que esse método permite a simulação de uma grande variedade de geometrias, tem um custo computacional baixo e fornece resultados cujo grau de precisão é compatível com o de projeto conceitual onde metodologias ainda mais simplificadas, como interpolação de dados históricos, são comumente utilizadas.

A estrutura da asa será representada por uma viga de Euler-Bernoulli, cuja deformação será calculada através do método de elementos finitos. Essa modelagem, também

conhecida como *stick model*, é amplamente aplicada na literatura e na industria uma vez que permite a obtenção de resultados com uma precisão aceitável com um custo computacional baixo e sem demandar um conhecimento detalhado sobre a geometria da estrutura.

A ferramenta *Flying Circus* foi estruturada como um *package Python* contendo vários *sub-packages* cada um deles responsável por um aspecto dos cálculos necessários para a obtenção do resultado desejado, ou pós-processamento dos dados obtidos.

Os sub-pacotes que compõem o pacote *Flying Circus* são: *geometry*, *aerodynamics*, *structures*, *aeroelasticity*, *loads*, e *visualization*.

O pacote *Flying Circus* foi projetado para que, fazendo uso das funções e métodos implementados, o usuário possa construir seu próprio fluxo de trabalho. Embora existam funções que são capazes de realizar todas as etapas necessárias para o uso de todas as capacidades da ferramenta elas são fornecidas como facilitadoras e exemplos de uso, não como forma única de utilização.

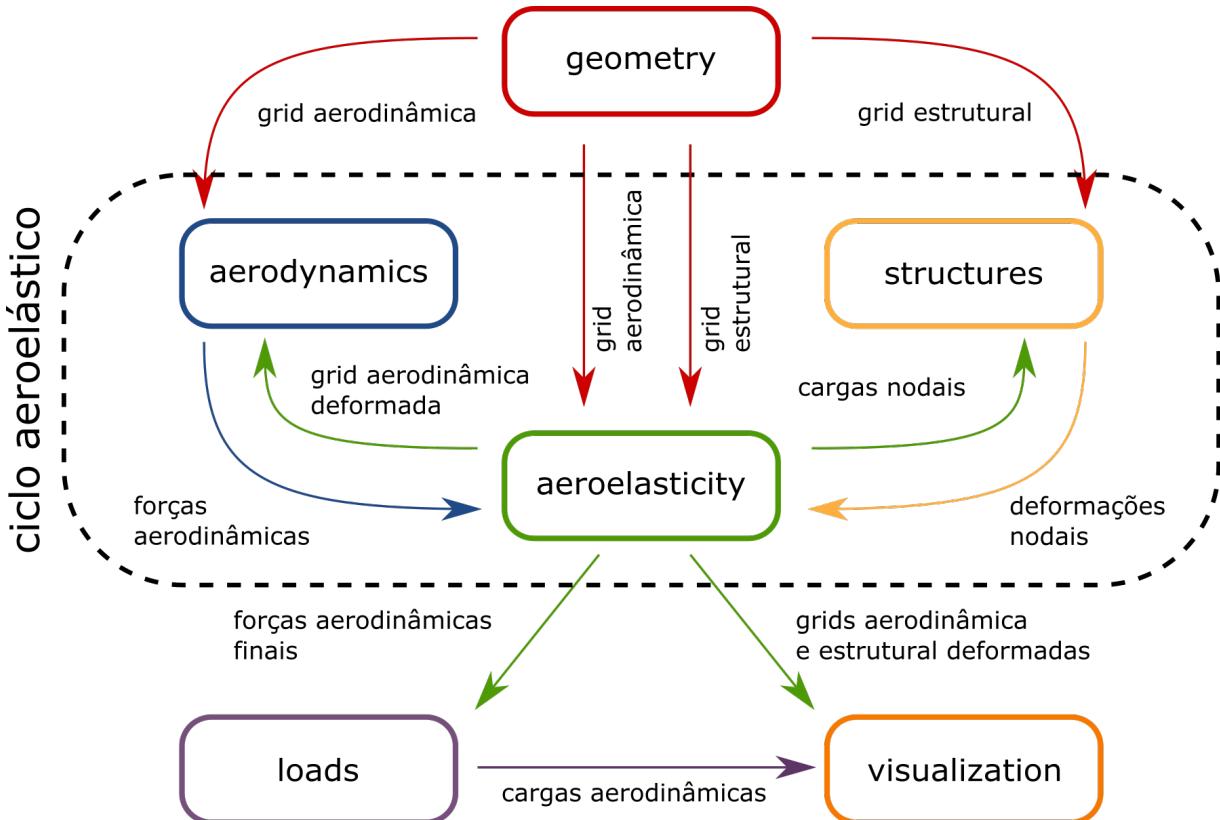
Essa estrutura modular tem duas grandes vantagens, a primeira delas é que a expansão das capacidades da ferramenta e sua integração com outros programas e fluxos de trabalho é muito simples. Por exemplo, desde que formatadas da forma adequada as entradas para o pacote estrutural podem ter sua origem no pacote aerodinâmico, em um software comercial ou até mesmo em dados experimentais.

A segunda vantagem é que o usuário pode utilizar apenas as rotinas que lhe interessam, economizando tempo de processamento. Por exemplo, um usuário que não esteja interessado nos efeitos da flexibilidade da estrutura pode utilizar o pacote aerodinâmico para suas análises sem desperdiçar capacidade computacional com as iterações necessárias para o acoplamento aeroelástico.

A figura 3.1 mostra de forma esquematizada o fluxo de informações dentro da ferramenta desenvolvida.

De forma resumida os fluxo de informação funciona da seguinte forma:

1. O usuário entra com as informações geométrica necessárias usando o subpacote *geometry*, com essa informações e configurações adicionais fornecidas pelo usuário são geradas *grids* aerodinâmicas e estruturais.
2. Usando a *grid* aerodinâmica o subpacote *aerodynamics* calcula as forças aerodinâmicas.
3. Usando as grids aerodinâmicas e estruturais o subpacote *aeroelasticity* calcula qual a carga nodal a ser aplicada em cada um dos nós estruturais.
4. Usando as cargas nodais e a grid estrutural o subpacote *structures* calcula as deformações em cada um dos nós estruturais.

FIGURA 3.1 – Fluxo de Informação entre os *subpackages* da Ferramenta

5. Usando as deformações nodais o subpacote *aeroelasticity* deforma a grid aerodinâmica e a envia para o subpacote *aerodynamics* para que as forças aerodinâmicas sejam recalculadas. Esse ciclo se repete até que a convergência das deformações seja alcançada.
6. Usando as forças aerodinâmicas finais o subpacote *loads* calcula a somatória das forças agindo na aeronave e sua distribuição ao longo da envergadura das superfícies.
7. Usando as grids deformadas e as forças calculadas o subpacote *visualization* gera vários tipos de visualizações para pós processamento dos resultados.

O restante do capítulo descreve a função de cada um dos subpacotes e descreve as metodologias adotadas para a realização das diversas análises. Para a descrição completa de todas as funções e objetos o código completo da ferramenta está disponibilizada no anexo A e na referência (COSTA, 2019).

3.1 Subpacote *geometry*

O subpacote *geometry* fornece objetos com os quais é possível descrever uma grande variedade de aeronaves. Utilizando os objetos fornecidos o usuário descreve a aeronave e

logo em seguida gera as malhas aerodinâmicas e estruturais necessárias para a realização das análises subsequentes.

Os principais objetos são:

surface Cria uma superfície aerodinâmica usando como entradas, comprimento, corda da raiz, corda da ponta, ângulo de diedro, ângulo de enflechamento do bordo de ataque, torção geométrica da ponta em relação a raiz, e posição da dobradiça da superfície de controle, se ela existir.

macrosurface Objeto que agrupa várias *surfaces* para gerar uma superfície mais complexa, como uma asa ou empennagem. Tem como entradas principais a posição do bordo de ataque da raiz, uma lista com as *surfaces* que ela contém e seu ângulo de incidência em relação ao sistema de coordenadas da aeronave.

beam Objeto que define uma viga, usado para modelagem de fuselagens.

aircraft Objeto que contém toda a informação fornecida sobre a aeronave.

panel Objeto que descreve um painel aerodinâmico, e calcula todas as suas características. É usado pelo método de *vortex lattice*.

O sistema de coordenadas utilizado pelo subpacote *geometry* é orientado da seguinte forma: o eixo *X* é orientado do nariz da aeronave em direção a sua cauda, o eixo *Y* é perpendicular ao eixo *X* e é positivo ao longo da asa direita, o eixo *Z* é ortogonal ao plano *XY* e positivo no sentido para a parte superior da aeronave. A figura 3.2 ilustra o sistema de coordenadas utilizado.

Produtos escalares e vetoriais são muito utilizados durante todas as etapas da geração da geometria, para tornar o código mais eficiente foi utilizada um pacote de funções fornecido por Fechner (2019) baseado no pacote *Numba*.

3.2 Subpacote *aerodynamics*

O subpacote *aerodynamics* tem como função calcular as cargas aerodinâmicas aplicadas nas superfícies sustentadoras da aeronave, baseado em sua geometria, condições atmosféricas e atitude em relação ao escoamento.

Para o cálculo aerodinâmico é utilizado o método de *vortex lattice*, descrito anteriormente. O método foi implementado com base no equacionamento descrito por Drela (2014) e Katz e Plotkin (2001) com pequenas modificações para melhor adequá-lo à estrutura do programa desenvolvido.

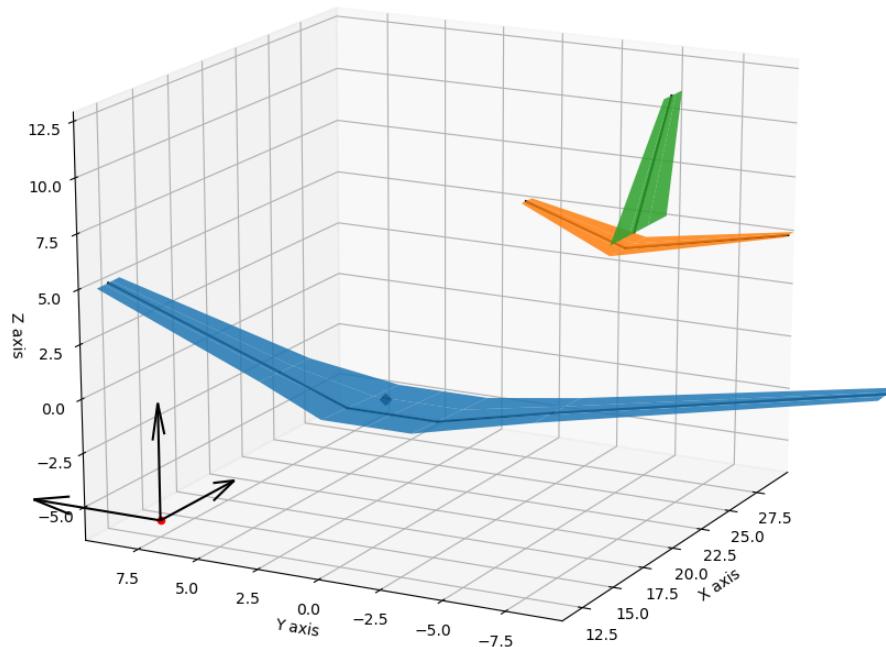


FIGURA 3.2 – Sistema de coordenadas utilizado pelo pacote *geometry* juntamente com o modelo de uma aeronave comercial

O subpacote *aerodynamics* é composto por três módulos, a descrição de suas principais funções é feita a seguir.

3.2.1 Módulo *functions*

O módulo *functions* realiza três funções: calcular a velocidade induzida por um vórtice ferradura, calcular a força gerada em uma superfície ligada ao vórtice ferradura e calcular as propriedades da atmosfera para uma determinada altitude usando o modelo de atmosfera padrão.

A velocidade induzida por um vórtice ferradura é calculada pela função `horse_shoe_-ind_vel`. Essa função define um vórtice ferradura usando um segmento de vórtice ligando dois pontos e dois segmentos de vórtice se estendendo até o infinito (esteira) alinhados com o eixo *X*, dado um valor de circulação e a localização de um ponto a função calcula os componentes *X*, *Y* e *Z* da velocidade induzida pelo vórtice ferradura nesse ponto.

Um segmento de vórtice é singular, ou seja ele tem um raio de valor igual a zero. Essa natureza gera problemas para o cálculo da velocidade induzida em pontos muito próximos do vórtice uma vez que a velocidade calculada tende ao infinito a medida que o ponto se aproxima do vórtice. Esse fenômeno é inconsistente com a realidade do escoamento e para evitar os problemas que podem vir a acontecer é definido um parâmetro `vortex_radius`. Caso a distância de um ponto até o vórtice seja menor do que o valor de `vortex_radius` a velocidade induzida pelo vórtice no ponto é considerada igual a zero.

A força gerada por um vórtice ferradura em uma superfície a ele conectada é calculada pela função `horse_shoe_aero_force`, essa força depende da intensidade do vórtice (circulação), da densidade do ar e do vetor de velocidade do escoamento local no vórtice.

As condições atmosféricas para uma determinada altitude são calculadas usando uma versão modificada do código *atmos* (CARMICHAEL, 2018).

3.2.2 Módulo *objects*

O módulo *objects* implementa o objeto `PanelHorseShoe`, esse objeto tem como base o objeto `Panel` do *subpackage geometry*. O objeto criado calcula a posição dos segmentos de vórtice usando as propriedades geométricas do painel e fornece métodos para calcular a velocidade induzida e a força gerada pelo vórtice ferradura.

3.2.3 Módulo *vlm*

O módulo *vlm* implementa o método de vórtice lattice usando as funções e objetos dos módulos anteriores. Sua função principal é a `aero_loads` a qual usando como entradas a malha aerodinâmica, o vetor de velocidades, o vetor de velocidades angulares, os ângulos de atitude da aeronave em relação ao escoamento, a altitude de voo e o centro de rotação da aeronave (em geral o seu centro de gravidade) calcula a intensidade dos vórtices ferradura de cada um dos painéis e a força gerada pelo vórtice em cada um de seus respectivos painéis.

3.3 Subpacote *structures*

O subpacote *structures* fornece os objetos e funções necessários para o cálculo das deformações sofridas pela estrutura utilizando o método dos elementos finitos. O subpacote tem como entrada as cargas aplicadas à estrutura e como saída as deflexões e rotações sofridas pelos nós com os quais a estrutura foi discretizada.

Seu principal módulo é o *fem* que implementa o método dos elementos finitos usando

elementos lineares de viga de Euler-Bernoulli em 3 dimensões, é seguida a formulação fornecida por Logan (2011) descrita anteriormente.

Para a solução dos sistemas lineares, necessária para o método dos elementos finitos, foram utilizadas as funções fornecidas pelo pacote *Scipy*.

3.4 Subpacote *aeroelasticity*

O subpacote *aeroelasticity* utiliza os subpacotes *aerodynamics* e *structures* para realizar o cálculo das cargas aerodinâmicas sofridas pela asa considerando as deflexões por ela sofrida devido ao carregamento aerodinâmico, em outras palavras, ele considera os efeitos de aeroelasticidade estática no cálculo aerodinâmico.

O cálculo realizado não considera efeitos dinâmicos, é considerado que a estrutura se ajusta de forma instantânea ao carregamento, ou seja, a abordagem adotada pode ser considerada do tipo quasi-estático.

Para o cálculo da deformação final da estrutura é adotado um método iterativo, para a transferência das cargas aerodinâmicas para a estrutura e das deformações da estrutura para os painéis aerodinâmicos é adotado um critério de proximidade.

A abordagem utilizada segue o seguinte algoritmo:

1. Calcular cargas aerodinâmicas geradas pela geometria da aeronave
2. Aplicar as cargas calculadas na estrutura da aeronave
3. Calcular a deformação da estrutura decorrente das cargas aerodinâmicas
4. Deformar a geometria da aeronave de acordo com a deformação da estrutura
5. Repetir passos de 1 a 4 até que a deformação da estrutura entre uma iteração e outra atinja o critério de convergência escolhido

A seguir a metodologia adotada em cada um dos passos é explicada em mais detalhes.

3.4.1 Passo 1: Cálculo das cargas aerodinâmicas

O cálculo das cargas aerodinâmicas é realizado pelo subpacote *aerodynamics* tomando como input a geometria da aeronave e as condições de voo.

O método de *vortex lattice* utilizado fornece como resultado a intensidade da circulação em cada um dos vórtices ferraduras dos painéis aerodinâmicos, tendo essa informação

é possível calcular a força aerodinâmica no centro aerodinâmico, situado a 25% da corda, de cada um dos painéis.

3.4.2 Passo 2: Aplicação das cargas aerodinâmicas na estrutura

A aplicação das das cargas aerodinâmicas na estrutura é feita da seguinte maneira:

1. Para cada um dos painéis aerodinâmicos é identificado o nó do modelo estrutural que se encontra mais próximo do seu centro aerodinâmico.
2. Os componentes X , Y e Z das forças e momentos aerodinâmicos calculados são aplicados a esse nó estrutural.

Adotando essa metodologia os resultados obtidos são equivalentes a conectar cada um dos centros aerodinâmicos do modelo ao nó estrutural mais próximo com um elemento rígido.

3.4.3 Passo 3: Cálculo da deformação da estrutura

As deformações (translação e rotação) de cada um dos nós da estrutura de vigas é calculada pelo subpacote *structures* usando o método dos elementos finitos.

3.4.4 Passo 4: Deformação da geometria da aeronave

Para a transferência das deformações do modelo de viga para os painéis aerodinâmicos é adotada uma metodologia similar à apresentada anteriormente.

1. Cada um dos painéis aerodinâmicos é definido por quatro vértices. Para cada um desses vértices é identificado o nó estrutural mais próximo.
2. Cada um dos vértices é translacionado na mesma direção da deformação sofrida pelo nó estrutural.
3. Cada um dos vértices é rotacionado ao redor do seu respectivo nó estrutural pelos mesmos ângulos de rotação pelo qual o nó estrutural foi rotacionado pela deformação.

Novamente a metodologia adotada é equivalente a conectar cada um dos vértices ao nó estrutural mais próximo por um elemento rígido.

Para garantir que a metodologia aqui adotada gera resultados coerentes é importante garantir que a malha estrutural seja refinado o suficiente para que sempre exista um nó estrutural próximo ao vértice do painel aerodinâmico. Caso isso não aconteça a abordagem adotada pode gerar deformações excessivas.

3.4.5 Passo 5: Iteração

Utilizando a geometria deformada calculada no passo anterior as cargas aerodinâmicas são recalculadas, repetindo os passos de 1 a 4 até que o critério de convergência adotado seja alcançado ou o número máximo de iterações seja atingido.

O critério de convergência é definido a partir da escolha de um nó de controle. Quando a deformação desse nó entre uma iteração e outra é menor do que um valor pré-estabelecido o programa considera que a convergência foi alcançada e para de iterar.

As cargas aerodinâmicas são sempre aplicadas à estrutura não deformada, uma vez que a formulação adotada não permite a aplicação de uma pré tensão aos elementos de viga, ou seja, aplicar as novas cargas a estrutura deformada seria equivalente a aplicar as cargas a uma estrutura cuja forma original é idêntica à forma deformada calculada.

3.5 Subpacote *loads*

O subpacote *loads* recebe como entrada as cargas em cada um dos painéis aerodinâmicos, calculadas pelo subpacote *aeroelasticity*, e calcula a somatória de todas as forças e momentos do centro de gravidade da aeronave, os coeficientes aerodinâmicos C_l , C_d e C_m , bem como a distribuição de cargas ao longo da envergadura das *macrosurfaces* definidas.

3.6 Subpacote *visualization*

O subpacote *visualization* fornece uma série de funções que permitem gerar gráficos 2D e 3D para a visualização dos resultados. É possível gerar imagens com a aeronave descrita, as malhas aerodinâmicas e estruturais gerada (tanto a original como a deformada), a variação de pressão em cada um dos painéis aerodinâmicos e outros resultados relevantes.

Ele é baseado no pacote *Matplotlib* (MATPLOTLIB-DEVELOPMENT-TEAM, 2019), amplamente utilizado no meio científico.

4 Resultados

4.1 Validação

Com o objetivo de validar a ferramenta *Flying Circus* foi selecionado um caso na literatura com os quais os resultados obtidos foram comparados.

4.1.1 Asa Smith

O caso aqui referido como “Asa Smith” é descrito por Smith, Patil e Hodges (2001) e se trata da análise de uma asa retangular de alto alongamento usando um modelo de viga não-linear para a estrutura e um código de CFD (Euler) para o cálculo aerodinâmico.

Smith, Patil e Hodges utilizaram o código CFD ENS3DAE, optando por uma simulação baseada nas Equações de Euler (escoamento não viscoso), *loosely coupled* com um modelo de viga não-linear desenvolvido por Hodges (1990).

Para o cálculo aeroelástico foi utilizado uma abordagem iterativa, similar àquela usada nesse trabalho, e a simulação foi considerada convergida quando a diferença das deflexões da estrutura entre uma iteração e outra se tornavam menores que 2%. A asa foi engastada pela raiz.

As cargas aerodinâmicas foram transferidas calculando as forças e momentos no centro da corda e interpolando os valores encontrados ao longo da envergadura de forma a adequar a informação ao formato exigido pelo modelo estrutural.

As propriedades da asa são resumidas na tabela 4.1. Smith, Patil e Hodges não fornecem a rígidez no sentido longitudinal da asa (EA), para esse trabalho ela foi estimada em 2000 kN.

Smith, Patil e Hodges fornecem resultados para duas situações de voo: a primeira a 20000 metros de altura, 25 m/s de velocidade verdadeira e ângulo de ataque igual a 2 graus, a segunda na mesma altitude e velocidade mas com ângulo de ataque igual a 4 graus. Foram utilizados 3 métodos diferentes, um método de painéis acoplado com um modelo de viga não-linear, usando o software ENS3DAE (CFD Euler) acoplado a um

TABELA 4.1 – Propriedades da Asa Smith

Propriedade da Asa	Valor
Aerofólio	NACA 0012
Semi Envergadura	16 <i>m</i>
Corda	1 <i>m</i>
Eixo Elástico	0.5 <i>c</i>
Rigidez a Flexão - EI_{yy}	2e4 <i>N.m</i> ²
Rigidez a Flexão - EI_{zz}	1e4 <i>N.m</i> ²
Rigidez a Torção - GJ	1e4 <i>N.m</i> ²
Rigidez a Tração - EA	2e6 <i>N</i>

modelo de viga linear, e utilizando o mesmo software acomplado a um modelo de viga não-linear.

4.1.1.1 Análise e Resultados

Primeiramente foi realizado um estudo de sensibilidade da malha de painéis aerodinâmicos e elementos de vigas utilizados.

Mantendo-se a razão *envergadura/corda* de cada painél igual a 2 (na indústria é considerada uma boa prática que essa razão esteja entre 1 e 3) foram realizadas simulações para o caso de alfa igual a 2 graus variando o número de painéis ao longo da corda de 1 até 10. Foram usados dois elementos de viga para cada faixa de painéis, seguindo a recomendação anterior. Os resultados para as deflexões e para o coeficiente de sustentação da asa com a variação do número de painéis está apresentado nas figuras 4.1 e 4.2, respectivamente. O tempo de execução é apresentado na figura 4.3, foi utilizado um computador com 4 núcleos de processamento com frequência máxima de 3.1GHz e 8gb de memória ram. O critério de convergência adotado foi de 1%.

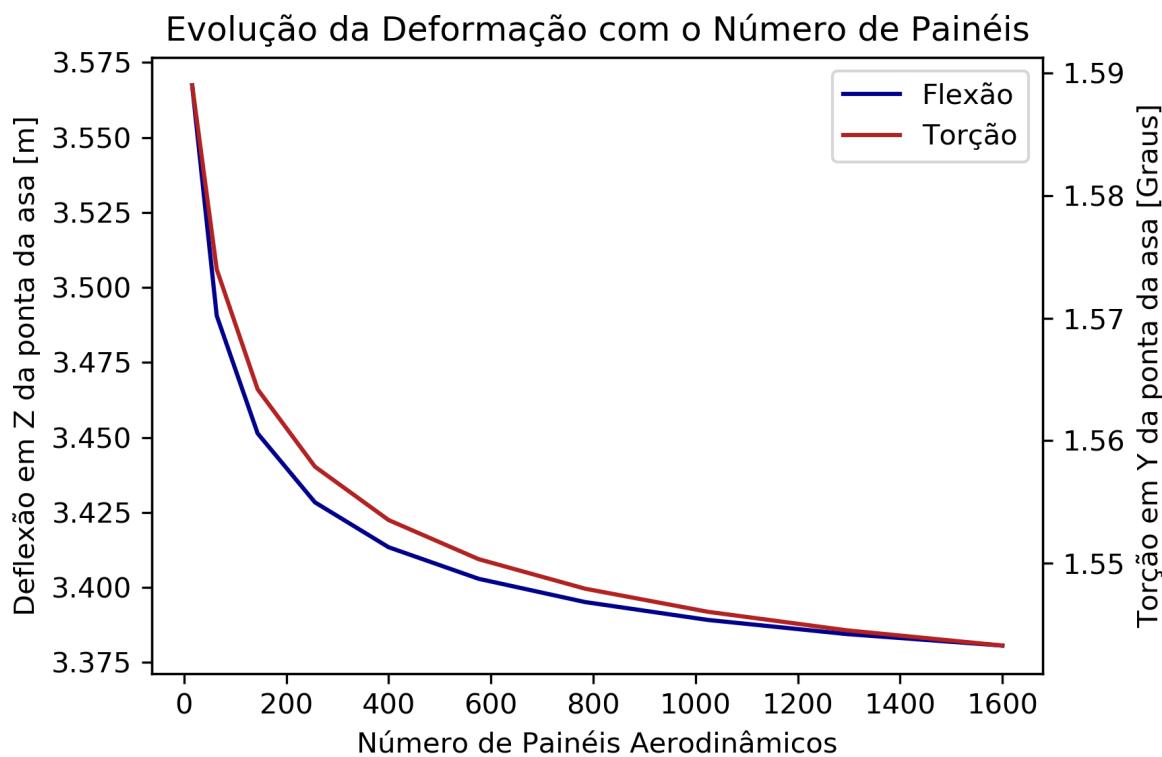


FIGURA 4.1 – Evolução das deflexões calculadas com o número de Painéis

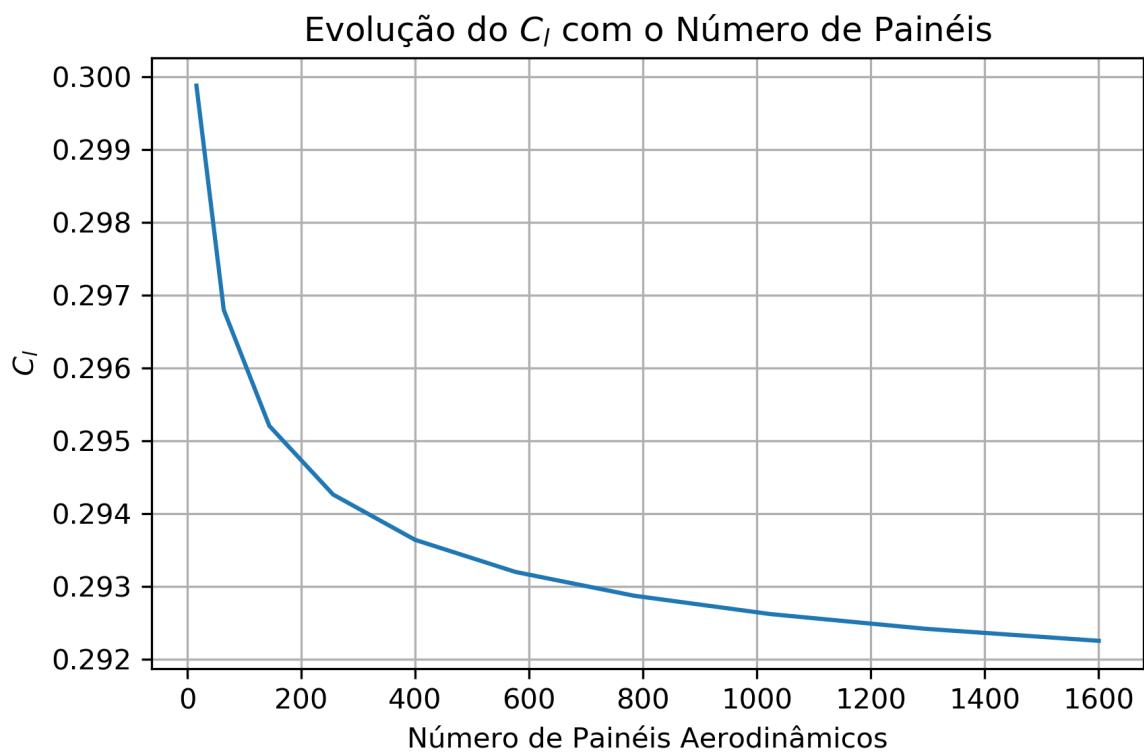


FIGURA 4.2 – Evolução do C_l calculado com o número de painéis

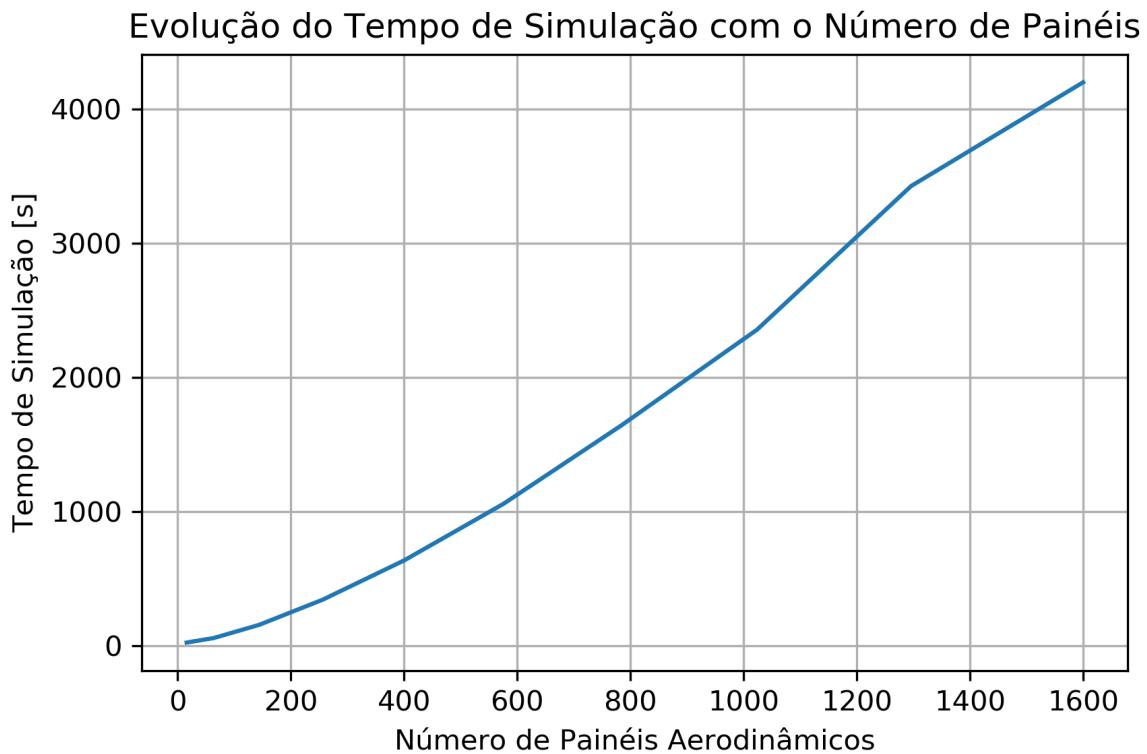


FIGURA 4.3 – Evolução do tempo de simulação com o número de painéis

Nota-se que o tempo de execução aumenta aproximadamente linearmente com o número de painéis, uma vez que o número total de painéis é o produto do número de painéis na corda pelo número de painéis na envergadura refinamentos em qualquer uma das direções leva a um aumento rápido no número total de painéis. Para que a simulação seja o mais rápida possível é preferível que o número de painéis seja mantido o menor possível desde que os resultados sejam precisos o suficiente. Aumentar o número de painéis além de 400 gera ganhos muito pequenos (0.47% de diferença no C_l , 0.96% de diferença na flexão, 0.66% de diferença na torção, em relação ao número máximo de painéis simulados) portanto para o restante desse caso foram utilizados 5 painéis ao longo da corda e 80 painéis ao longo da envergadura, totalizando 400 painéis e 160 elementos de viga.

Utilizando a ferramenta *Flying Circus* foram simulados os dois casos apresentados por Smith, Patil e Hodges tanto para a asa rígida como para a asa flexível. Os resultados obtidos foram comparados com aqueles apresentados na referência.

Apresentados na figura 4.4 estão as malhas aerodinâmicas e estruturais originais e as malhas deformadas.

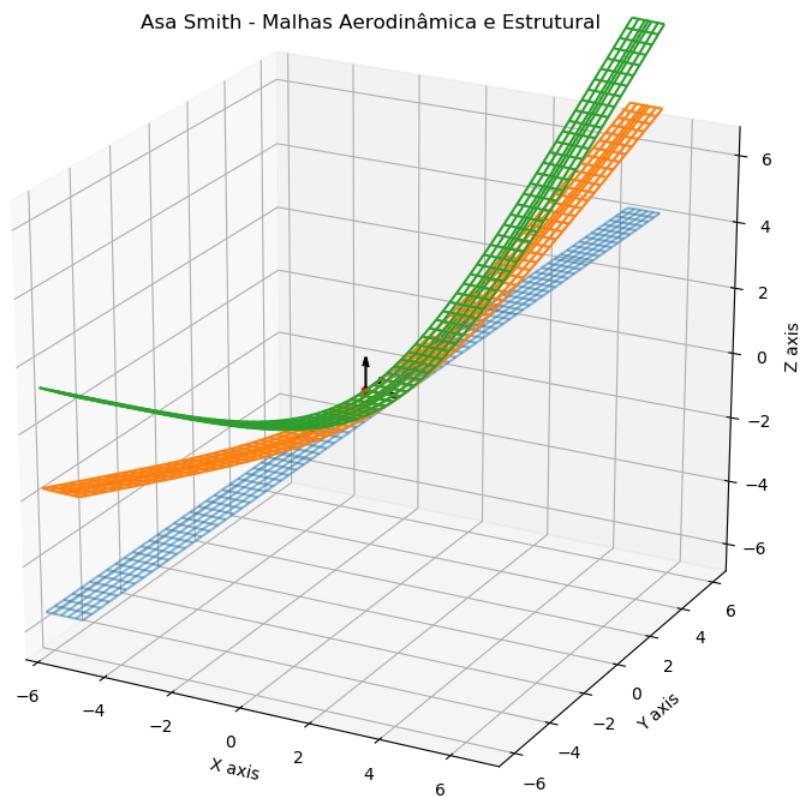
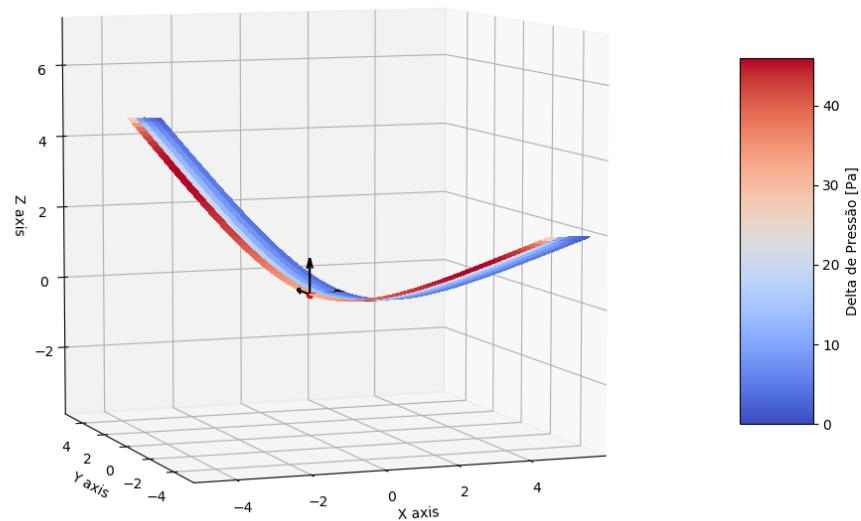


FIGURA 4.4 – Malhas original e deformadas, em azul a malha original, em laranja a deformada para $\alpha = 2$ e em verde a deformada para $\alpha = 4$

Usando a força e a área de cada painel foi possível calcular a diferença de pressão entre o intradorso e o extradorso, essa informação foi sobreposta a malha deformada e mostrada nas figuras 4.5 e 4.6 para os casos de 2 e 4 graus de ângulo de ataque respectivamente.

Asa Smith - Delta de pressão para $\alpha = 2^\circ$ FIGURA 4.5 – Delta de pressão entre intradorso e extradorso para $\alpha = 2$ graus

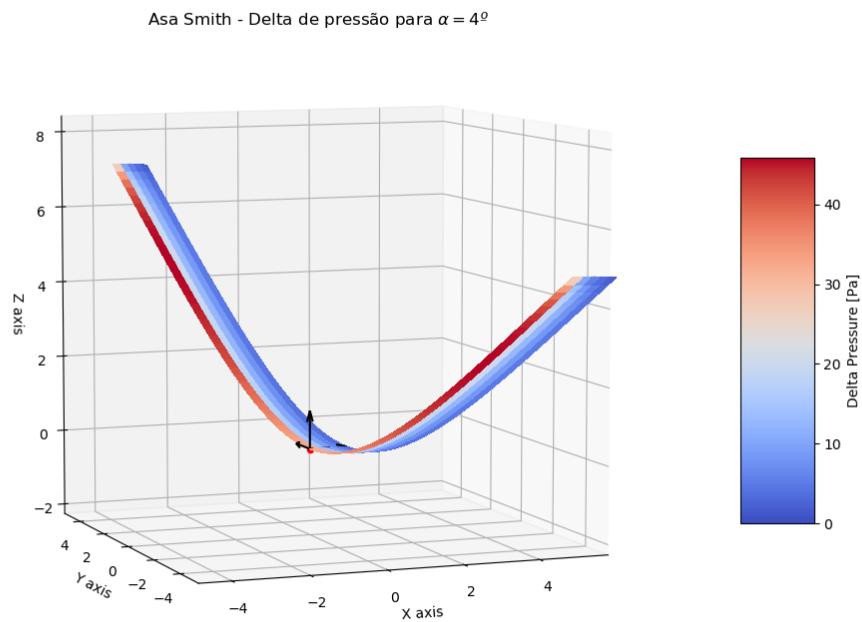


FIGURA 4.6 – Delta de pressão entre intradorso e extradorso para $\alpha = 4$ graus

A figura 4.7 e a figura 4.8 mostram como a deformação sofrida pela asa evolui a medida que as iterações acontecem, um total de 7 iterações foram necessárias para o caso de $\alpha = 2$ e 6 iterações o caso de $\alpha = 4$.

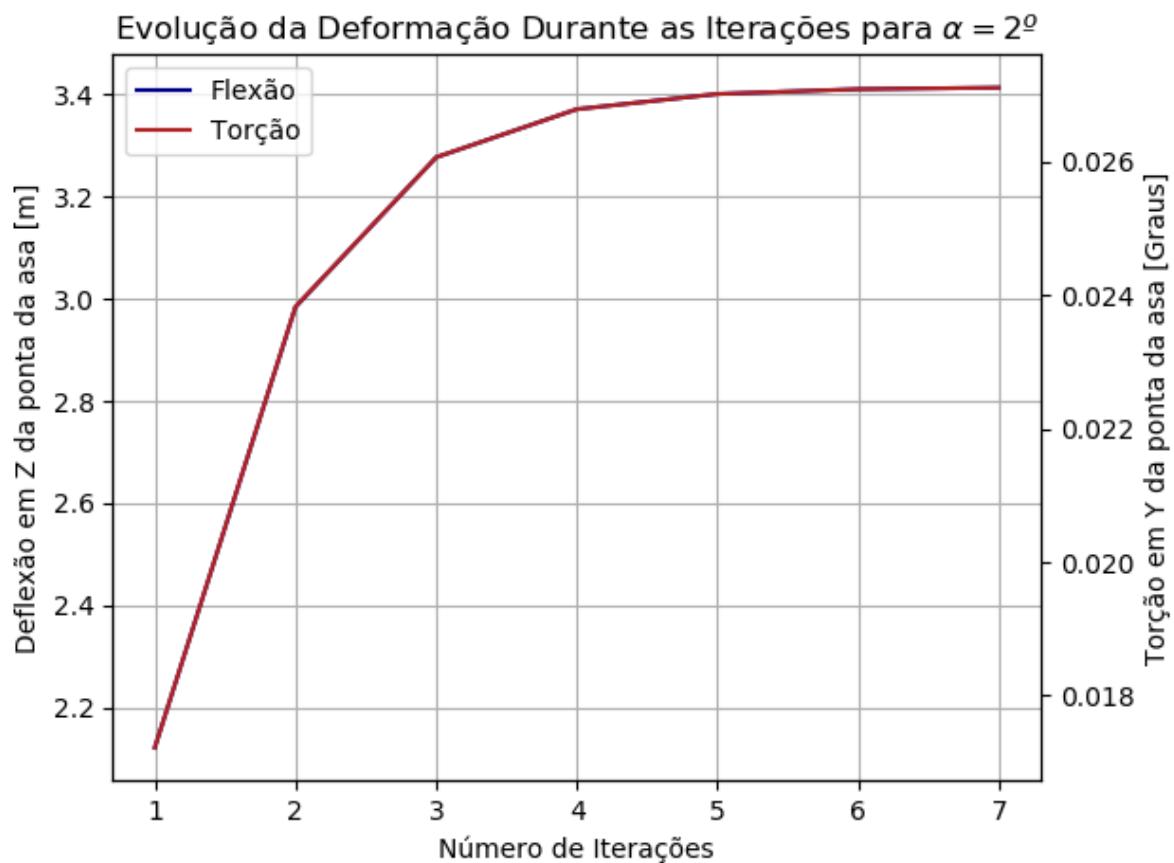


FIGURA 4.7 – Evolução na deformação da ponta da asa durante as iterações para $\alpha = 2$ graus



FIGURA 4.8 – Evolução na deformação da ponta da asa durante as iterações para $\alpha = 4$ graus

A figura 4.9 apresenta o valor de C_l calculado nas simulações, e a figura 4.10 expõe a distribuição de sustentação ao longo da envergadura da asa para os dois ângulos de ataque.

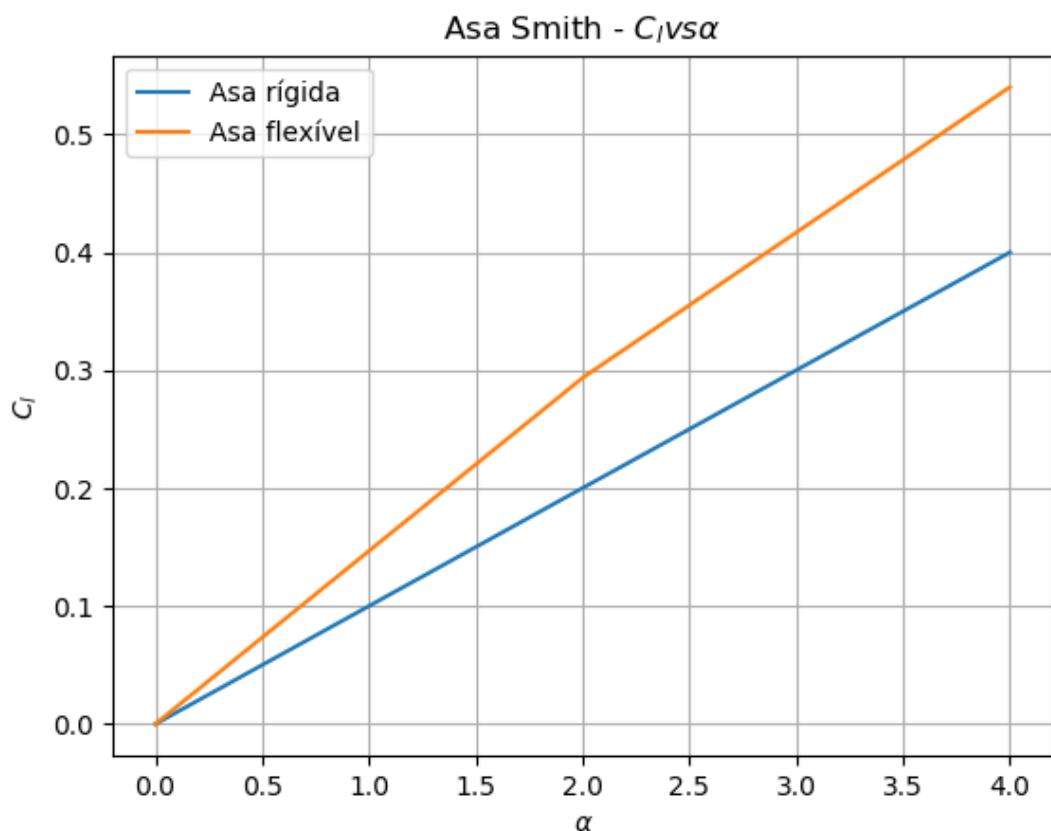


FIGURA 4.9 – C_l vs. α para a Asa Smith, altitude: 20000m, TAS: 25 m/s

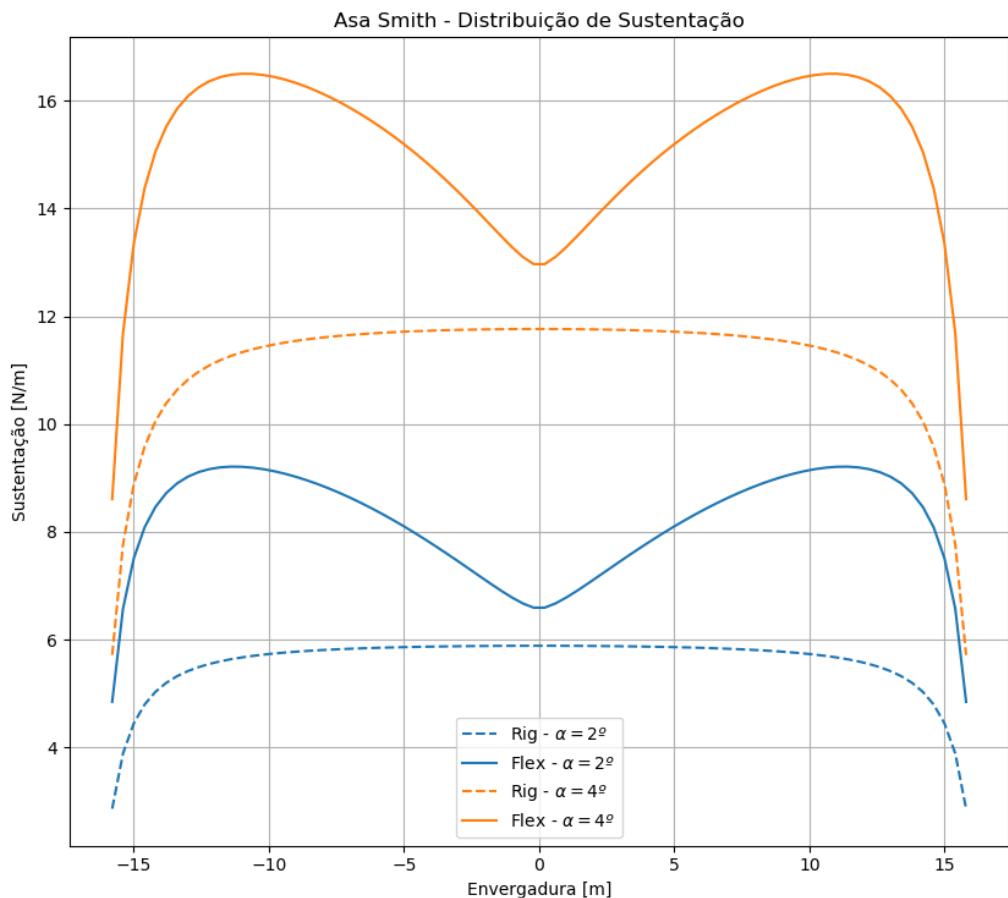
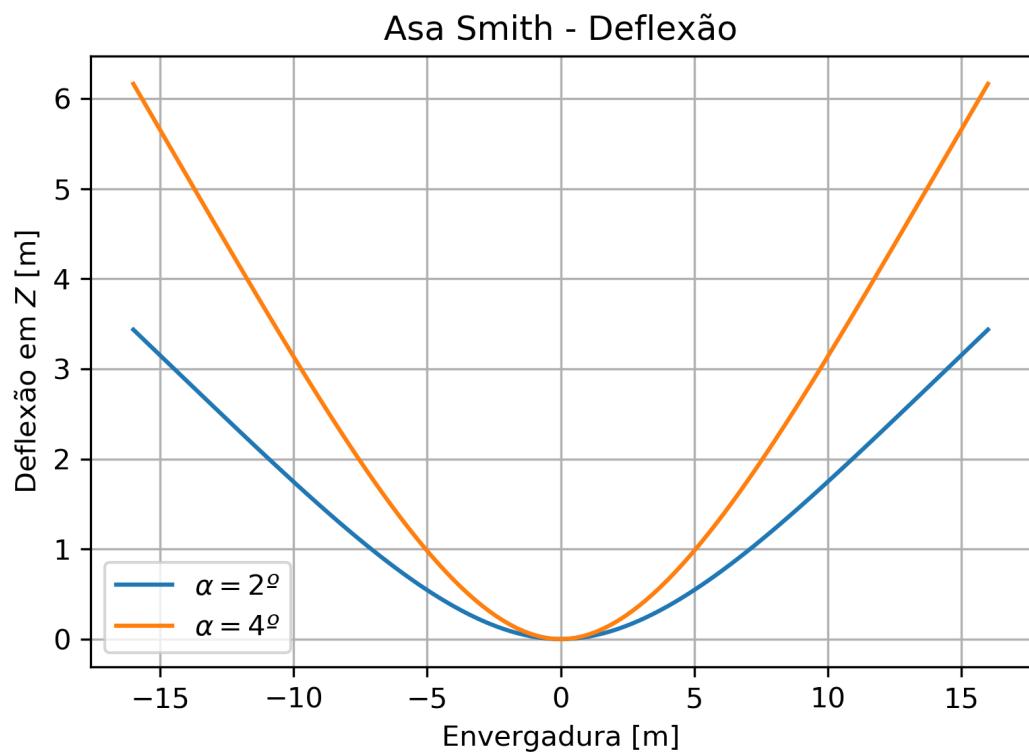
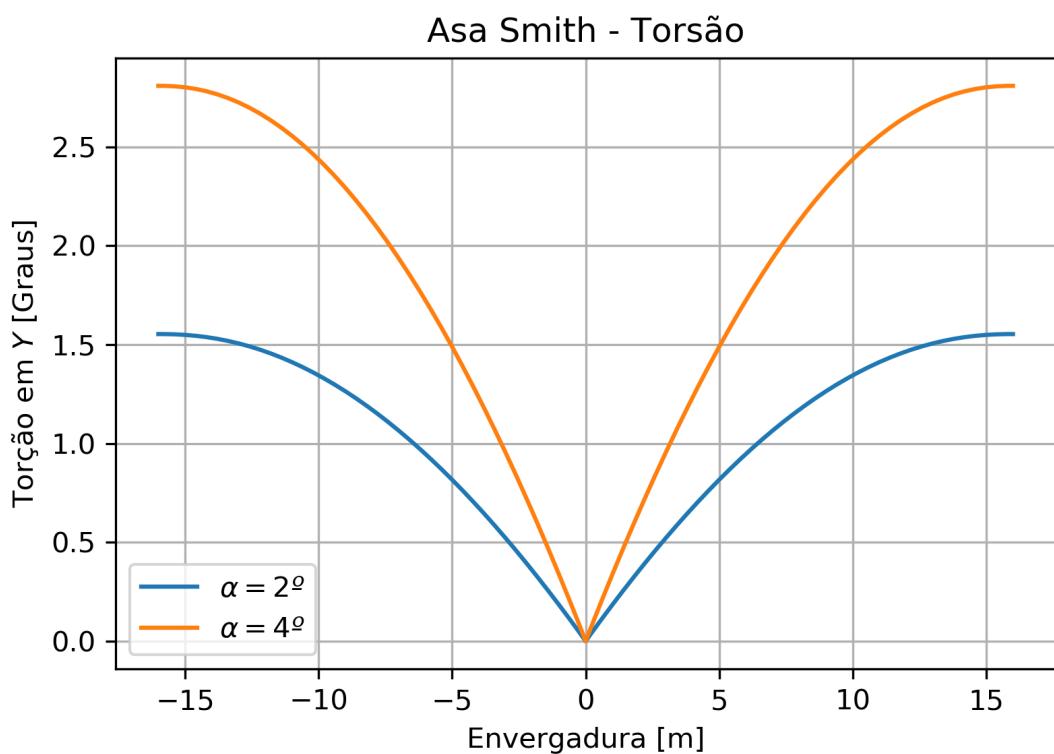


FIGURA 4.10 – Distribuição de sustentação ao longo da envergadura, para a asa rígida e flexível

A figura 4.11 apresenta o valor da deflexão sofrida pela asa, e a figura 4.12 expõe a torção calculada para a asa ao longo do eixo Y.

FIGURA 4.11 – Deflexão em Z ao longo da envergaduraFIGURA 4.12 – Torção em Y ao longo da envergadura

É possível verificar que a asa flexível gera mais sustentação do que a asa rígida. Ao sofrer deformação a asa está sujeita a dois fenômenos, um que tenta aumentar a sustentação gerada e outro que tenta diminuí-la. A torção que as seções da asa sofrem aumentam seu ângulo de ataque efetivo em relação ao escoamento, aumentando assim sua sustentação. Ao mesmo tempo a flexão faz com que parte da força gerada seja projetada na direção lateral, o que na prática diminui a sustentação gerada. No caso estudado o efeito da torção foi preponderante e fez com que a sustentação total fosse superior a aquela da asa rígida. Também é interessante notar que a máxima sustentação na asa flexível não acontece no centro da asa, como é o caso da asa rígida mas sim ao longo de sua envergadura. Isso se deve ao ângulo de ataque gerado pela torção da asa em torno do seu eixo elástico.

Os resultados aqui encontrados são comparados com aqueles apresentados por Smith, Patil e Hedges nas figuras 4.13, 4.14, 4.15 e 4.16.

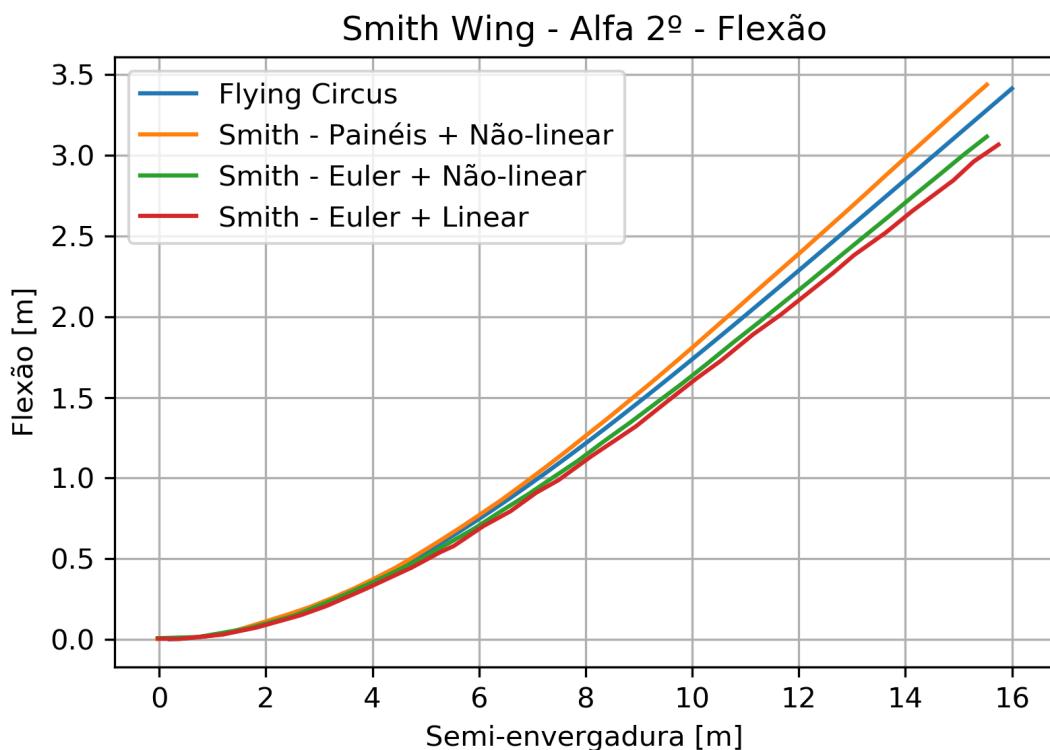


FIGURA 4.13 – Comparaçāo entre os resultados de flexão para $\alpha = 2$ graus

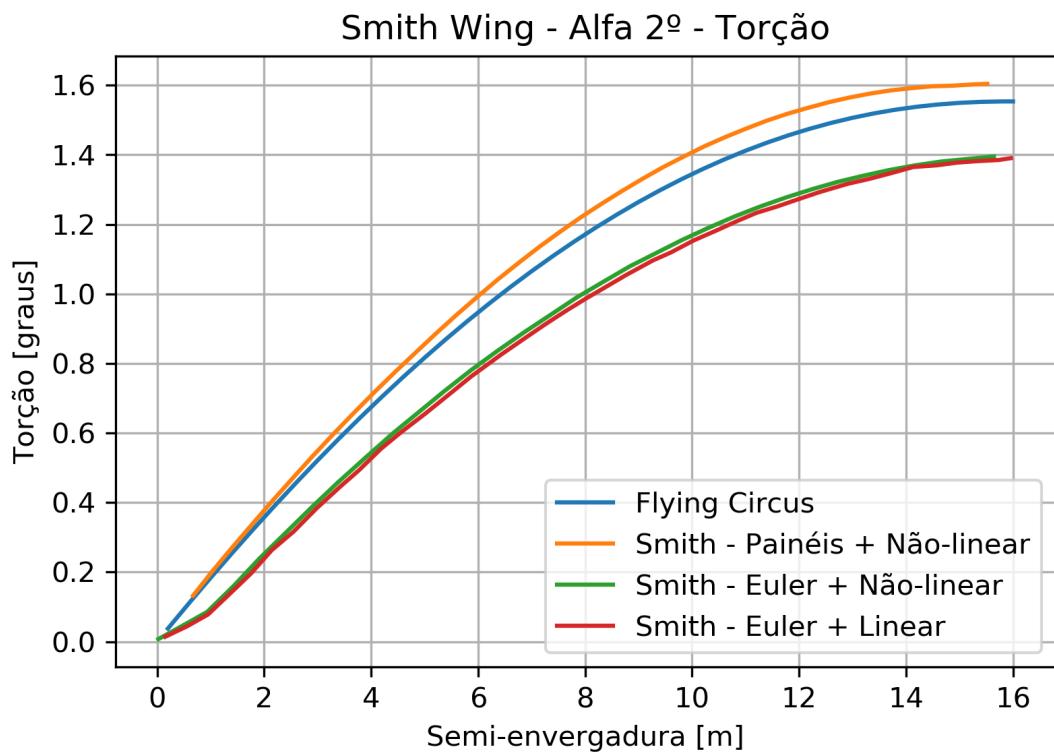


FIGURA 4.14 – Comparação entre os resultados de torção para $\alpha = 2$ graus

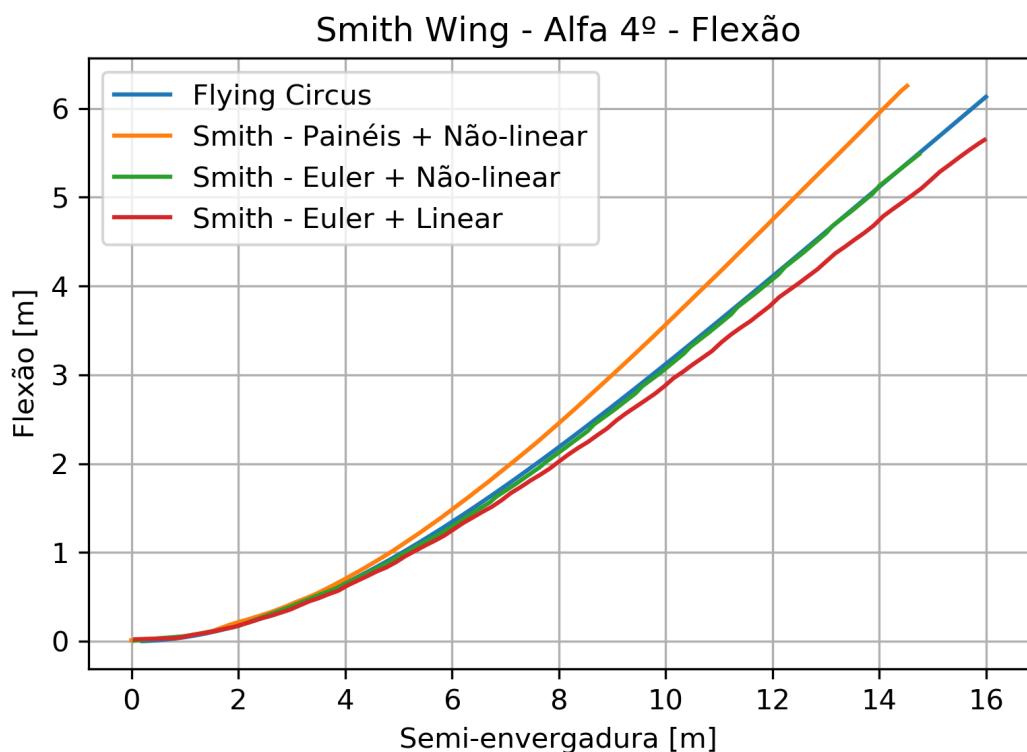


FIGURA 4.15 – Comparação entre os resultados de flexão para $\alpha = 4$ graus



FIGURA 4.16 – Comparação entre os resultados de torção para $\alpha = 4$ graus

Um dos primeiros fatos a serem notados é a diferença, para os dois casos, entre o comprimento da asa calculado usando um viga linear e uma viga não-linear. A viga não-linear se comporta de forma próxima a real, ou seja, a medida que a ponta da asa sobe ela também se move em direção à raiz da asa, ou seja a viga sofre uma rotação mas mantém o seu comprimento. O mesmo não acontece com a metodologia linear onde a ponta da asa se desloca em Z mas mantém a mesma posição em Y , isso já era esperado e faz parte da natureza da formulação linear, entretanto esse fenômeno acaba por extender a asa. Isso é notado para os dois casos, em especial o de ângulo de ataque igual a 4 graus onde, devido a maior intensidade das forças, as limitações da metodologia linear se mostram mais claramente.

Para ambos os ângulos de ataque, a deformação calculada (tanto na flexão como na torção) pela ferramenta *Flying Circus* se encontra entre os resultados obtidos pelo método dos painéis e pelos métodos de CFD. As metodologias de CFD conseguem capturar não-linearidades aerodinâmicas que escapam aos métodos lineares, por conta disso eles tendem a ser menos conservativos o que pode ser observado nos resultados obtidos.

A diferença observada entre os resultados da ferramenta *Flying Circus* e a metodologia de painéis pode ser explicada por diferenças no cálculo aerodinâmico, na forma como as forças aerodinâmicas são transferidas para a estrutura, na maneira como as deformações são transferidas para os painéis aerodinâmicos e pela natureza não-linear do modelo de

viga adotado.

4.2 Estudo de Caso - Aeronave Hale

Patil, Hodges e Cesnik (2001) realizaram várias análises de uma aeronave HALE (*High Altitude Long Endurance*), o mesmo modelo é aqui utilizado para demonstrar exemplos de uso da ferramente *Flying Circus*.

4.2.1 Descrição do Modelo e da Simulação

O modelo utilizado é composto por uma asa retangular, uma empunagem horizontal retangular, uma fuselagem e um *tail boom*. Os autores não fornecem dados sobre a empunagem vertical da aeronave, aqui ela foi modelada como sendo idêntica a uma semi-asa da empunagem horizontal.

A figura 4.17 fornecida por Patil, Hodges e Cesnik resume o modelo da aeronave. Pequenas alterações foram realizadas para facilitar a simulação da aeronave, as próximas seções descrevem em detalhes o modelo utilizado.

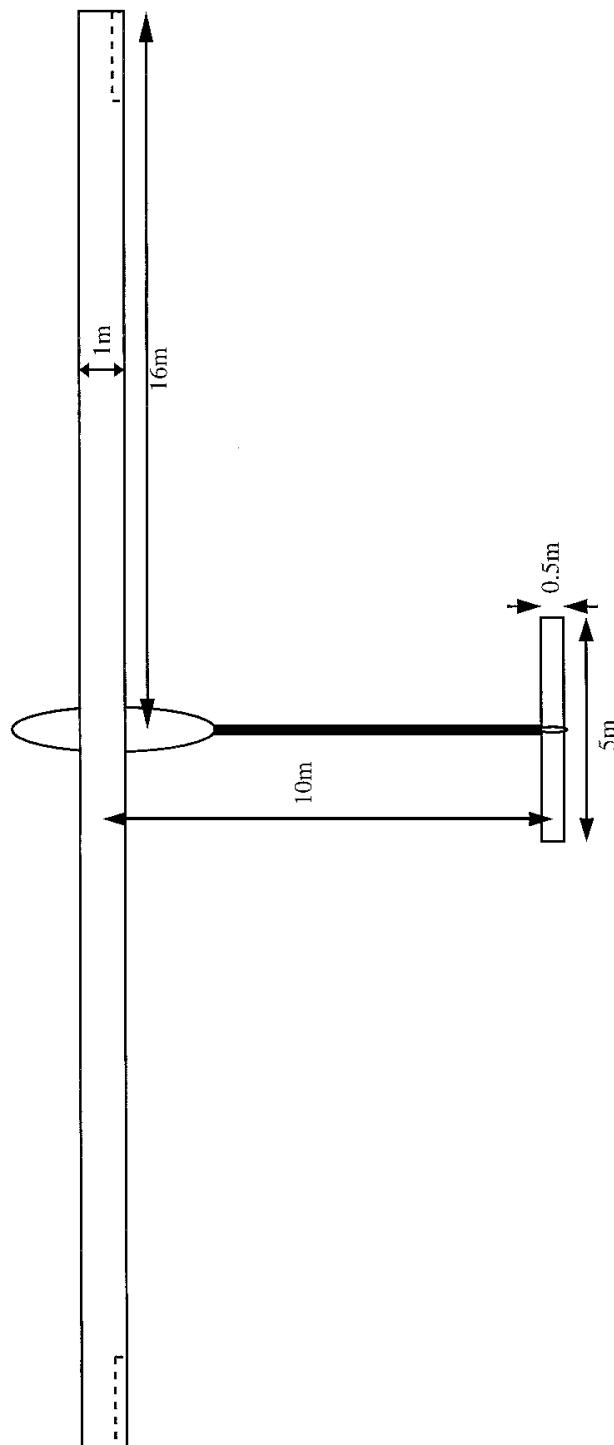


FIGURA 4.17 – Ilustração fornecida por Patil, Hodges e Cesnik da aeronave estudada

4.2.2 Modelo

A aeronave foi modelada por três superfícies aerodinâmicas: a asa, a empenagem horizontal e a empenagem vertical; e por duas vigas uma aqui se referida como fuselagem que conecta a raiz da asa com o centro de gravidade da aeronave e outro chamada de *tail boom* que conecta o C.G. da aeronave à sua empenagem. A posição do centro de

gravidade da aeronave foi estimada como sendo no bordo de fuga da asa.

Patil, Hodges e Cesnik não fornecem informações sobre a rigidez da fuselagem, do *tail boom* ou da empunagem. No modelo adotado esses componentes possuem a mesma rigidez da asa.

A tabela 4.2 resume as propriedades da aeronave e a figuras 4.18, 4.19, apresentam 2 vistas da aeronave.

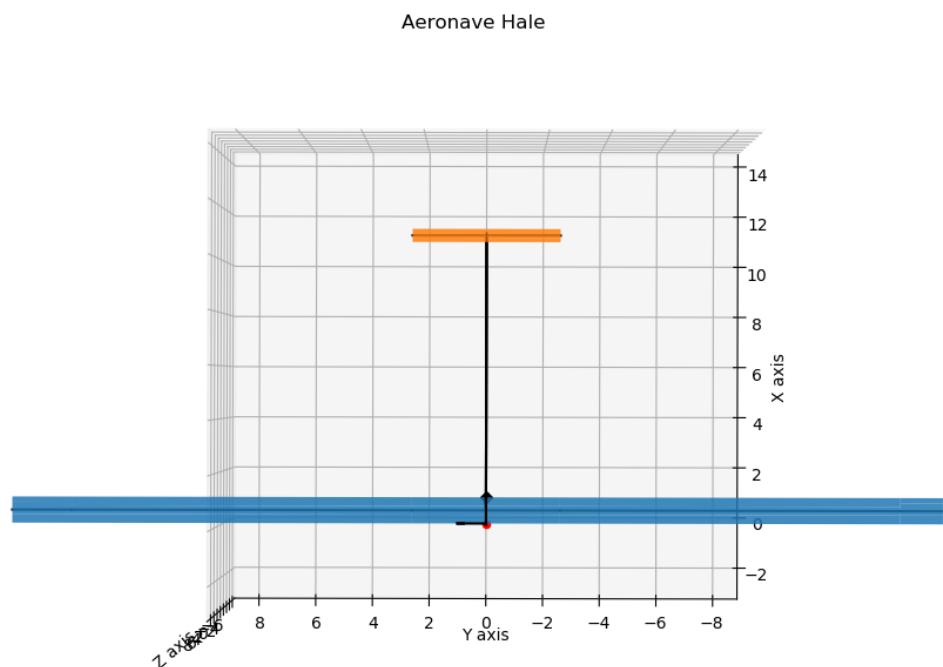


FIGURA 4.18 – Vista superior do modelo utilizado

TABELA 4.2 – Propriedades da aeronave HALE

Propriedade	Valor
Asa - Envergadura	$16m$
Asa - Corda	$1.0m$
Asa - Eixo Elástico	$0.5c$
Asa - Posição do Bordo de Ataque	$0.0m$
Asa - Rigidez a Flexão - EI_{yy}	$2e4N.m^2$
Asa - Rigidez a Flexão - EI_{zz}	$1e4N.m^2$
Asa - Rigidez a Torção - GJ	$1e4N.m^2$
Asa - Rigidez a Tração - EA	$2e6N$
Emp. Horizontal - Envergadura	$5.0m$
Emp. Horizontal - Corda	$0.5m$
Emp. Horizontal - Eixo Elástico	$0.5c$
Emp. Horizontal - Posição do Bordo de Ataque	$10.75m$
Emp. Horizontal - Rigidez a Flexão - EI_{yy}	$2e4N.m^2$
Emp. Horizontal - Rigidez a Flexão - EI_{zz}	$1e4N.m^2$
Emp. Horizontal - Rigidez a Torção - GJ	$1e4N.m^2$
Emp. Horizontal - Rigidez a Tração - EA	$2e6N$
Emp. Vertical - Envergadura	$2.5m$
Emp. Vertical - Corda	$0.5m$
Emp. Vertical - Eixo Elástico	$0.5c$
Emp. Vertical - Posição do Bordo de Ataque	$10.75m$
Emp. Vertical - Rigidez a Flexão - EI_{yy}	$2e4N.m^2$
Emp. Vertical - Rigidez a Flexão - EI_{zz}	$1e4N.m^2$
Emp. Vertical - Rigidez a Torção - GJ	$1e4N.m^2$
Emp. Vertical - Rigidez a Tração - EA	$2e6N$
Fuselagem - Posição inicial	$0.5m$
Fuselagem - Posição final	$1.0m$
Fuselagem - Rigidez a Flexão - EI_{yy}	$2e4N.m^2$
Fuselagem - Rigidez a Flexão - EI_{zz}	$1e4N.m^2$
Fuselagem - Rigidez a Torção - GJ	$1e4N.m^2$
Fuselagem - Rigidez a Tração - EA	$2e6N$
Tail boom - Posição inicial	$1.0m$
Tail boom - Posição final	$11.0m$
Tail boom - Rigidez a Flexão - EI_{yy}	$2e4N.m^2$
Tail boom - Rigidez a Flexão - EI_{zz}	$1e4N.m^2$
Tail boom - Rigidez a Torção - GJ	$1e4N.m^2$
Tail boom - Rigidez a Tração - EA	$2e6N$
Posição do Centro de Gravidade em X	$1.0m$
Posição do Centro de Gravidade em Y	$0.0m$
Posição do Centro de Gravidade em Z	$0.0m$

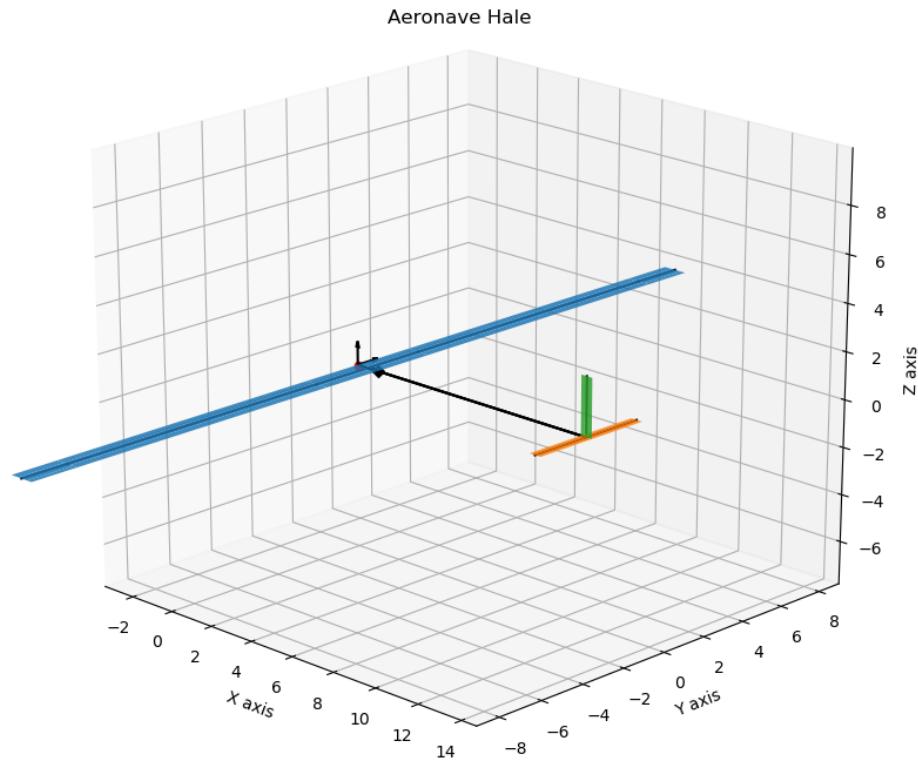


FIGURA 4.19 – Vista em 3/4 do modelo utilizado

A asa da aeronave HALE é muito similar a Asa Smith portanto os resultados anteriores foram utilizados para orientar a criação da malha, a tabela 4.3 resume as propriedades da malha utilizada.

4.2.3 Simulação

A aeronave foi engastada no seu centro de gravidade e foram realizadas simulações considerando uma altitude de 20000 m, uma velocidade verdadeira (*TAS*) de 25m/s e ângulos de ataque variando de 0 a 5 graus.

Foram calculados a somatória das forças no C.G. da aeronave e calculados o C_l e o C_m , bem como as deflexões nas superfícies. Foi utilizado um critério de convergência de 1% e, após testes em alguns pontos, a ponta da empunhadura horizontal foi escolhida como nó de controle.

TABELA 4.3 – Propriedades da Malha Aerodinâmica e Estrutural da Aeronave Hale

Propriedade	Valor
Asa - Painéis na Envergadura	64
Asa - Painéis na Corda	5
Asa - Elementos de Viga	128
Emp. Horizontal - Painéis na Envergadura	10
Emp. Horizontal - Painéis na Corda	5
Emp. Horizontal - Elementos de Viga	20
Emp. Vertical - Painéis na Envergadura	5
Emp. Vertical - Painéis na Corda	5
Emp. Vertical - Elementos de Viga	10
Fuselagem - Elementos de Viga	2
<i>Tail Boom</i> - Elementos de Viga	40

4.2.4 Resultados

As figuras 4.20, 4.21 e 4.22 mostram a deformação da estrutura para as 5 condições de voo estudadas. Notasse que o comportamento da asa da aeronave é muito semelhante ao observado na Asa Smith, com uma diferença importante: Uma vez que o engaste se encontra na fuselagem, a qual é flexível, a raiz da asa está livre para se movimentar. Essa característica faz com que o ângulo de ataque da raiz da asa seja diferente daquele observado na aeronave rígida, esse ângulo adicional altera as características de voo da aeronave e se não for considerado pode gerar problemas graves, a aeronave flexível pode entrar em estol antes do que a aeronave rígida, por exemplo.

A deformação para $\alpha = 5$ chega a quase 50% da semi-envergadura. Essa deformação é considerável e a metodologia linear adotada para o cálculo estrutural começa a se mostrar inadequada, especialmente quando se considera o efeito de extensão da asa discutido anteriormente.

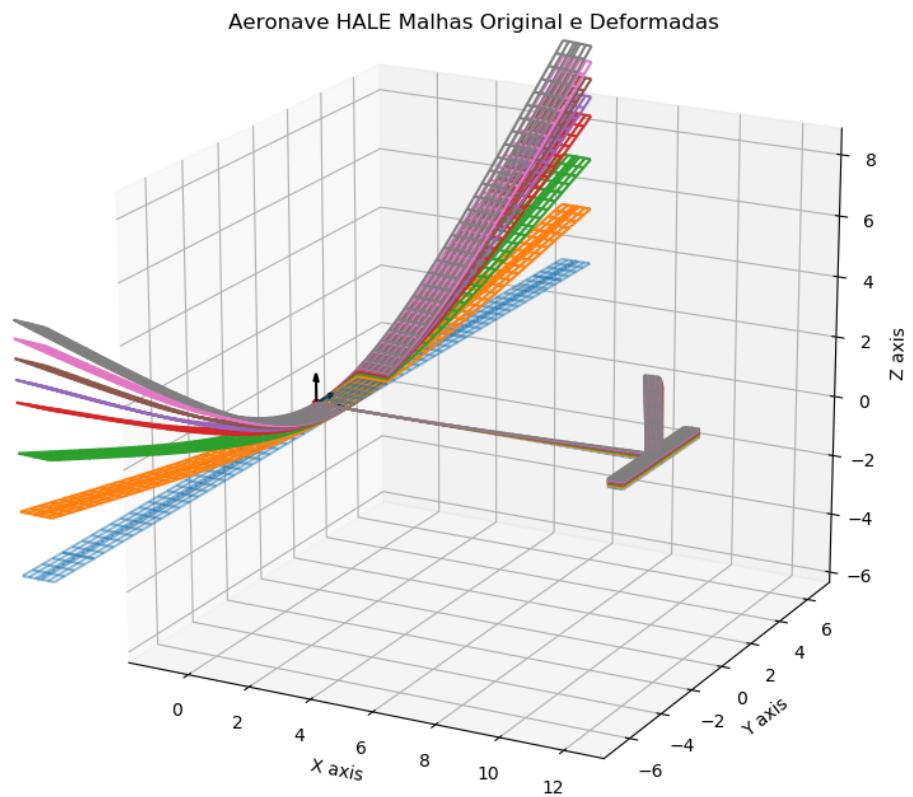


FIGURA 4.20 – Vista em 3/4 das malhas aerodinâmica e estrutural deformadas

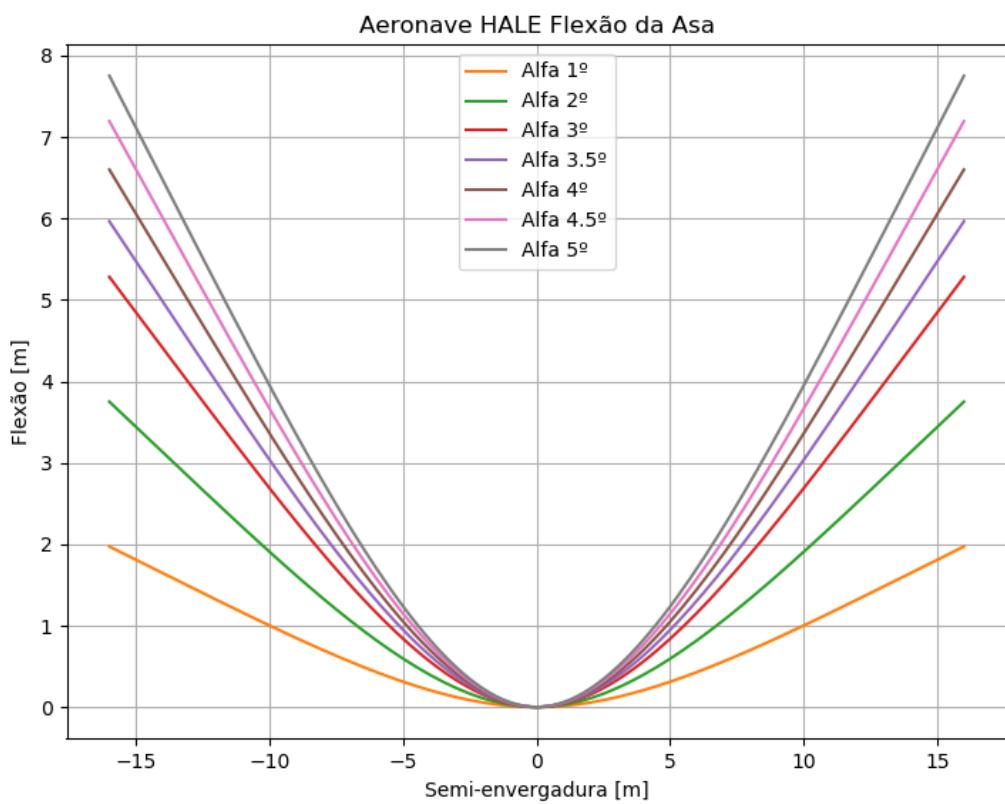


FIGURA 4.21 – Flexão da asa

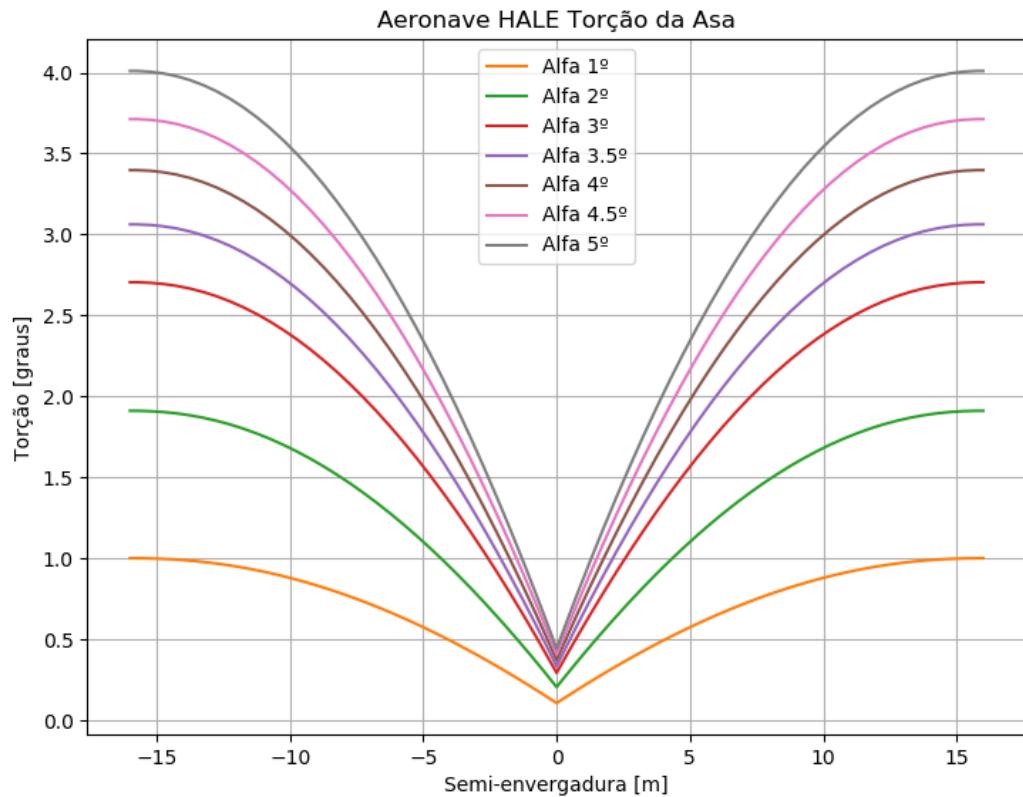


FIGURA 4.22 – Torção da asa

As figuras 4.23 mostra o delta de pressão na superfície da asa flexível, novamente aqui é o comportamento observado é similar ao da Asa Smith.

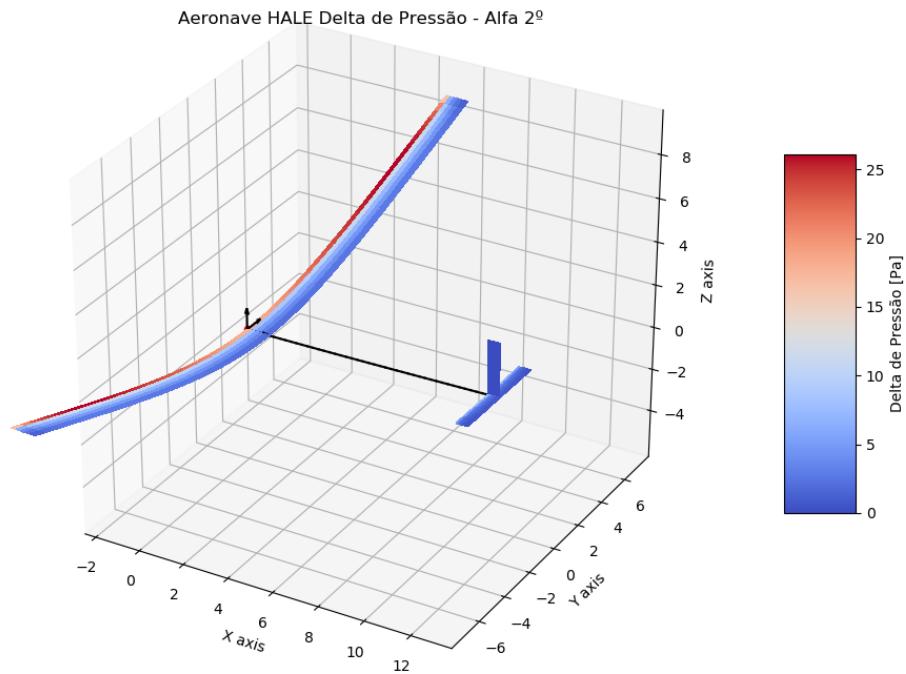


FIGURA 4.23 – Vista em 3/4 da deformação e delta de pressão para $\alpha = 2$ graus

As figuras 4.24, 4.26 e 4.25 apresentam as curvas de C_l , C_m e C_d versus α . Como era esperado a aeronave flexível gera mais sustentação do que a aeronave rígida, consequentemente ela também gera mais arrasto. Aqui notasse outro fator importante decorrente da flexibilidade da aeronave, a derivada de C_m muda de sinal a partir de $\alpha = 4$ graus, esse fenômeno pode ser explicado observando-se as figuras 4.27 e 4.28. Nota-se que para a asa a aeronave flexível gera mais sustentação do que a asa rígida, devido principalmente a torção por ela sofrida, entretanto na empenagem horizontal acontece o oposto a empenagem flexível gera menos sustentação que a empenagem rígida. Esse fenômeno acontece por conta da deformação sofrida pela fuselagem que acaba por rotacionar a empenagem no sentido negativo de α , diminuindo assim o seu ângulo de ataque efetivo.

Aqui novamente observa-se uma diferença importante entre a aeronave rígida e a flexível. A trimagem da aeronave flexível é consideravelmente afetada pela deformação por ela sofrida e suas características de estabilidade não só são diferentes daquelas da aeronave rígida como também podem se alterar de acordo com as condições de voo.

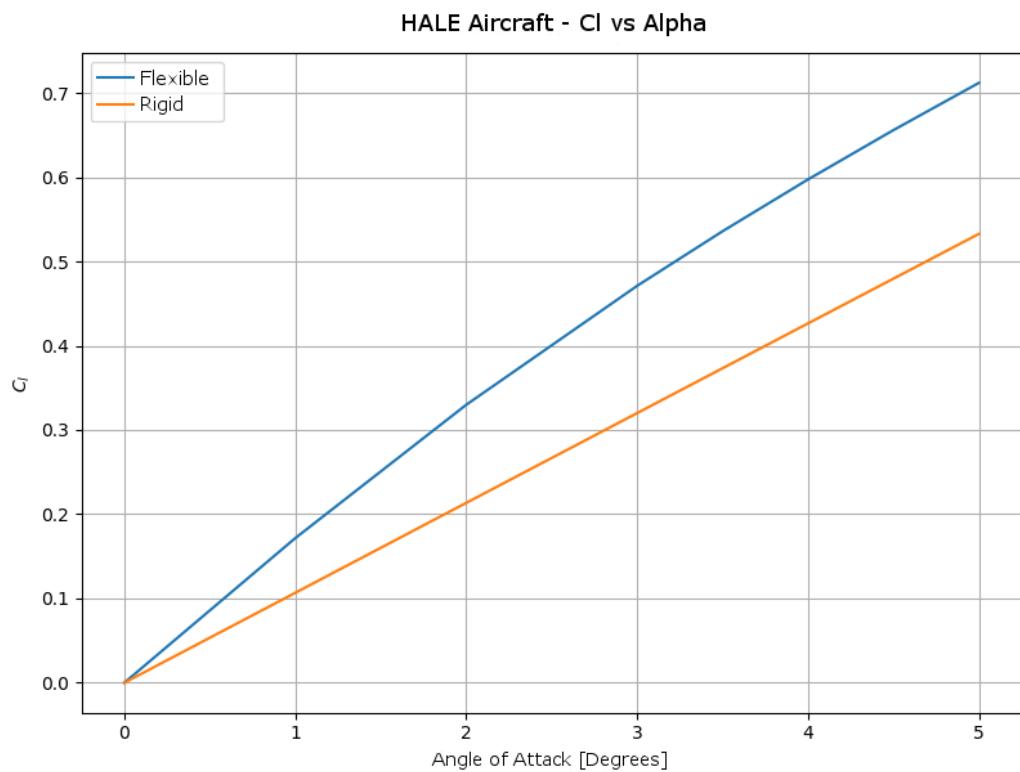


FIGURA 4.24 – Curva de C_l da aeronave para os ângulos de ataque simulados

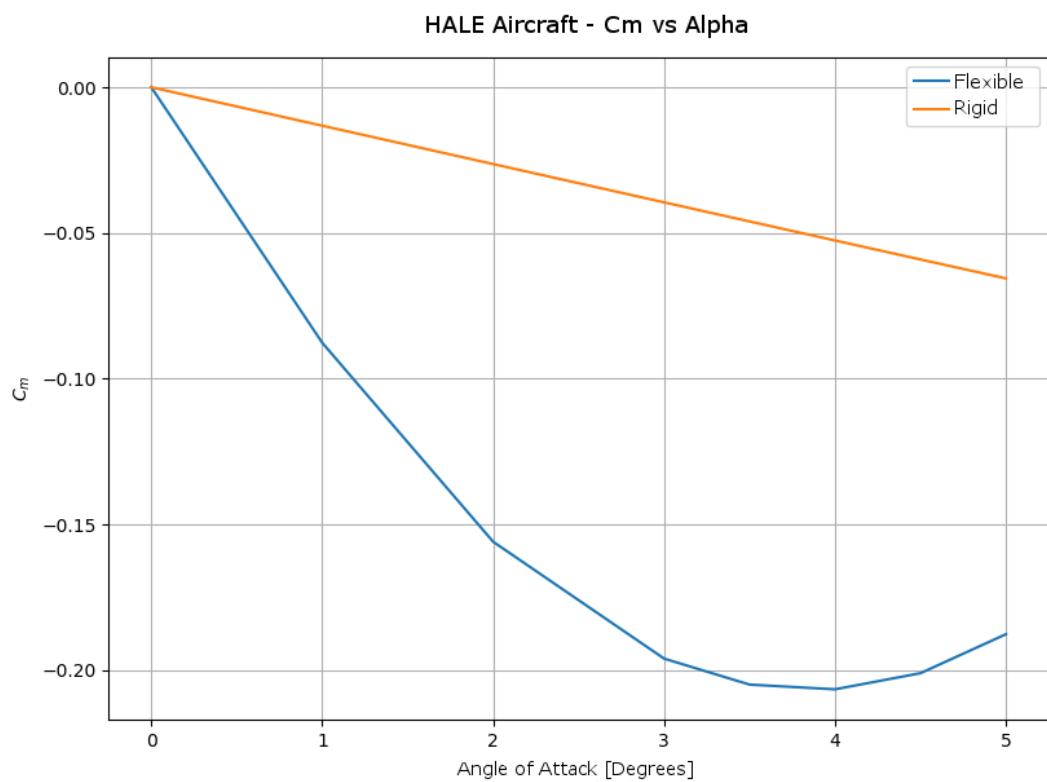


FIGURA 4.25 – Curva de C_m da aeronave para os ângulos de ataque simulados

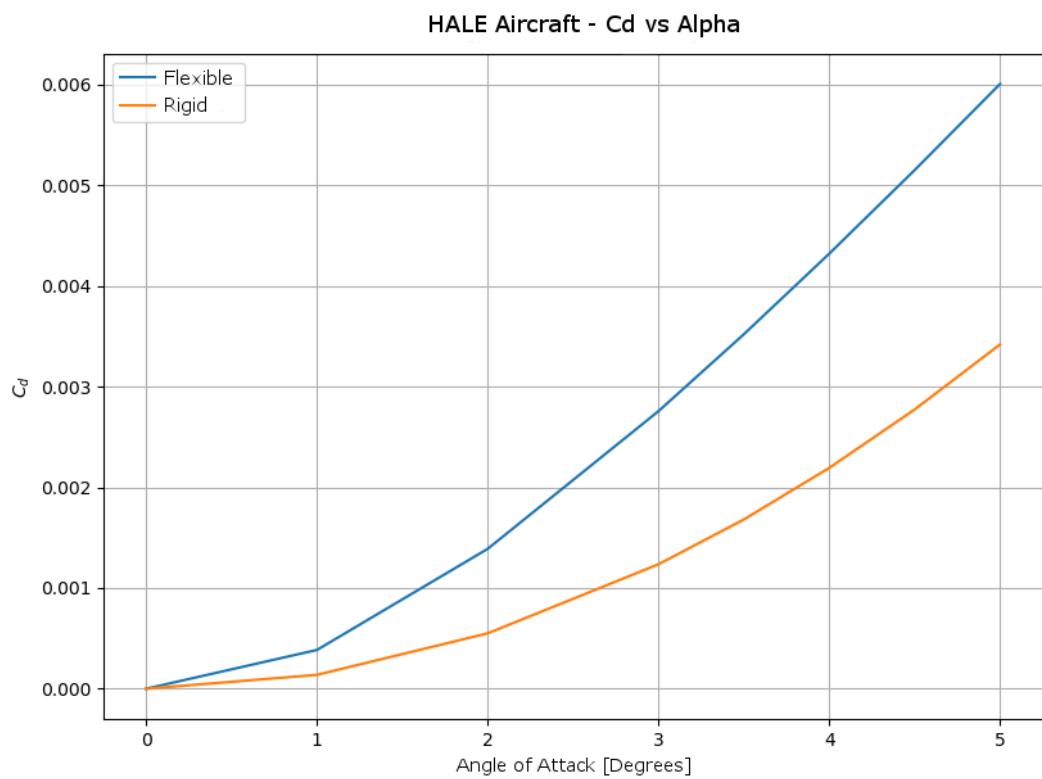


FIGURA 4.26 – Curva de C_d da aeronave para os ângulos de ataque simulados

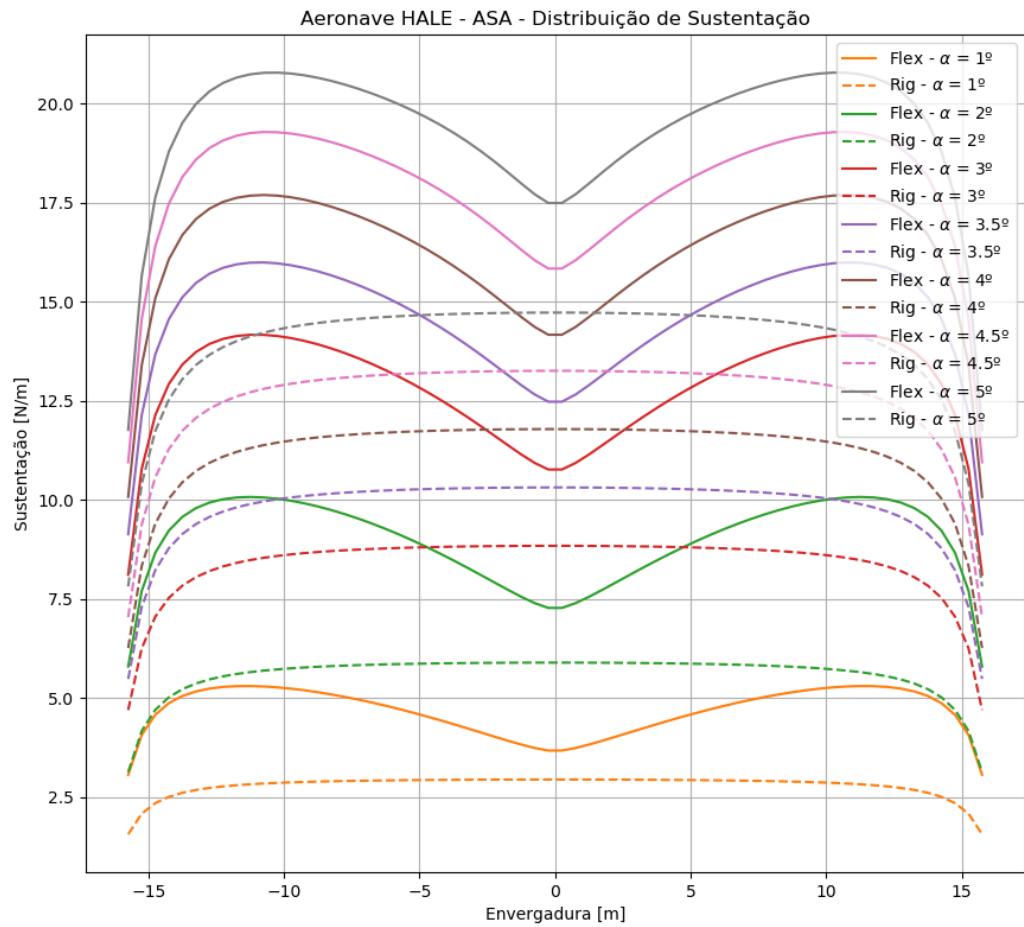


FIGURA 4.27 – Distribuição de sustentação ao longo da asa

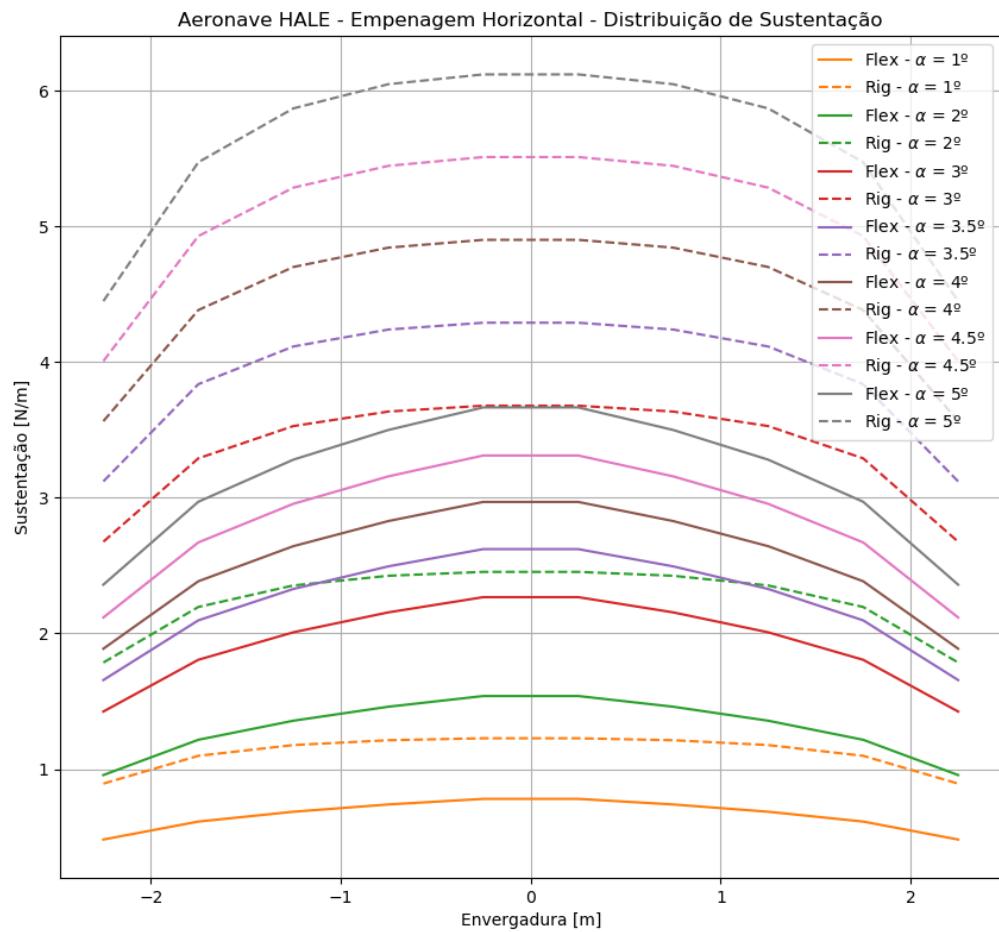


FIGURA 4.28 – Distribuição de sustentação ao longo da empenagem horizontal

5 Conclusão

5.1 Comentários Finais

Foi desenvolvida uma ferramenta, camada *Flying Circus* usando a linguagem de programação *Python* e bibliotecas de código aberto para a análise de aeroelasticidade estática para projeto conceitual de aeronaves. Os resultados obtidos por essa ferramenta foram validados contra outros resultados obtidos na literatura e posteriormente a ferramenta foi utilizada para avaliar um aeronave do tipo HALE.

A ferramenta usa um método de *vortex lattice* para os cálculos aerodinâmicos, o método de elementos finitos de viga de Euler-Bernoulli para o cálculo estrutural, um critério de proximidade para o acoplamento aerodinâmico/estrutural e um método iterativo para o cálculo aeroelástico.

Os resultados obtidos estão coerentes com aqueles observados na literatura e possuem um grau de confiabilidade compatível com a etapa de projeto conceitual.

O estudo de caso da aeronave HALE mostrou como a ferramenta é capaz de prever fenômenos importante para o projeto de aeronaves flexíveis. Entretanto o mesmo estudo demonstrou as limitações das características lineares da abordagem adotada.

Considera-se que esse trabalho cumpriu seus principais objetivos e que a ferramenta, desde que utilizada dentro de suas limitações, é uma contribuição útil para o desenvolvimento de metodologias para o projeto e análise de aeronaves flexíveis.

5.2 Sugestões para Trabalhos Futuros

A natureza *open source* da ferramenta *Flying Circus* e seu projeto modular tornam o seu aprimoramento e expansão relativamente simples, uma vez que seus componentes podem ser aprimorados de forma independente uns dos outros e que não existem custos financeiros ligados à obtenção ou execução do código desenvolvido.

Seguem a seguir sugestões de aprimoramento da ferramenta e para trabalhos futuros.

1. Implementação de uma metodologia não-linear para o cálculo estrutural de forma a tornar a ferramenta mais adequada ao estudo de aeronaves muito flexíveis.
2. Implementação de uma abordagem mais robusta para o acoplamento aerodinâmico/estrutural permitindo a análise de geometrias mais complexas e diminuindo a necessidade de ajuste das malhas.
3. Otimização das rotinas de cálculo, em especial a de deformação da malha aerodinâmica.
4. Implementação de correções de compressibilidade e camada limite para o cálculo aerodinâmico aumentando assim o seu grau de confiabilidade.
5. Acoplamento da ferramenta com um simulador de voo de forma a permitir a inclusão de efeitos de aeroelasticidade estática na dinâmica de voo e controle de aeronaves flexíveis.
6. Implementação de metodologias de cálculos não-estacionário para a aerodinâmica e para a avaliação da dinâmica estrutural.

Referências

- ABBAS, A.; VICENTE, J. de; VALERO, E. Aerodynamic technologies to improve aircraft performance. **Aerospace Science and Technology**, Elsevier BV, v. 28, n. 1, p. 100–132, jul 2013.
- AFONSO, F.; VALE, J.; OLIVEIRA, É.; LAU, F.; SULEMAN, A. A review on non-linear aeroelasticity of high aspect-ratio wings. **Progress in Aerospace Sciences**, Elsevier BV, v. 89, p. 40–57, feb 2017.
- ANACONDA. **Numba: A High Performance Python Compiler**. 2019. Disponível em: <<http://numba.pydata.org/>>.
- ANDERSON, J. D. **Fundamentals of Aerodynamics**. New York: McGraw-Hill Education, 2010.
- CARMICHAEL, R. **Properties Of The U.S. Standard Atmosphere 1976**. 2018. Disponível em: <<http://www.pdas.com/atmos.html>>.
- CEASION. **CEASIOM - Conceptual Aircraft Design tool**. 2019. Disponível em: <<https://www.ceasiom.com/wp/>>.
- CLEANSKYJU. **Welcome to Clean Sky | Clean Sky**. 2018. Disponível em: <<https://www.cleansky.eu/>>.
- CONCEPTUAL-RESEARCH-CORPORATION. **RDS-win Aircraft Design Software**. 2019. Disponível em: <<http://www.aircraftdesign.com/rds.shtml>>.
- COSTA, J. P. M. C. da. **Flying Circus: A Python Library for Aeronautics**. 2019. Disponível em: <<https://joaopaulomcc.com/flyingcircus/>>.
- DARCORPORATION. **Advanced Aircraft Analysis|DARcorporation|Aeronautical Software**. 2019. Disponível em: <<https://www.darcorp.com/advanced-aircraft-analysis-software/>>.
- DEYOUNG, J. Historical evolution of vortex-lattice methods. In: **Vortex-Lattice Utilization**. [S.l.]: NASA - National Aeronautics and Space Administrations, 1976.
- DRELA, M. Integrated simulation model for preliminary aerodynamic, structural, and control-law design of aircraft. In: **40th Structures, Structural Dynamics, and Materials Conference and Exhibit**. [S.l.]: American Institute of Aeronautics and Astronautics, 1999.
- DRELA, M. **ASWING**. 2010. Disponível em: <<http://web.mit.edu/drela/Public/web/aswing/>>.
- DRELA, M. **Flight Vehicle Aerodynamics**. 1. ed. Cambridge: MIT Press, 2014.

- FECHNER, U. **Fast linear algebra for 3D vectors using numba 0.17 or newer.** 2019. Disponível em: <<http://kieranwynn.github.io/pyquaternion/>>.
- GROZDANOV, A. **Transonic static aeroelasticity using the 2.5d nonlinear vortex lattice method.** Dissertação (Mestrado) — Université de Montréal, Montreal, 12 2017.
- HODGES, D. H. A mixed variational formulation based on exact intrinsic equations for dynamics of moving beams. **International Journal of Solids and Structures**, v. 26, n. 11, p. 1253 – 1273, 1990. ISSN 0020-7683.
- HODGES, D. H.; PIERCE, G. A. **Introduction to Structural Dynamics and Aeroelasticity.** 2. ed. Cambridge: Cambridge University Press, 2009.
- HOUGHTON, E. L.; CARPENTER, P. W.; COLLICOTT, S. H.; VALENTINE, D. **Aerodynamics for Engineering Students.** 6. ed. Oxford: Butterworth-Heinemann, 2012.
- IATA. **Economic Performance of the Airline Industry.** 2018.
- IATA. **IATA - Climate Change.** 2019. Disponível em: <<https://www.iata.org/policy/environment/Pages/climate-change.aspx>>.
- KATZ, J.; PLOTKIN, A. **Low-Speed Aerodynamics.** 2. ed. Cambridge: Cambridge University Press, 2001.
- LOGAN, D. L. **A First Course in the Finite Element Method.** 5. ed. Cambridge: CL Engineering, 2011.
- MATPLOTLIB-DEVELOPMENT-TEAM. **Matplotlib: Python plotting.** 2019. Disponível em: <<https://matplotlib.org/>>.
- NASA. **Begginer's Guide to Aerodynamics.** 2015. Disponível em: <<https://www.grc.nasa.gov/www/k-12/airplane/bga.html>>.
- NUMPY-DEVELOPERS. **NumPy – Numpy.** 2019. Disponível em: <<https://www.numpy.org/>>.
- PATIL, M. J.; HODGES, D. H.; CESNIK, C. E. S. Nonlinear aeroelasticity and flight dynamics of high-altitude long-endurance aircraft. **Journal of Aircraft**, American Institute of Aeronautics and Astronautics (AIAA), v. 38, n. 1, p. 88–94, jan 2001.
- PYTHON-SOFTWARE-FOUNDATION. **Welcome to Python.org.** 2019. Disponível em: <<https://www.python.org/>>.
- RAYMER, D. P. **Aircraft Design: A Conceptual Approach.** 2. ed. Washington: American Institute of Aeronautics, 1992.
- RENTEMA, D. **AIDA: Artificial Intelligence supported conceptual Design of Aircraft.** Delft: [s.n.], 2004.
- RIBEIRO, F. L. C. **Dinâmica de Voo de Aeronaves Muito Flexíveis.** Dissertação (Mestrado) — Instituto Tecnológico de Aeronáutica, São José dos Campos, 11 2011.
- ROSKAM, J. **Airplane design; parts I to VIII.** 1. ed. Ottawa: DARcorporation, 1987–1990.
- RUGGERI, M. C. **Development of Methodologies of Aeroelastic Analysis for the Design of Flexible Aircraft Wings.** Dissertação (Mestrado) — Instituto Tecnológico de Aeronáutica, São José dos Campos, 12 2015.

- SCIPY-DEVELOPERS. **SciPy – SciPy**. 2019. Disponível em: <<https://www.scipy.org/>>.
- SMITH, M.; PATIL, M.; HODGES, D. CFD-based analysis of nonlinear aeroelastic behavior of high-aspect ratio wings. In: **19th AIAA Applied Aerodynamics Conference**. [S.l.]: American Institute of Aeronautics and Astronautics, 2001.
- STANFORD-AEROSPACE-DESIGN-LAB. **SUAVE**. 2019. Disponível em: <<http://suave.stanford.edu/index.html>>.
- TRAVAGLINI, L. **PyPAD: A Framework for Multidisciplinary Aircraft Design**. Milão: [s.n.], 2016.
- WRIGHT, J. R.; COOPER, J. E. **Introduction to Aircraft Aeroelasticity and Loads**. 2. ed. Chichester: John Wiley & Sons Ltd, 2014.
- WYNN, K. **pyquaternion**. 2019. Disponível em: <<http://kieranwynn.github.io/pyquaternion/>>.

Apêndice A - Código Fonte

O código da ferramenta *Flying Circus* foi estruturado como um módulo Python contendo vários submódulos, cada um deles composto vários arquivos, a figura mostra a estrutura de diretórios

```
flying_circus
|
|--- __init__.py
|
|--- mathematics.py
|
|--- aerodynamics
|     |--- __init__.py
|     |--- functions.py
|     |--- objects.py
|     |--- vlm.py
|
|--- aeroelasticity
|     |--- __init__.py
|     |--- functions.py
|
|--- geometry
|     |--- __init__.py
|     |--- functions.py
|     |--- objects.py
|
|--- loads
|     |--- __init__.py
|     |--- functions.py
|     |--- objects.py
|
|--- structures
|     |--- __init__.py
|     |--- fem.py
|     |--- functions.py
|     |--- objects.py
|
|--- visualization
|     |--- __init__.py
|     |--- plot_3D.py
|     |--- plot_2D.py
```

A.1 __init__.py

```
1 from . import aerodynamics
2 from . import aeroelasticity
3 #from . import control
```

```

4 #from . import flight_mechanics
5 from . import geometry
6 from . import structures
7 from . import visualization
8 from . import mathematics

```

A.2 mathematics.py

```

1 -*- coding: utf-8 -*-
2 """
3 * This file is part of FreeKiteSim.
4 *
5 * FreeKiteSim -- A kite-power system power simulation software.
6 * Copyright (C) 2013 by Uwe Fechner, Delft University
7 * of Technology, The Netherlands. All rights reserved.
8 *
9 * FreeKiteSim is free software; you can redistribute it and/or
10 * modify it under the terms of the GNU Lesser General Public
11 * License as published by the Free Software Foundation; either
12 * version 3 of the License, or (at your option) any later version.
13 *
14 * FreeKiteSim is distributed in the hope that it will be useful,
15 * but WITHOUT ANY WARRANTY; without even the implied warranty of
16 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
17 * Lesser General Public License for more details.
18 *
19 * You should have received a copy of the GNU Lesser General Public
20 * License along with SystemOptimizer; if not, write to the Free Software
21 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
22 """
23 """ Fast implementation of the following linear algebra functions for 3-dimensional arrays:
24     sum2, sum3, sub2, sub3, mul2, mul3, div2, div3, neg_sum, copy2
25     cross, cross3, dot, norm, normalize, normalize1, normalize2
26     The number behind the name is the number of parameters for the functions, that do
27     not allocate memory. These functions always store the result in the last parameter.
28     Average speed-up factor compared to numpy between 2 and 50, heavily machine dependant,
29     but also dependent on the call context: If these procedures are called from another
30     procedure, that was also compiled with Numba they can be much faster than when called
31     from standard Python code.
32     This version was tested with Numba 0.18. """
33 # TODO: add test case for dot with 3x3 rotation matrix
34 # pylint: disable=E0611
35 from numba import jit, double
36 import math
37 import numpy as np
38
39 @jit(nopython = True)
40 def sum2(vec, result):
41     """ Calculate the sum of two 3d vectors and store the result in the second parameter. """
42     result[0] = vec[0] + result[0]
43     result[1] = vec[1] + result[1]
44     result[2] = vec[2] + result[2]
45
46 @jit(nopython = True)
47 def sum3(vec1, vec2, result):
48     """ Calculate the sum of two 3d vectors and store the result in the third parameter. """
49     result[0] = vec1[0] + vec2[0]
50     result[1] = vec1[1] + vec2[1]
51     result[2] = vec1[2] + vec2[2]
52
53 @jit(nopython = True)
54 def sub2(vec, result):

```

```

55     """ Calculate the difference of two 3d vectors and store the result in the second parameter.
56     """
57     result[0] = result[0] - vec[0]
58     result[1] = result[1] - vec[1]
59     result[2] = result[2] - vec[2]
60
61 @jit(nopython = True)
62 def sub3(vec1, vec2, result):
63     """ Calculate the difference of two 3d vectors and store the result in the third parameter.
64     """
65     result[0] = vec1[0] - vec2[0]
66     result[1] = vec1[1] - vec2[1]
67     result[2] = vec1[2] - vec2[2]
68
69 @jit(nopython = True)
70 def mul2(a, result):
71     """ Calculate the product of a scalar and a 3d vector and store the result in the second
72     parameter."""
73     result[0] = a * result[0]
74     result[1] = a * result[1]
75     result[2] = a * result[2]
76
77 @jit(nopython = True)
78 def mul3(a, vec, result):
79     """ Calculate the product of a scalar and a 3d vector. """
80     result[0] = a * vec[0]
81     result[1] = a * vec[1]
82     result[2] = a * vec[2]
83
84 @jit(nopython = True)
85 def div2(a, result):
86     """ Divide a 3d vector by a scalar and store the result in the second parameter."""
87     result[0] = result[0] / a
88     result[1] = result[1] / a
89     result[2] = result[2] / a
90
91 @jit(nopython = True)
92 def div3(a, vec, result):
93     """ Divide a 3d vector by a scalar and store the result in the third parameter. """
94     result[0] = vec[0] / a
95     result[1] = vec[1] / a
96     result[2] = vec[2] / a
97
98 @jit(nopython = True)
99 def neg_sum(a, b, c, result):
100    """ Calculate the sum of three vectors and multiply the result with -1. """
101    result[0] = -(a[0] + b[0] + c[0])
102    result[1] = -(a[1] + b[1] + c[1])
103    result[2] = -(a[2] + b[2] + c[2])
104
105 @jit(nopython = True)
106 def copy2(a, result):
107    """ Calculate the difference of two 3d vectors. """
108    result[0] = a[0]
109    result[1] = a[1]
110    result[2] = a[2]
111
112 @jit
113 def cross(vec1, vec2):
114    """ Calculate the cross product of two 3d vectors. """
115    result = np.zeros(3)
116    return cross_(vec1, vec2, result)
117
118    """ Calculate the cross product of two 3d vectors. """

```

```

119     a1, a2, a3 = double(vec1[0]), double(vec1[1]), double(vec1[2])
120     b1, b2, b3 = double(vec2[0]), double(vec2[1]), double(vec2[2])
121     result[0] = a2 * b3 - a3 * b2
122     result[1] = a3 * b1 - a1 * b3
123     result[2] = a1 * b2 - a2 * b1
124     return result
125
126 @jit(nopython=True)
127 def cross3(vec1, vec2, result):
128     """ Calculate the cross product of two 3d vectors. """
129     a1, a2, a3 = double(vec1[0]), double(vec1[1]), double(vec1[2])
130     b1, b2, b3 = double(vec2[0]), double(vec2[1]), double(vec2[2])
131     result[0] = a2 * b3 - a3 * b2
132     result[1] = a3 * b1 - a1 * b3
133     result[2] = a1 * b2 - a2 * b1
134
135
136 @jit(nopython=True)
137 def dot(vec1, vec2):
138     """ Calculate the dot product of two 3d vectors. """
139     return vec1[0] * vec2[0] + vec1[1] * vec2[1] + vec1[2] * vec2[2]
140
141 @jit(nopython=True)
142 def norm(vec):
143     """ Calculate the norm of a 3d vector. """
144     return math.sqrt(vec[0]*vec[0] + vec[1]*vec[1] + vec[2]*vec[2])
145
146 @jit()
147 def normalize(vec):
148     """ Calculate the normalized vector (norm: one). """
149     result = np.copy(vec)
150     return normalize1(result)
151
152 @jit(nopython = True)
153 def normalize1(vec):
154     norm_=norm(vec)
155     if norm_ < 1e-6:
156         vec[0] = 0.
157         vec[1] = 0.
158         vec[2] = 0.
159     else:
160         vec[0] = vec[0] / norm_
161         vec[1] = vec[1] / norm_
162         vec[2] = vec[2] / norm_
163     return vec
164
165 @jit(nopython = True)
166 def normalize2(vec, result):
167     norm_=norm(vec)
168     if norm_ < 1e-6:
169         result[0] = 0.
170         result[1] = 0.
171         result[2] = 0.
172     else:
173         result[0] = vec[0] / norm_
174         result[1] = vec[1] / norm_
175         result[2] = vec[2] / norm_
176
177 def init():
178     """ call all functions once to compile them """
179     vec1, vec2 = np.array((1.0, 2.0, 3.0)), np.array((2.0, 3.0, 4.0))
180     result = np.zeros(3)
181     a = 1.24
182     sum2(vec1, result)
183     sum3(vec1, vec2, result)
184     sub2(vec1, result)
185     sub3(vec1, vec2, result)

```

```

186     mul2(a, result)
187     mul3(a, vec1, result)
188     div2(a, result)
189     div3(a, vec1, result)
190     result = normalize(vec2)
191     normalize1(vec1)
192     normalize2(vec1, result)
193     cross(vec1, vec2)
194     cross3(vec1, vec2, result)
195     dot(vec1, vec2)
196     norm(vec1)
197
198 init()
199
200
201 """
202 Results on i7-3770 CPU @ 3.40GHz
203
204 time for empty loop 0.000405073165894
205
206 time for sum2 with numba [ts]: 0.25
207 time sum with numpy [ts]: 0.53
208 speedup of sum2 with numba: 2.13
209
210 time for numba norm [ts]: 0.19
211 time for linalg norm [ts]: 5.17
212 speedup of norm with numba: 27.25
213
214 time for numba dot [ts]: 0.27
215 time for numpy dot [ts]: 0.68
216 speedup of dot with numba: 2.47
217
218 time for numba cross [ts]: 1.39
219 time for numba cross3 [ts]: 0.30
220 time for numpy cross [ts]: 14.75
221 speedup of cross with numba: 48.37
222
223 time for numba normalize [ts]: 1.83
224 time for numba normalize2 [ts]: 0.27
225 time for numpy normalize [ts]: 7.54
226 speedup of normalize with numba: 27.93
227 """

```

A.3 aerodynamics

A.3.1 __init__.py

```

1 from . import objects
2 from . import functions
3 from . import vlm

```

A.3.2 functions.py

```

1 import numpy as np
2 from numpy import sin, cos, tan, arccos, arcsin, arctan
3
4 from .. import mathematics as m
5 from .. import geometry as geo

```

```

6 from numba import jit
7
8 #
=====

9
10
11 @jit(nopython=True)
12 def horse_shoe_ind_vel(point_a, point_b, target_point, circulation, vortex_radius=0.001):
13     """
14         reference: Flight Vehicle Aerodynamics - Mark Drela
15     """
16     r = target_point
17     ra = point_a
18     rb = point_b
19     a = r - ra
20     b = r - rb
21
22     X = np.array([1.0, 0.0, 0.0])
23
24     # Vortex segment induced velocity
25     if geo.functions.distance_point_to_line(point_a, point_b, target_point) <= vortex_radius:
26         seg_vel = np.zeros(3)
27     else:
28         seg_vel = (m.cross(a, b) / (m.norm(a) * m.norm(b) + m.dot(a, b))) * (
29             1 / m.norm(a) + 1 / m.norm(b)
30         )
31
32     # Wake a induced velocity
33     if (
34         geo.functions.distance_point_to_line(point_a, (point_a + X), target_point)
35         <= vortex_radius
36     ):
37         wake_a_vel = np.zeros(3)
38
39     else:
40         wake_a_vel = (m.cross(a, X) / (m.norm(a) - m.dot(a, X))) * (1 / m.norm(a))
41
42     if (
43         geo.functions.distance_point_to_line(point_b, (point_b + X), target_point)
44         <= vortex_radius
45     ):
46         wake_b_vel = np.zeros(3)
47
48     else:
49         wake_b_vel = (m.cross(b, X) / (m.norm(b) - m.dot(b, X))) * (1 / m.norm(b))
50
51     # Horse shoe induced velocity
52     hs_vel = 0.25 * (circulation / np.pi) * (seg_vel + wake_a_vel - wake_b_vel)
53
54     return hs_vel
55
56 #
=====

57
58 @jit(nopython=True)
59 def horse_shoe_aero_force(point_a, point_b, circulation, flow_vector, air_density):
60     """
61         reference: Flight Vehicle Aerodynamics - Mark Drela
62     """
63     ra = point_a
64     rb = point_b
65     l = rb - ra
66
67     aero_force = air_density * m.cross(flow_vector, l) * circulation
68

```

```

69     return aero_force
70
71 #
72 """
73 atmosphere.py
74
75 Collection of function for calculation of atmospheric properties based on the
76 1976 Standard Atmosphere.
77 Based, with very slight modifications, on the Tables.py program found at
78 http://www.pdas.com/atmos.html.
79
80 Below is the original author information
81 -----
82 Tables.py - Make tables of atmospheric properties      (Python 3)
83
84 Adapted by
85     Richard J. Kwan, Lightsaber Computing
86 from original programs by
87     Ralph L. Carmichael, Public Domain Aeronautical Software
88
89 Revision History
90 Date          Vers Person Statement of Changes
91 2004 Oct 04   1.0   RJK    Initial program
92 2017 Jun 04   1.1   RLC    All indents are with spaces; all prints in ( )
93                           New version for Python 3 that does integer div with //
94 """
95
96 import sys
97 import math
98
99 version = "1.1 (2017 Jun 04)"
100 greeting = "Tables - A Python program to compute atmosphere tables"
101 author = "Ralph L. Carmichael, Public Domain Aeronautical Software"
102 modifier = ""
103 farewell = "Four files added to your directory."
104 finalmess = "Normal termination of tables."
105
106 # PHYSICAL CONSTANTS
107
108 FT2METERS = 0.3048      # mult. ft. to get meters (exact)
109 KELVIN2RANKINE = 1.8      # mult deg K to get deg R
110 PSF2NSM = 47.880258      # mult lb/sq.ft to get sq.m
111 SCF2KCM = 515.379        # mult slugs/cu.ft to get kg/cu.m
112 TZERO = 288.15           # sea-level temperature, kelvins
113 PZERO = 101325.0          # sea-level pressure, N/sq.m
114 RHOZERO = 1.225          # sea-level density, kg/cu.m
115 AZERO = 340.294          # speed of sound at S.L. m/sec
116 BETAVISC = 1.458E-6       # viscosity constant
117 SUTH = 110.4              # Sutherland's constant, kelvins
118
119
120 def LongUSTable():
121     Itxt = open('us1py.prt', 'w')
122     Itxt.write(' alt    sigma      delta      theta ')
123     Itxt.write(' temp    press      dens      a      visc  k.visc\n')
124     Itxt.write('          Kft          ')
125     Itxt.write(' degR lb/sq.ft  s/cu.ft      fps s/ft-s sq.ft/s\n')
126
127     for i in range(-1, 57):
128         altKm=5*i*FT2METERS
129         (sigma, delta, theta) = Atmosphere(altKm)
130         Itxt.write("%4i %9.3E %9.3E %6.4f % "
131                     (5*i, sigma, delta, theta))
132         temp=(KELVIN2RANKINE*TZERO)*theta
133         pressure=(PZERO/PSF2NSM)*delta

```

```

134     density=(RHOZERO/SCF2KCM)*sigma
135     asound=(AZERO/FT2METERS)*math.sqrt(theta)
136     Itxt.write("%5.1f %8.3E %9.3E %6.1f" %
137         (temp, pressure, density, asound))
138     viscosity=(1.0/PSF2NSM)*MetricViscosity(theta)
139     kinematicViscosity=viscosity/density
140     Itxt.write("%6.3f %8.2E\n" %
141         (1.0E6*viscosity, kinematicViscosity))
142
143     Itxt.close()
144
145
146 def ShortUSTable():
147     Itxt = open('us2py.prt', 'w')
148     Itxt.write(' alt sigma delta theta ')
149     Itxt.write(' temp press dens a visc k.visc ratio\n')
150     Itxt.write(' Kft ')
151     Itxt.write(' degR psf s/cu.ft fps s/ft-sec sq.ft/s 1/ft\n')
152
153     for i in range(-1, 66):
154         altKm=i*FT2METERS
155         (sigma, delta, theta) = SimpleAtmosphere(altKm)
156         Itxt.write("%4i %6.4f %6.4f %6.4f" %
157             (i, sigma, delta, theta))
158         temp=KELVIN2RANKINE*TZERO*theta
159         pressure=PZERO*delta/47.88
160         density=RHOZERO*sigma/515.379
161         asound=(AZERO/FT2METERS)*math.sqrt(theta);
162         Itxt.write("%6.1f %6.1f %9.7f %6.1f" %
163             (temp, pressure, density, asound))
164         viscosity=(1.0/PSF2NSM)*MetricViscosity(theta)
165         kinematicViscosity=viscosity/density
166         vratio=asound/kinematicViscosity
167         Itxt.write("%6.3f %8.2E %4.2f\n"%
168             (1.0E6*viscosity, kinematicViscosity, 1.0E-6*vratio))
169
170     Itxt.close()
171
172
173 def LongSITable():
174     Itxt = open('si1py.prt', 'w')
175     Itxt.write(' alt sigma delta theta ')
176     Itxt.write(' temp press dens a visc k.visc\n')
177     Itxt.write(' Km ')
178     Itxt.write(' K N/sq.m kg/cu.m m/sec kg/m-s sq.m/s\n')
179
180     for i in range(-1,44):
181         altKm=2*i
182         (sigma, delta, theta) = Atmosphere(altKm)
183         Itxt.write("%4i %8.4E %8.4E %6.4f" %
184             (2*i, sigma,delta,theta))
185         temp=TZERO*theta
186         pressure=PZERO*delta
187         density=RHOZERO*sigma
188         asound=AZERO*math.sqrt(theta)
189         Itxt.write("%6.1f %8.3E %8.3E %5.1f" %
190             (temp,pressure,density,asound))
191         viscosity=MetricViscosity(theta)
192         kinematicViscosity=viscosity/density
193         Itxt.write("%6.2f %8.2E\n" %
194             (1.0E6*viscosity, kinematicViscosity))
195
196     Itxt.close()
197
198
199 def ShortSITable():
200     Itxt = open('si2py.prt', 'w')

```

```

201     Itxt.write(' alt  sigma   delta   theta ')
202     Itxt.write(' temp   press   dens   a      visc   k.visc ratio\n')
203     Itxt.write(' Km          ')
204     Itxt.write(' K   N/sq.m   kcm   m/sec kg/m-s   sq.m/s   1/m\n')
205
206     for i in range(-1, 41):
207         altKm=0.5*i
208         (sigma, delta, theta) = SimpleAtmosphere(altKm)
209         Itxt.write("%4.1f %6.4f %6.4f %6.4f" %
210             (altKm, sigma, delta, theta))
211         temp=TZERO*theta
212         pressure=PZERO*delta
213         density=RHOZERO*sigma
214         asound=AZERO*math.sqrt(theta)
215         Itxt.write("%6.1f %6.0f %5.3f %5.1f" %
216             (temp, pressure,density,asound))
217         viscosity=MetricViscosity(theta)
218         kinematicViscosity=viscosity/density
219         vratio=asound/kinematicViscosity
220         Itxt.write("%6.2f %8.2E %5.2f\n" %
221             (1.0E6*viscosity, kinematicViscosity, 1.0E-6*vratio))
222
223     Itxt.close()
224
225
226 def Atmosphere(alt):
227     """ Compute temperature, density, and pressure in standard atmosphere.
228     Correct to 86 km. Only approximate thereafter.
229     Input:
230     alt geometric altitude, km.
231     Return: (sigma, delta, theta)
232     sigma density/sea-level standard density
233     delta pressure/sea-level standard pressure
234     theta temperature/sea-level std. temperature
235     """
236
237     REARTH = 6369.0      # radius of the Earth (km)
238     GMR = 34.163195
239     NTAB = 8            # length of tables
240
241     htab = [ 0.0, 11.0, 20.0, 32.0, 47.0,
242             51.0, 71.0, 84.852 ]
243     ttab = [ 288.15, 216.65, 216.65, 228.65, 270.65,
244             270.65, 214.65, 186.946 ]
245     ptab = [ 1.0, 2.2336110E-1, 5.4032950E-2, 8.5666784E-3, 1.0945601E-3,
246             6.6063531E-4, 3.9046834E-5, 3.68501E-6 ]
247     gtab = [ -6.5, 0.0, 1.0, 2.8, 0, -2.8, -2.0, 0.0 ]
248
249     h = alt*REARTH/(alt+REARTH) # geometric to geopotential altitude
250
251     i=0; j=len(htab)
252     while (j > i+1):
253         k = (i+j)//2      # this is floor division in Python 3
254         if h < htab[k]:
255             j = k
256         else:
257             i = k
258     tgrad = gtab[i]      # temp. gradient of local layer
259     tbase = ttab[i]      # base temp. of local layer
260     deltah=h-htab[i]    # height above local base
261     tlocal=tbase+tgrad*deltah  # local temperature
262     theta = tlocal/ttab[0] # temperature ratio
263
264     if 0.0 == tgrad:
265         delta=ptab[i]*math.exp(-GMR*deltah/tbase)
266     else:
267         delta=ptab[i]*math.pow(tbase/tlocal, GMR/tgrad)

```

```
268     sigma = delta/theta
269     return ( sigma, delta, theta )
270
271
272 def SimpleAtmosphere(alt):
273     """ Compute temperature, density, and pressure in simplified
274     standard atmosphere.
275
276     Correct to 20 km. Only approximate thereafter.
277
278     Input:
279         alt geometric altitude, km.
280     Return: (sigma, delta, theta)
281         sigma    density/sea-level standard density
282         delta    pressure/sea-level standard pressure
283         theta    temperature/sea-level std. temperature
284     """
285
286     REARTH = 6369.0      # radius of the Earth (km)
287     GMR = 34.163195     # gas constant
288
289     h = alt*REARTH/(alt+REARTH) # geometric to geopotential altitude
290
291     if h<11.0:          # troposphere
292         theta=(288.15-6.5*h)/288.15
293         delta=math.pow(theta, GMR/6.5)
294     else:               # stratosphere
295         theta=216.65/288.15
296         delta=0.2233611*math.exp(-GMR*(h-11.0)/216.65)
297     sigma = delta/theta
298     return ( sigma, delta, theta )
299
300
301 def MetricViscosity(theta):
302     t=theta*TZERO
303     return BETAVISC*math.sqrt(t*t*t)/(t+SUTH)
304
305
306 def main():
307     print ("Executing ", sys.argv[0])
308     print (greeting)
309     print (author)
310     if modifier != "":
311         print ("Modified by ", modifier)
312         print ("    version ", version)
313     LongUSTable()
314     ShortUSTable()
315     LongSITable()
316     ShortSITable()
317
318     print (farewell)
319     print (finalmess)
320
321
322 def ISA(altitude):
323     """
324         Args:
325             altitude [float]: altitude in meters
326     """
327
328     # Convert altitude from meters to kilometers
329     altitude = altitude / 1000
330
331     h0_density_SI = 1.22500
332     h0_pressure_SI = 101325
333     h0_temperature_SI = 288.150
334
335     atm_prop = Atmosphere(altitude)
```

```

335     density_ratio = atm_prop[0]
336     pressure_ratio = atm_prop[1]
337     temperature_ratio = atm_prop[2]
338
339     density = h0_density_SI * density_ratio
340     pressure = h0_pressure_SI * pressure_ratio
341     temperature = h0_temperature_SI * temperature_ratio
342
343     return density, pressure, temperature
344
345
346 if __name__ == "__main__":
347     main()

```

A.3.3 objects.py

```

1 from . import functions
2 from .. import geometry as geo
3 #
4 =====
5
6
7 class PanelHorseShoe(geo.objects.Panel):
8     def __init__(self, xx, yy, zz):
9         super().__init__(xx, yy, zz)
10
11     self.horse_shoe_point_a = self.l_chord_1_4
12     self.horse_shoe_point_b = self.r_chord_1_4
13
14     def induced_velocity(self, target_point, circulation):
15         hs_induced_velocity = functions.horse_shoe_ind_vel(
16             self.horse_shoe_point_a, self.horse_shoe_point_b, target_point, circulation
17         )
18
19         return hs_induced_velocity
20
21     def aero_force(self, circulation, flow_vector, air_density):
22         hs_aero_force = functions.horse_shoe_aero_force(
23             self.horse_shoe_point_a, self.horse_shoe_point_b, circulation, flow_vector,
24             air_density
25         )
26
27         return hs_aero_force
28 #
29 =====

```

A.3.4 vlm.py

```

1 """
2 vortex_lattice_method.py
3
4 Implementation of the vortex lattice method.
5
6 Reference: "Low-Speed Aerodynamics", Second Edition, Joseph Katz and Allen Plotkin
7
8 Author: João Paulo Monteiro Cruvinel da Costa

```

```

9 email: joaopaulomcc@gmail.com / joao.cruvinel@embraer.com.br
10 github: joaopaulomcc
11 """
12 #
=====

13 # IMPORTS
14 import numpy as np
15 import scipy as sc
16 import time
17
18 from numpy import sin, cos, tan, pi
19 import scipy.sparse.linalg as spla
20
21 from .. import mathematics as m
22 from .. import geometry as geo
23 from . import objects
24 from . import functions
25 from numba import jit
26
27 #
=====

28 # VORTEX LATTICE METHOD
29
30 #@jit
31 def create_panel_grid(macro_surface_mesh):
32
33     n_span_panels = 0
34     n_chord_panels = 0
35
36     # Count number of chord and spam panels
37     for surface_mesh in macro_surface_mesh:
38         i, j = np.shape(surface_mesh["xx"])
39         n_chord_panels = i - 1
40         n_span_panels += j - 1
41
42     # Initialize panel grid
43     panel_grid = np.empty((n_chord_panels, n_span_panels), dtype="object")
44
45     # Populate Panel Grid
46     span_index = 0
47     for surface_mesh in macro_surface_mesh:
48         n_x, n_y = np.shape(surface_mesh["xx"])
49         n_x -= 1
50         n_y -= 1
51
52         for i in range(n_x):
53             for j in range(n_y):
54                 xx_slice = surface_mesh["xx"][i : i + 2, j : j + 2]
55                 yy_slice = surface_mesh["yy"][i : i + 2, j : j + 2]
56                 zz_slice = surface_mesh["zz"][i : i + 2, j : j + 2]
57                 panel_grid[i][j + span_index] = objects.PanelHorseShoe(
58                     xx_slice, yy_slice, zz_slice
59                 )
60
61             span_index += n_y
62
63     return panel_grid
64
65
66 #
=====

67
68
69 def flatten(panel_matrix):

```

```

70     """Flatten a matrix into a vector, basicaly the lines are concatenated one into another
71     into a long vector
72     """
73
74     # shape = np.shape(panel_matrix)
75     panel_vector = np.copy(np.reshape(panel_matrix, np.size(panel_matrix)))
76
77     # panel_vector = [item for sublist in panel_matrix for item in sublist]
78
79     return panel_vector
80
81
82 #
-----
```

```

83
84
85 @jit
86 def gamma_solver(influence_coef_matrix, right_hand_side_vector):
87     """Receives a vector of panel objects and the airflow velocity. Using this information
88     calculates the influence matrix, the right hand side velocity vector and solves the
89     resulting
90     linear system. Returns a vector with the circulation for each one of the panels.
91
92     Args:
93
94     Returns:
95     """
96     # Solve linear system using scipy library
97
98     gamma, info = spla.gmres(influence_coef_matrix, right_hand_side_vector)
99
100    # Warn user if the solver can not find a solution for the system
101    if info != 0:
102
103        print("aerodynamics.vlm.gamma_solver : ERROR: Solver did not converge!")
104        return None
105
106    else:
107        return gamma
108
109
110 #
-----
```

```

111
112
113 #@jit
114 def aero_loads(
115     aircraft_aero_mesh,
116     velocity_vector,
117     rotation_vector,
118     attitude_vector,
119     altitude,
120     center,
121     influence_coef_matrix=None,
122 ):
123
124     velocity_field_function = geo.functions.velocity_field_function_generator(
125         velocity_vector, rotation_vector, attitude_vector, center
126     )
127
128     aircraft_panel_vector = np.array([], dtype="object")
129     shapes = []
130     components_panel_grid = []
131     components_panel_vector = []
```

```
132
133     #print("Generating Panel Grid")
134     for component_mesh in aircraft_aero_mesh:
135
136         # Generates components panel vector
137         component_panel_grid = create_panel_grid(component_mesh)
138         component_panel_vector = flatten(component_panel_grid)
139
140         # Adds component panel vector to aircraft
141         aircraft_panel_vector = np.copy(
142             np.append(aircraft_panel_vector, component_panel_vector)
143         )
144         # aircraft_panel_vector += component_panel_vector
145
146         # Save shape for reconstruction
147         shapes.append(np.shape(component_panel_grid))
148
149         # Save panel vector and grid
150         components_panel_grid.append(component_panel_grid)
151         components_panel_vector.append(component_panel_vector)
152
153     # Calculate Influence Coefficient Matrix
154     #print("Calculating Influence Coefficient Matrix")
155     if influence_coef_matrix is None:
156         influence_coef_matrix = calc_influence_matrix(aircraft_panel_vector)
157
158     # Calculate right hand side vector
159     #print("Calculating Right Hand Side Vector")
160     right_hand_side_vector = calc_rhs_vector(
161         aircraft_panel_vector, velocity_field_function
162     )
163
164     # Calculate vortex circulation intensity
165     #print("Solving system to find gamma")
166     gamma_vector = gamma_solver(influence_coef_matrix, right_hand_side_vector)
167
168     if gamma_vector is None:
169
170         print("FATAL ERROR")
171         return None
172
173     # Calculate Local Flow Vector
174
175     flow_vector = calc_local_flow_vector(
176         aircraft_panel_vector,
177         gamma_vector,
178         velocity_vector,
179         rotation_vector,
180         attitude_vector,
181         center,
182     )
183
184     # Calculate Aerodynamic Forces
185     air_density, air_pressure, air_temperature = functions.ISA(altitude)
186
187     force_vector = np.empty(len(gamma_vector), dtype="object")
188
189     #print("Calculating Aerodynamic Forces")
190     for i, panel_info in enumerate(zip(gamma_vector, aircraft_panel_vector)):
191
192         panel_gamma = panel_info[0]
193         panel = panel_info[1]
194
195         # Calculate force acting on panel in the geometrical reference system
196         force_vector[i] = panel.aero_force(panel_gamma, flow_vector[i], air_density)
197
198     # Separate results by component
```

```

199     components_force_vector = []
200     components_force_grid = []
201
202     components_gamma_vector = []
203     components_gamma_grid = []
204
205     first_index = 0
206     for shape in shapes:
207         n_itens = shape[0] * shape[1]
208
209         force_vector_slice = force_vector[(first_index) : (first_index + n_itens)]
210         components_force_vector.append(force_vector_slice)
211         components_force_grid.append(np.reshape(force_vector_slice, shape))
212
213         gamma_vector_slice = gamma_vector[(first_index) : (first_index + n_itens)]
214         components_gamma_vector.append(gamma_vector_slice)
215         components_gamma_grid.append(np.reshape(gamma_vector_slice, shape))
216
217         first_index += n_itens
218
219     return (
220         components_force_vector,
221         components_panel_vector,
222         components_gamma_vector,
223         components_force_grid,
224         components_panel_grid,
225         components_gamma_grid,
226         influence_coef_matrix,
227     )
228
229
230 #
-----
```

```

231
232 @jit
233 def calc_influence_matrix(panel_vector):
234
235     n_panels = len(panel_vector)
236     influence_coef_matrix = np.zeros((n_panels, n_panels))
237
238     # For each colocation point i calculate the influence of panel j with a cirulation of 1
239
240     for i in range(n_panels):
241
242         for j in range(n_panels):
243
244             ind_vel = panel_vector[j].induced_velocity(panel_vector[i].col_point, 1)
245
246             influence_coef_matrix[i][j] = m.dot(ind_vel, panel_vector[i].n)
247
248     return influence_coef_matrix
249
250
251 #
-----
```

```

252
253 @jit
254 def calc_rhs_vector(panel_vector, velocity_field_function):
255
256     n_panels = len(panel_vector)
257     right_hand_side_vector = np.zeros((n_panels, 1))
258
259     # For each colocation point i calculate the influence of panel j with a cirulation of 1
260
261     for i, panel in enumerate(panel_vector):
```

```
262     flow_velocity = velocity_field_function(panel.col_point)
263     right_hand_side_vector[i][0] = -m.dot(flow_velocity, panel.n)
264
265     return right_hand_side_vector
266
267
268 #
-----#
269
270 @jit
271 def calc_panels_ind_velocity(panel_vector, gamma_vector, point):
272
273     total_ind_velocity = np.zeros(3)
274
275     for panel, gamma in zip(panel_vector, gamma_vector):
276
277         ind_velocity = panel.induced_velocity(point, gamma)
278
279         total_ind_velocity += ind_velocity
280
281     return total_ind_velocity
282
283
284 #
-----#
285
286 @jit
287 def calc_local_flow_vector(
288     panel_vector,
289     gamma_vector,
290     velocity_vector,
291     rotation_vector,
292     attitude_vector,
293     attitude_center,
294 ):
295
296     flow_vector = np.empty(len(gamma_vector), dtype="object")
297
298     velocity_field_function = geo.functions.velocity_field_function_generator(
299         velocity_vector, rotation_vector, attitude_vector, attitude_center
300     )
301
302     for i, panel in enumerate(panel_vector):
303
304         # Calculate Flow vector at panel aerodynamic center
305
306         flow_vector[i] = calc_panels_ind_velocity(
307             panel_vector, gamma_vector, panel.aero_center
308         ) + velocity_field_function(panel.aero_center)
309
310     return flow_vector
311
312
313 #
-----#
314
315 @jit
316 def calc_panels_delta_pressure(panel_grid, force_grid):
317
318     n_chord_panels = np.shape(panel_grid)[0]
319     n_spam_panels = np.shape(panel_grid)[1]
320
321     delta_p_grid = np.zeros(np.shape(panel_grid))
322     force_magnitude_grid = np.zeros(np.shape(panel_grid))
```

```

323
324     for i in range(n_chord_panels):
325         for j in range(n_spam_panels):
326             force_magnitude_grid[i][j] = m.norm(force_grid[i][j])
327             delta_p_grid[i][j] = force_magnitude_grid[i][j] / panel_grid[i][j].area
328
329     return delta_p_grid, force_magnitude_grid

```

A.4 *aeroelasticity*

A.4.1 `__init__.py`

```
1 from . import functions
```

A.4.2 `functions.py`

```

1 """
2 Aeroelasticity submodule
3 """
4 import sys
5 import datetime
6 import time
7
8 import numpy as np
9
10 from pyquaternion import Quaternion
11 from numba import jit
12
13 from .. import geometry as geo
14 from .. import aerodynamics as aero
15 from .. import structures as struct
16 from .. import mathematics as m
17 from .. import visualization as vis
18
19 #
=====

20
21 @jit
22 def calculate_loads_to_nodes_weight_matrix(
23     macrosurface_aero_grid, macrosurface_struct_grid, algorithm="closest"
24 ):
25
26     node_vector = geo.functions.create_structure_node_vector(macrosurface_struct_grid)
27     panel_grid = aero.vlm.create_panel_grid(macrosurface_aero_grid)
28     panel_vector = aero.vlm.flatten(panel_grid)
29
30     weight_matrix = np.zeros((len(node_vector), len(panel_vector)))
31
32     if algorithm == "closest":
33
34         closest_node = None
35         min_distance = float("inf")
36
37         for j, panel in enumerate(panel_vector):
38
39             for i, node in enumerate(node_vector):

```

```
41         distance = geo.functions.distance_between_points(
42             panel.aero_center, node.xyz
43         )
44
45         if distance <= min_distance:
46
47             closest_node_index = i
48             min_distance = distance
49
50             weight_matrix[closest_node_index][j] = 1
51
52             closest_node = None
53             min_distance = float("inf")
54
55     return weight_matrix
56
57
58 #
=====

59
60 @jit
61 def calculate_deformation_to_aero_grid_weight_matrix(
62     macrosurface_aero_grid, macrosurface_struct_grid, algorithm="closest"
63 ):
64
65     node_vector = geo.functions.create_structure_node_vector(macrosurface_struct_grid)
66
67     aero_grid = geo.functions.macrosurface_aero_grid_to_single_grid(
68         macrosurface_aero_grid
69     )
70
71     aero_points_vector = geo.functions.grid_to_vector(
72         aero_grid["xx"], aero_grid["yy"], aero_grid["zz"]
73     ).transpose()
74
75     weight_matrix = np.zeros((len(aero_points_vector), len(node_vector)))
76
77     if algorithm == "closest":
78
79         closest_node = None
80         min_distance = float("inf")
81
82         for i, point in enumerate(aero_points_vector):
83
84             for j, node in enumerate(node_vector):
85
86                 distance = geo.functions.distance_between_points(point, node.xyz)
87
88                 if distance <= min_distance:
89
90                     closest_node_index = j
91                     min_distance = distance
92
93                     weight_matrix[i][closest_node_index] = 1
94
95                     closest_node = None
96                     min_distance = float("inf")
97
98     return weight_matrix
99
100
101 #
=====

102
103 @jit
```

```
104 def generated_aero_loads(
105     macrosurface_aero_grid,
106     macrosurface_force_grid,
107     macrosurface_struct_grid,
108     algorithm="closest",
109     weight_matrix=None,
110 ):
111
112     macrosurface_loads = []
113
114     node_vector = geo.functions.create_structure_node_vector(macrosurface_struct_grid)
115     panel_grid = aero.vlm.create_panel_grid(macrosurface_aero_grid)
116     panel_vector = aero.vlm.flatten(panel_grid)
117     force_vector = aero.vlm.flatten(macrosurface_force_grid)
118
119     # Changes force_vector from array of arrays to a single numpy array
120     force_vector = np.stack(force_vector)
121
122     if weight_matrix is None:
123
124         weight_matrix = calculate_loads_to_nodes_weight_matrix(
125             macrosurface_aero_grid, macrosurface_struct_grid, algorithm=algorithm
126         )
127
128     for i, node_line in enumerate(weight_matrix):
129
130         node = node_vector[i]
131
132         node_force = np.zeros(3)
133         node_moment = np.zeros(3)
134
135         for j, panel_weight in enumerate(node_line):
136
137             panel = panel_vector[j]
138
139             force = force_vector[j] * panel_weight
140             r = panel.aero_center - node.xyz
141             moment = m.cross(r, force)
142
143             node_force += force
144             node_moment += moment
145
146             load_components = np.array(
147                 [
148                     node_force[0],
149                     node_force[1],
150                     node_force[2],
151                     node_moment[0],
152                     node_moment[1],
153                     node_moment[2],
154                 ]
155             )
156
157             load = struct.objects.Load(application_node=node, load=load_components)
158
159             macrosurface_loads.append(load)
160
161     return macrosurface_loads
162
163
164 #
#=====
165
166 @jit
167 def deform_aero_grid(
168     macrosurface_aero_grid,
```

```

169     macrosurface_struct_grid,
170     struct_deformations,
171     algorithm="closest",
172     weight_matrix=None,
173 ):
174
175     node_vector = geo.functions.create_structure_node_vector(macrosurface_struct_grid)
176
177     surface_grids_shapes = []
178     for surface_aero_grid in macrosurface_aero_grid:
179         shape = np.shape(surface_aero_grid["xx"])
180         surface_grids_shapes.append(shape)
181
182     aero_grid = geo.functions.macrosurface_aero_grid_to_single_grid(
183         macrosurface_aero_grid
184     )
185
186     aero_points_vector = geo.functions.grid_to_vector(
187         aero_grid["xx"], aero_grid["yy"], aero_grid["zz"]
188     ).transpose()
189
190     if weight_matrix is None:
191
192         weight_matrix = calculate_deformation_to_aero_grid_weight_matrix(
193             macrosurface_aero_grid, macrosurface_struct_grid, algorithm="closest"
194         )
195
196     x_axis = np.array([1.0, 0.0, 0.0])
197     y_axis = np.array([0.0, 1.0, 0.0])
198     z_axis = np.array([0.0, 0.0, 1.0])
199
200     deformed_points_vector = np.zeros(np.shape(aero_points_vector))
201
202     macrosurface_struct_deformations = []
203
204     for node in node_vector:
205         node_deformation = struct_deformations[node.number]
206         macrosurface_struct_deformations.append(node_deformation)
207
208     for i, point_line in enumerate(weight_matrix):
209
210         # Use Node Object to rotate and translate point
211
212         point = aero_points_vector[i]
213         point_node_object = geo.objects.Node(point, Quaternion())
214
215         for j, node_weight in enumerate(point_line):
216
217             node = node_vector[j]
218             deformation = macrosurface_struct_deformations[j]
219
220             # Apply rotations to point in relation to its correspondent structural node
221             x_rot_quat = Quaternion(axis=x_axis, angle=(deformation[3] * node_weight))
222             y_rot_quat = Quaternion(axis=y_axis, angle=(deformation[4] * node_weight))
223             z_rot_quat = Quaternion(axis=z_axis, angle=(deformation[5] * node_weight))
224
225             # X -> Y -> Z rotation
226             # very small rotations are commutative, kind of
227             point_node_object = point_node_object.rotate(x_rot_quat, node.xyz)
228             point_node_object = point_node_object.rotate(y_rot_quat, node.xyz)
229             point_node_object = point_node_object.rotate(z_rot_quat, node.xyz)
230
231             # Apply translation
232             translation_vector = deformation[:3] * node_weight
233             point_node_object = point_node_object.translate(translation_vector)
234
235             deformed_points_vector[i][0] = point_node_object.x

```

```

236     deformed_points_vector[i][1] = point_node_object.y
237     deformed_points_vector[i][2] = point_node_object.z
238
239     x_grid, y_grid, z_grid = geo.functions.vector_to_grid(
240         deformed_points_vector.transpose(), np.shape(aero_grid["xx"]))
241     )
242
243     deformed_macrosurface_single_aero_grid = {"xx": x_grid, "yy": y_grid, "zz": z_grid}
244
245     deformed_macrosurface_aero_grid = []
246
247     slices = []
248     slice_ends = 0
249     for i in range(len(surface_grids_shapes) - 1):
250         span_n_points = surface_grids_shapes[i][1]
251         slice_ends += span_n_points
252         slices.append(slice_ends)
253
254     x_grids = np.split(deformed_macrosurface_single_aero_grid["xx"], slices, axis=1)
255     y_grids = np.split(deformed_macrosurface_single_aero_grid["yy"], slices, axis=1)
256     z_grids = np.split(deformed_macrosurface_single_aero_grid["zz"], slices, axis=1)
257
258     for x_grid, y_grid, z_grid in zip(x_grids, y_grids, z_grids):
259
260         deformed_macrosurface_aero_grid.append(
261             {"xx": x_grid, "yy": y_grid, "zz": z_grid}
262         )
263
264     return deformed_macrosurface_aero_grid
265
266
267 #
=====

268 @jit
269 def generate_aircraft_grids(aircraft_object, aircraft_grid_data):
270
271     macrosurfaces_aero_grids = []
272     macrosurfaces_struct_grids = []
273     macrosurfaces_connections = []
274     macrosurfaces_components = []
275
276     beams_struct_grids = []
277
278     aircraft_components_list = []
279     aircraft_components_nodes_list = []
280     aircraft_connections_list = []
281
282     macrosurfaces_grid_data = aircraft_grid_data["macrosurfaces_grid_data"]
283     beams_grid_data = aircraft_grid_data["beams_grid_data"]
284
285     # Create aerodynamic and structural grids for the aircraft macrosurfaces
286     if aircraft_object.macrosurfaces:
287
288         for i, macrosurface in enumerate(aircraft_object.macrosurfaces):
289
290             grid_data = macrosurfaces_grid_data[i]
291
292             macrosurface_aero_grid, macrosurface_struct_grid = macrosurface.create_grids(
293                 n_chord_panels=grid_data["n_chord_panels"],
294                 n_span_panels_list=grid_data["n_span_panels_list"],
295                 n_beam_elements_list=grid_data["n_beam_elements_list"],
296                 chord_discretization=grid_data["chord_discretization"],
297                 span_discretization_list=grid_data["span_discretization_list"],
298                 torsion_function_list=grid_data["torsion_function_list"],
299                 control_surface_deflection_dict=grid_data[
300

```

```

301             "control_surface_deflection_dict"
302         ],
303     )
304
305     macrosurfaces_aero_grids.append(macrosurface_aero_grid)
306     macrosurfaces_struct_grids.append(macrosurface_struct_grid)
307
308     # Create connections for the macrosurface surfaces
309     macrosurface_connections = struct.fem.create_macrosurface_connections(
310         macrosurface
311     )
312     macrosurfaces_connections.append(macrosurface_connections)
313
314     # Add macrosurface's surfaces to a single vector
315     macrosurfaces_components.append(macrosurface.surface_list)
316
317 else:
318
319     print(
320         "geometry.function.generate_aircraft_grids: ERROR No macrosurfaces were found"
321     )
322     print("Quitting execution...")
323     sys.exit()
324
325 # Add each macrosurface's surfaces, it's nodes, and connections to a single list
326 for (
327     macrosurface_surfaces,
328     macrosurface_surfaces_nodes,
329     macrosurface_connections,
330 ) in zip(
331     macrosurfaces_components, macrosurfaces_struct_grids, macrosurfaces_connections
332 ):
333
334     for connection in macrosurface_connections:
335         aircraft_connections_list.append(connection)
336
337     for surface, surface_nodes in zip(
338         macrosurface_surfaces, macrosurface_surfaces_nodes
339     ):
340
341         aircraft_components_list.append(surface)
342         aircraft_components_nodes_list.append(surface_nodes)
343
344     # Structural grids for the aircraft beams
345 if aircraft_object.beams:
346
347     for i, beam in enumerate(aircraft_object.beams):
348
349         grid_data = beams_grid_data[i]
350         beam_struct_grid = beam.create_grid(n_elements=grid_data["n_elements"])
351
352         beams_struct_grids.append(beam_struct_grid)
353
354     # Add aircraft's beams, and it's nodes to the aircraft components and nodes list
355     for beam, beam_nodes_list in zip(aircraft_object.beams, beams_struct_grids):
356         aircraft_components_list.append(beam)
357         aircraft_components_nodes_list.append(beam_nodes_list)
358
359     # Add the structural connections to the aircraft connections list
360     if aircraft_object.connections:
361         for connection in aircraft_object.connections:
362             aircraft_connections_list.append(connection)
363
364     # Number the aircraft structural nodes
365     struct.fem.number_nodes(
366         aircraft_components_list,
367         aircraft_components_nodes_list,

```

```

368     aircraft_connections_list,
369 )
370
371 if aircraft_object.beams:
372
373     aircraft_grids = {
374         "macrosurfaces_aero_grids": macrosurfaces_aero_grids,
375         "macrosurfaces_struct_grids": macrosurfaces_struct_grids,
376         "beams_struct_grids": beams_struct_grids,
377     }
378
379 else:
380
381     aircraft_grids = {
382         "macrosurfaces_aero_grids": macrosurfaces_aero_grids,
383         "macrosurfaces_struct_grids": macrosurfaces_struct_grids,
384         "beams_struct_grids": None,
385     }
386
387 return aircraft_grids
388
389
390 #
=====
391
392 @jit
393 def generate_aircraft_constraints(aircraft, aircraft_grids, constraints_data_list):
394
395     aircraft_constraints = []
396
397     for constraint_data in constraints_data_list:
398
399         # Run through all surfaces in the aircraft searching for the identifier in each
400         # of the constraints described in constraints_data_list
401
402         for i, macrosurface in enumerate(aircraft.macrosurfaces):
403
404             for j, surface in enumerate(macrosurface.surface_list):
405
406                 if surface.identifier == constraint_data["component_identifier"]:
407
408                     surface_grid = aircraft_grids["macrosurfaces_struct_grids"][i][j]
409
410                     if constraint_data["fixation_point"] == "ROOT":
411
412                         # If root select root node
413                         constraint = struct.objects.Constraint(
414                             application_node=surface_grid[0],
415                             dof_constraints=constraint_data["dof_constraints"],
416                         )
417
418                     elif constraint_data["fixation_point"] == "TIP":
419
420                         # If tip select tip node
421                         constraint = struct.objects.Constraint(
422                             application_node=surface_grid[-1],
423                             dof_constraints=constraint_data["dof_constraints"],
424                         )
425
426                     aircraft_constraints.append(constraint)
427
428         # Run through all beams in the aircraft searching for the identifier in each
429         # of the constraints described in constraints_data_list
430
431         if aircraft.beams:
432

```

```
433     for i, beam in enumerate(aircraft.beams):
434
435         if beam.identifier == constraint_data["component_identifier"]:
436
437             beam_grid = aircraft_grids["beams_struct_grids"][i]
438
439             if constraint_data["fixation_point"] == "ROOT":
440
441                 # If root select root node
442                 constraint = struct.objects.Constraint(
443                     application_node=beam_grid[0],
444                     dof_constraints=constraint_data["dof_constraints"],
445                 )
446
447             elif constraint_data["fixation_point"] == "TIP":
448
449                 # If tip select tip node
450                 constraint = struct.objects.Constraint(
451                     application_node=beam_grid[-1],
452                     dof_constraints=constraint_data["dof_constraints"],
453                 )
454
455             aircraft_constraints.append(constraint)
456
457     return aircraft_constraints
458
459
460 #
=====

461
462
463 def calculate_aircraft_loads(
464     aircraft_object,
465     aircraft_grid_data,
466     flight_condition_data,
467     simulation_options={
468         "flexible_aircraft": True,
469         "status_messages": True,
470         "max_iterations": 50,
471         "bending_convergence_criteria": 0.01,
472         "torsion_convergence_criteria": 0.01,
473         "fem_prop_choice": "ROOT",
474         "interaction_algorithm": "closest",
475         "output_iteration_results": True,
476     },
477     aircraft_constraints_data=None,
478     influence_coef_matrix=None,
479 ):
480
481     output_iter = simulation_options["output_iteration_results"]
482     iteration_results = []
483
484     status = simulation_options["status_messages"]
485     simulation_start_time = time.time()
486
487     if status:
488         print("# Running simulation")
489
490     # Generate aircraft grids
491     if status:
492         print("- Generating aircraft grids ...")
493
494     grid_start_time = time.time()
495
496     macrosurfaces_grid_data = aircraft_grid_data["macrosurfaces_grid_data"]
497     beams_grid_data = aircraft_grid_data["beams_grid_data"]
```

```
498
499     aircraft_grids = generate_aircraft_grids(
500         aircraft_object=aircraft_object, aircraft_grid_data=aircraft_grid_data
501     )
502
503     grid_end_time = time.time()
504
505     aircraft_macrosurfaces_aero_grids = aircraft_grids["macrosurfaces_aero_grids"]
506     aircraft_macrosurfaces_struct_grids = aircraft_grids["macrosurfaces_struct_grids"]
507     aircraft_beams_struct_grids = aircraft_grids["beams_struct_grids"]
508
509     control_node_string = simulation_options["control_node_string"]
510     control_node_number = find_control_node_number(aircraft_object, aircraft_grids,
511         control_node_string)
512
513     if status:
514         print(
515             f"- Generating aircraft grids - Completed in {str(datetime.timedelta(seconds=(grid_end_time - grid_start_time)))}"
516         )
517
518     if simulation_options["flexible_aircraft"]:
519
520         # Generate aircraft FEM elements
521
522         fem_start_time = time.time()
523
524         if status:
525             print(f"- Generating aircraft FEM Elements ...")
526
527         fem_prop_choice = simulation_options["fem_prop_choice"]
528
529         aircraft_fem_elements = struct.fem.generate_aircraft_fem_elements(
530             aircraft=aircraft_object,
531             aircraft_grids=aircraft_grids,
532             prop_choice=fem_prop_choice,
533         )
534
535         aircraft_macrosurfaces_fem_elements = aircraft_fem_elements[
536             "macrosurfaces_fem_elements"
537         ]
538         aircraft_beams_fem_elements = aircraft_fem_elements["beams_fem_elements"]
539
540         fem_end_time = time.time()
541
542         if status:
543             print(
544                 f"- Generating aircraft FEM Elements - Completed in {str(datetime.timedelta(seconds=(fem_end_time - fem_start_time)))}"
545             )
546
547         # Generate aircraft constraints
548
549         aircraft_constraints = generate_aircraft_constraints(
550             aircraft=aircraft_object,
551             aircraft_grids=aircraft_grids,
552             constraints_data_list=aircraft_constraints_data,
553         )
554
555         # Generate fluid/structure interaction matrices
556
557         if status:
558             print(f"- Generating aircraft fluid/structure interaction matrices ...")
559
560             matrix_start_time = time.time()
561
562             loads_to_nodes_macrosurfaces_weight_matrices = []
```

```
562     deformation_to_aero_grid_macrosurfaces_weight_matrices = []
563     interaction_algorithm = simulation_options["interaction_algorithm"]
564
565     for i, macrosurface in enumerate(aircraft_object.macrosurfaces):
566
567         macrosurface_aero_grid = aircraft_macrosurfaces_aero_grids[i]
568         macrosurface_struct_grid = aircraft_macrosurfaces_struct_grids[i]
569
570         loads_to_nodes_matrix = calculate_loads_to_nodes_weight_matrix(
571             macrosurface_aero_grid, macrosurface_struct_grid, interaction_algorithm
572         )
573
574         deformation_to_aero_grid_weight_matrix =
575         calculate_deformation_to_aero_grid_weight_matrix(
576             macrosurface_aero_grid, macrosurface_struct_grid, interaction_algorithm
577         )
578
579         loads_to_nodes_macrosurfaces_weight_matrices.append(loads_to_nodes_matrix)
580         deformation_to_aero_grid_macrosurfaces_weight_matrices.append(
581             deformation_to_aero_grid_weight_matrix
582         )
583
584         matrix_end_time = time.time()
585
586     if status:
587         print(
588             f"-- Generating aircraft fluid/structure interaction matrices - Completed in {str(
589                 (datetime.timedelta(seconds=(matrix_end_time - matrix_start_time)))}"
590             )
591
592     # Aeroelastic Calculation Loop
593
594     aelast_loop_start_time = time.time()
595
596     if status:
597         print(f"-- Running aeroelastic calculation ...")
598
599     iteration_number = 0
600     bending_delta = float("inf")
601     torsion_delta = float("inf")
602
603     max_iterations = simulation_options["max_iterations"]
604     bending_convergence_criteria = simulation_options[
605         "bending_convergence_criteria"
606     ]
607     torsion_convergence_criteria = simulation_options[
608         "torsion_convergence_criteria"
609     ]
610
611     old_deformation = np.array([0, 0, 0, 0, 0, 0])
612
613     while (bending_delta > bending_convergence_criteria) or (
614         torsion_delta > torsion_convergence_criteria
615     ):
616
617         if iteration_number >= max_iterations:
618             if status:
619                 print(
620                     f"      - Maximum number of iterations, {max_iterations}, reached."
621                 )
622             break
623
624         else:
625             iteration_number += 1
626
627         iteration_start_time = time.time()
```

```
627     if status:
628         print(f"      - Iteration {iteration_number}")
629
630     # Calculate aerodynamic loads
631
632     if iteration_number == 1:
633         aircraft_deformed_macrosurfaces_aero_grids = (
634             aircraft_macrosurfaces_aero_grids
635         )
636
637     aero_start_time = time.time()
638     (
639         aircraft_force_vector,
640         aircraft_panel_vector,
641         aircraft_gamma_vector,
642         aircraft_force_grid,
643         aircraft_panel_grid,
644         aircraft_gamma_grid,
645         influence_coef_matrix,
646     ) = aero.vlm.aero_loads(
647         aircraft_aero_mesh=aircraft_deformed_macrosurfaces_aero_grids,
648         velocity_vector=flight_condition_data["translation_velocity"],
649         rotation_vector=flight_condition_data["rotation_velocity"],
650         attitude_vector=flight_condition_data["attitude_angles_deg"],
651         altitude=flight_condition_data["altitude"],
652         center=flight_condition_data["center_of_rotation"],
653         influence_coef_matrix=influence_coef_matrix,
654     )
655     aero_end_time = time.time()
656
657     if iteration_number == 1:
658         original_aircraft_panel_grid = aircraft_panel_grid
659
660     if status:
661         print(
662             f"      . Aerodynamic calculation completed in {str(datetime.timedelta(
663                 seconds=(aero_end_time - aero_start_time)))}"
664         )
665
666     # Calculate structure deformation
667
668     struct_start_time = time.time()
669
670     # Calculate aerodynamic loads in the structure
671     aircraft_macrosurfaces_aero_loads = []
672
673     for (
674         macrosurface_aero_grid,
675         macrosurface_force_grid,
676         macrosurface_struct_grid,
677         loads_to_nodes_weight_matrix,
678     ) in zip(
679         aircraft_macrosurfaces_aero_grids,
680         aircraft_force_grid,
681         aircraft_macrosurfaces_struct_grids,
682         loads_to_nodes_macrosurfaces_weight_matrices,
683     ):
684
685         macrosurface_aero_loads = generated_aero_loads(
686             macrosurface_aero_grid,
687             macrosurface_force_grid,
688             macrosurface_struct_grid,
689             interaction_algorithm,
690             loads_to_nodes_weight_matrix,
691         )
692
693     aircraft_macrosurfaces_aero_loads.append(macrosurface_aero_loads)
```

```
693     # Prepare input for structural solver
694
695     struct_grid = []
696     struct_fem_elements = []
697     struct_loads = []
698     struct_constraints = []
699
700
701     # Add all surfaces grids to a vector
702     for macrosurface_struct_grid in aircraft_macrosurfaces_struct_grids:
703
704         struct_grid.extend(macrosurface_struct_grid)
705
706     for macrosurface_fem_elements in aircraft_macrosurfaces_fem_elements:
707
708         struct_fem_elements.extend(macrosurface_fem_elements)
709
710     # Add all beams to a vector
711     if aircraft_object.beams:
712
713         struct_grid.extend(aircraft_beams_struct_grids)
714
715         struct_fem_elements.extend(aircraft_beams_fem_elements)
716
717     # Add all loads to a vector
718     for macrosurface_aero_loads in aircraft_macrosurfaces_aero_loads:
719
720         struct_loads.extend(macrosurface_aero_loads)
721
722     # Add constraints to a vector
723     struct_constraints.extend(aircraft_constraints)
724
725     # Calculate structure deformations
726     deformations, internal_loads = struct.fem.structural_solver(
727         struct_grid, struct_fem_elements, struct_loads, struct_constraints
728     )
729
730     struct_end_time = time.time()
731     if status:
732         print(
733             f"          . Structural calculation completed in {str(datetime.timedelta(
734                 seconds=(struct_end_time - struct_start_time)))}"
735         )
736
737     # Deform aerodynamic grid
738
739     def_start_time = time.time()
740
741     aircraft_deformed_macrosurfaces_aero_grids = []
742     aircraft_deformed_macrosurfaces_aero_panels = []
743
744     for (
745         macrosurface_struct_grid,
746         macrosurface_aero_grid,
747         deformation_to_aero_grid_weight_matrix,
748     ) in zip(
749         aircraft_macrosurfaces_struct_grids,
750         aircraft_macrosurfaces_aero_grids,
751         deformation_to_aero_grid_macrosurfaces_weight_matrices,
752     ):
753
754         deformed_macrosurface_aero_grid = deform_aero_grid(
755             macrosurface_aero_grid,
756             macrosurface_struct_grid,
757             deformations,
758             weight_matrix=deformation_to_aero_grid_weight_matrix,
759         )
760
761         aircraft_deformed_macrosurfaces_aero_grids.append(deformed_macrosurface_aero_grid)
762
763         for panel in range(len(deformation_to_aero_grid_weight_matrix)):
764
765             aircraft_deformed_macrosurfaces_aero_panels.append(
766                 Panel(
767                     panel,
768                     deformed_macrosurface_aero_grid,
769                     deformation_to_aero_grid_weight_matrix[panel],
770                     deformation_to_aero_grid_weight_matrix[panel].shape[1]
771                 )
772             )
773
774     aircraft_deformed_macrosurfaces_aero_grids = np.array(
775         aircraft_deformed_macrosurfaces_aero_grids
776     )
777
778     aircraft_deformed_macrosurfaces_aero_panels = np.array(
779         aircraft_deformed_macrosurfaces_aero_panels
780     )
781
782     return aircraft_deformed_macrosurfaces_aero_grids, aircraft_deformed_macrosurfaces_aero_panels
```

```

759
760         deformed_macrosurface_aero_panels = aero.vlm.create_panel_grid(
761             deformed_macrosurface_aero_grid
762         )
763
764         aircraft_deformed_macrosurfaces_aero_grids.append(
765             deformed_macrosurface_aero_grid
766         )
767         aircraft_deformed_macrosurfaces_aero_panels.append(
768             deformed_macrosurface_aero_panels
769         )
770
771     def_end_time = time.time()
772     if status:
773         print(
774             f"          . Aerodynamic Grid deformation completed in {str(datetime.
775             timedelta(seconds=(def_end_time - def_start_time)))}"
776         )
777
778     iteration_end_time = time.time()
779
780     # Check convergence
781
782     new_deformation = deformations[control_node_number]
783
784     if np.array_equal(old_deformation, np.zeros([6])):
785         delta_deformation = np.full((6), float("inf"))
786
787     else:
788         delta_deformation = np.abs(new_deformation - old_deformation) / np.abs(
789             old_deformation)
790
791         bending_delta = m.norm(delta_deformation[:3])
792         torsion_delta = m.norm(delta_deformation[3:])
793
794     old_deformation = np.copy(new_deformation)
795
796     if output_iter:
797
798         this_iteration_results = {
799             "iteration_number": iteration_number,
800             "aircraft_deformed_macrosurfaces_aero_grids":
801                 aircraft_deformed_macrosurfaces_aero_grids,
802             "aircraft_macrosurfaces_panel_grid":
803                 aircraft_deformed_macrosurfaces_aero_panels,
804             "aircraft_gamma_grid": aircraft_gamma_grid,
805             "aircraft_force_grid": aircraft_force_grid,
806             "aircraft_struct_deformations": deformations,
807             "aircraft_struct_internal_loads": internal_loads,
808             "deformation_at_control_node": old_deformation,
809             "influence_coef_matrix": influence_coef_matrix,
810         }
811
812
813     iteration_results.append(this_iteration_results)
814
815     print(f"          . Bending Delta: {bending_delta}")
816     print(f"          . Torsion Delta: {torsion_delta}")
817
818     if status:
819         print(
820             f"          . Iteration {iteration_number} completed in {str(datetime.
821             timedelta(seconds=(iteration_end_time - iteration_start_time)))}"
822         )
823
824     elast_loop_end_time = time.time()
825
826     if status:
827         print(

```

```

821         f"- Running aeroelastic calculation - Completed in {str(datetime.timedelta(
822             seconds=(aelast_loop_end_time - aelast_loop_start_time)))}"
823             )
824
825         simulation_end_time = time.time()
826         if status:
827             print(
828                 f"# Running simulation - Completed in {str(datetime.timedelta(seconds=(
829                     simulation_end_time - simulation_start_time)))}"
830             )
831
832         final_results = {
833             "aircraft_deformed_macrosurfaces_aero_grids": aircraft_deformed_macrosurfaces_aero_grids,
834             "aircraft_deformed_macrosurfaces_aero_panels": aircraft_deformed_macrosurfaces_aero_panels,
835             "aircraft_gamma_grid": aircraft_gamma_grid,
836             "aircraft_force_grid": aircraft_force_grid,
837             "aircraft_struct_deformations": deformations,
838             "aircraft_struct_internal_loads": internal_loads,
839             "deformation_at_control_node": old_deformation,
840             "influence_coef_matrix": influence_coef_matrix,
841             "aircraft_original_grids": aircraft_grids,
842             "aircraft_struct_fem_elements": aircraft_fem_elements,
843             "original_aircraft_panel_grid": original_aircraft_panel_grid,
844         }
845
846         if output_iter:
847
848             return final_results, iteration_results
849
850         else:
851
852     else:
853
854         aero_start_time = time.time()
855
856         (
857             aircraft_force_vector,
858             aircraft_panel_vector,
859             aircraft_gamma_vector,
860             aircraft_force_grid,
861             aircraft_panel_grid,
862             aircraft_gamma_grid,
863             influence_coef_matrix,
864         ) = aero.vlm.aero_loads(
865             aircraft_aero_mesh=aircraft_macrosurfaces_aero_grids,
866             velocity_vector=flight_condition_data["translation_velocity"],
867             rotation_vector=flight_condition_data["rotation_velocity"],
868             attitude_vector=flight_condition_data["attitude_angles_deg"],
869             altitude=flight_condition_data["altitude"],
870             center=flight_condition_data["center_of_rotation"],
871             influence_coef_matrix=influence_coef_matrix,
872         )
873
874         aero_end_time = time.time()
875
876         if status:
877             print(
878                 f"          . Aerodynamic calculation completed in {str(datetime.timedelta(seconds
879                     =(aero_end_time - aero_start_time)))}"
880             )
881
882         results = {
883             "aircraft_macrosurfaces_panels": aircraft_panel_grid,

```

```
883     "aircraft_original_grids": aircraft_grids,
884     "aircraft_gamma_grid": aircraft_gamma_grid,
885     "aircraft_force_grid": aircraft_force_grid,
886     "influence_coef_matrix": influence_coef_matrix,
887   }
888
889   return results
890
891 # =====
892
893 @jit
894 def find_control_node_number(aircraft, aircraft_grids, control_node_string):
895
896   component_identifier, node_position = control_node_string.split("-")
897
898   # Run through all surfaces in the aircraft searching for the identifier in each
899   # of the constraints described in constraints_data_list
900
901   for i, macrosurface in enumerate(aircraft.macrosurfaces):
902
903     for j, surface in enumerate(macrosurface.surface_list):
904
905       if surface.identifier == component_identifier:
906
907         surface_grid = aircraft_grids["macrosurfaces_struct_grids"][i][j]
908
909         if node_position == "ROOT":
910
911           # If root select root node
912           node_number = surface_grid[0].number
913
914         elif node_position == "TIP":
915
916           # If tip select tip node
917           node_number = surface_grid[-1].number
918
919       return node_number
920
921   # Run through all beams in the aircraft searching for the identifier in each
922   # of the constraints described in constraints_data_list
923
924   if aircraft.beams:
925
926     for i, beam in enumerate(aircraft.beams):
927
928       if beam.identifier == component_identifier:
929
930         beam_grid = aircraft_grids["beams_struct_grids"][i]
931
932         if node_position == "ROOT":
933
934           # If root select root node
935           node_number = beam_grid[0],
936
937         elif node_position == "TIP":
938
939           # If tip select tip node
940           node_number = beam_grid[-1],
941
942       return node_number
943
944   print("ERROR: Control Node was not found")
945   return None
946
947
```

```

948 #
949 =====
950 @jit
951 def calculate_deformation_table(aircraft_original_grids, aircraft_struct_deformations):
952
953     aircraft_macrosurfaces_deformed_nodes = []
954
955     for macrosurface_struct_grid in aircraft_original_grids["macrosurfaces_struct_grids"]:
956
957         node_vector = geo.functions.create_structure_node_vector(macrosurface_struct_grid)
958
959         deformed_nodes = np.zeros((len(node_vector), 6))
960
961         for i, node in enumerate(node_vector):
962             deformed_nodes[i][0] = node.x + aircraft_struct_deformations[node.number][0]
963             deformed_nodes[i][1] = node.y + aircraft_struct_deformations[node.number][1]
964             deformed_nodes[i][2] = node.z + aircraft_struct_deformations[node.number][2]
965             deformed_nodes[i][3] = aircraft_struct_deformations[node.number][3]
966             deformed_nodes[i][4] = aircraft_struct_deformations[node.number][4]
967             deformed_nodes[i][5] = aircraft_struct_deformations[node.number][5]
968
969         aircraft_macrosurfaces_deformed_nodes.append(deformed_nodes)
970
971     aircraft_beams_deformed_nodes = []
972
973     if aircraft_original_grids["beams_struct_grids"]:
974
975         for beam in aircraft_original_grids["beams_struct_grids"]:
976
977             deformed_nodes = np.zeros((len(beam), 6))
978
979             for i, node in enumerate(beam):
980                 deformed_nodes[i][0] = node.x + aircraft_struct_deformations[node.number][0]
981                 deformed_nodes[i][1] = node.y + aircraft_struct_deformations[node.number][1]
982                 deformed_nodes[i][2] = node.z + aircraft_struct_deformations[node.number][2]
983                 deformed_nodes[i][3] = aircraft_struct_deformations[node.number][3]
984                 deformed_nodes[i][4] = aircraft_struct_deformations[node.number][4]
985                 deformed_nodes[i][5] = aircraft_struct_deformations[node.number][5]
986
987             aircraft_beams_deformed_nodes.append(deformed_nodes)
988
989     return {"aircraft_macrosurfaces_deformed_nodes": aircraft_macrosurfaces_deformed_nodes,
990             "aircraft_beams_deformed_nodes": aircraft_beams_deformed_nodes}

```

A.5 *geometry*

A.5.1 *__init__.py*

```

1 from . import objects
2 from . import functions

```

A.5.2 *functions.py*

```

1 import numpy as np
2 from numpy import sin, cos, tan, arccos, arcsin, arctan
3 from pyquaternion import Quaternion

```

```
4
5 from .. import mathematics as m
6 from numba import jit
7
8 import sys
9
10 #
-----
11
12
13 @jit(nopython=True)
14 def distance_point_to_line(line_point_1, line_point_2, point):
15     """
16     reference: http://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html
17     """
18     x0 = point
19     x1 = line_point_1
20     x2 = line_point_2
21
22     distance = m.norm(m.cross((x0 - x1), (x0 - x2))) / m.norm(x2 - x1)
23
24     return distance
25
26
27 #
-----
28
29
30 @jit(nopython=True)
31 def distance_between_points(point_1, point_2):
32
33     distance = (
34         (point_2[0] - point_1[0]) ** 2
35         + (point_2[1] - point_1[1]) ** 2
36         + (point_2[2] - point_1[2]) ** 2
37     ) ** 0.5
38
39     return distance
40
41
42 #
-----
43
44
45 def discretization(discretization_type, n_points, control_surface_hinge_position=None):
46
47     if discretization_type == "linear":
48         chord_points = np.linspace(0, 1, n_points)
49
50     elif discretization_type == "cos":
51         angles = np.linspace(np.pi, np.pi / 2, n_points)
52         chord_points = cos(angles) + 1
53
54     elif discretization_type == "sin":
55         angles = np.linspace(0, np.pi / 2, n_points)
56         chord_points = sin(angles)
57
58     elif discretization_type == "cos_sim":
59         angles = np.linspace(np.pi, 0, n_points)
60         chord_points = cos(angles) / 2 + 0.5
61
62     # Change discretization in order to put panel division at control surface hinge line
63     if control_surface_hinge_position is not None:
64         chord_points, hinge_index = replace_closest(
```

```
65         chord_points, control_surface_hinge_position
66     )
67 else:
68     hinge_index = None
69
70 return chord_points, hinge_index
71
72
73 #
-----
```

```
74
75
76 def replace_closest(array, value):
77
78     closest_index = (np.abs(array - value)).argmin()
79     new_array = array.copy()
80     new_array[closest_index] = value
81
82 return new_array, closest_index
83
84
85 #
-----
```

```
86
87
88 def grid_to_vector(x_grid, y_grid, z_grid):
89
90     x_vector = np.reshape(x_grid, x_grid.size)[np.newaxis]
91     y_vector = np.reshape(y_grid, y_grid.size)[np.newaxis]
92     z_vector = np.reshape(z_grid, z_grid.size)[np.newaxis]
93
94     points_vector = np.concatenate((x_vector, y_vector, z_vector), axis=0)
95
96 return points_vector
97
98
99 #
-----
```

```
100
101
102 def vector_to_grid(points_vector, shape):
103
104     x_grid = np.reshape(points_vector[0, :], shape)
105     y_grid = np.reshape(points_vector[1, :], shape)
106     z_grid = np.reshape(points_vector[2, :], shape)
107
108 return x_grid, y_grid, z_grid
109
110
111 #
-----
```

```
112
113
114 def rotate_point(point_coord, rot_axis, rot_center, rot_angle, degrees=False):
115     """Rotates a point around an axis
116
117     Args:
118         point_coord [[float, float, float]]: x, y and z coordinates of the points, every column
119             is a point
120         rot_axis [float, float, float]: vector that will be used as rotation axis
121         rot_center [float, float, float]: point that will be used as rotation center
122         rot_angle [float]: angle of rotation in radians (default) or degrees if degrees = True
123         degrees [bool]: True if the user wants to use angles in degrees
```

```

123
124     Returns:
125         point [float, float, float]: coordinates of the rotated point
126     """
127
128     # Converts inputs to numpy arrays, normalizes axis vector
129     rot_center = (rot_center[np.newaxis]).transpose()
130     U = m.normalize(rot_axis)
131
132     if degrees:
133         theta = np.radians(rot_angle)
134     else:
135         theta = rot_angle
136
137     u0 = U[0]
138     u1 = U[1]
139     u2 = U[2]
140
141     # Calculating rotation matrix
142     # reference: https://en.wikipedia.org/wiki/Rotation\_matrix - "Rotation matrix from axis and angle"
143
144     # Identity matrix
145     I = np.identity(3)
146
147     # Cross product matrix
148     CPM = np.array([[0.0, -u2, u1], [u2, 0.0, -u0], [-u1, u0, 0.0]])
149
150     # Tensor product U X U, this is NOT a cross product
151     TP = np.tensordot(U, U, axes=0)
152
153     # Rotation Matrix
154     R = cos(theta) * I + sin(theta) * CPM + (1 - cos(theta)) * TP
155
156     # Calculating rotated point
157
158     # Translates points so rotation center is the origin of the coordinate system
159     point_coord = point_coord - rot_center
160
161     # Rotates all points
162     rotated_points = R @ point_coord
163
164     # Undo translation
165     rotated_points = rotated_points + rot_center
166
167     return rotated_points
168
169
170 #
-----
```

```

171
172
173 def mirror_grid(grid_xx, grid_yy, grid_zz, mirror_plane):
174
175     if mirror_plane == "XY" or mirror_plane == "xy":
176
177         new_grid_zz = -grid_zz
178
179         new_grid_xx = grid_xx
180         new_grid_yy = grid_yy
181
182     elif mirror_plane == "XZ" or mirror_plane == "xz":
183
184         new_grid_yy = np.flip(-grid_yy, axis=1)
185
186         new_grid_xx = np.flip(grid_xx, axis=1)
```

```
187         new_grid_zz = np.flip(grid_zz, axis=1)
188
189     elif mirror_plane == "YZ" or mirror_plane == "yz":
190
191         new_grid_xx = np.flip(-grid_xx, axis=0)
192
193         new_grid_yy = np.flip(grid_yy, axis=0)
194         new_grid_zz = np.flip(grid_zz, axis=0)
195
196     else:
197         print("ERROR: Mirror plane not recognized")
198         return None
199
200     return new_grid_xx, new_grid_yy, new_grid_zz
201
202
203 #
-----
```

```
204
205
206 def translate_grid(
207     grid_xx, grid_yy, grid_zz, final_point, start_point=np.array([0, 0, 0])
208 ):
209
210     translation_vector = final_point - start_point
211     x_translation = translation_vector[0]
212     y_translation = translation_vector[1]
213     z_translation = translation_vector[2]
214
215     new_grid_xx = grid_xx + x_translation
216     new_grid_yy = grid_yy + y_translation
217     new_grid_zz = grid_zz + z_translation
218
219     return new_grid_xx, new_grid_yy, new_grid_zz
220
221
222 #
-----
```

```
223
224
225 def rotate_grid(grid_xx, grid_yy, grid_zz, rot_axis, rot_center, rot_angle):
226
227     points = grid_to_vector(grid_xx, grid_yy, grid_zz)
228
229     rot_points = rotate_point(points, rot_axis, rot_center, rot_angle)
230
231     shape = np.shape(grid_xx)
232     new_grid_xx, new_grid_yy, new_grid_zz = vector_to_grid(rot_points, shape)
233
234     return new_grid_xx, new_grid_yy, new_grid_zz
235
236
237 #
-----
```

```
238
239
240 def connect_surface_grid(
241     surface_list,
242     marco_surface_incidence,
243     macro_surface_position,
244     n_chord_panels,
245     n_span_panels_list,
246     chord_discretization,
247     span_discretization_list,
```

```

248     torsion_function_list,
249     torsion_center,
250     control_surface_dictionary,
251 ):
252
253     connected_grids = []
254
255     for i, surface in enumerate(surface_list):
256
257         n_span_panels = n_span_panels_list[i]
258         span_discretization = span_discretization_list[i]
259         torsion_function = torsion_function_list[i]
260
261         if surface.identifier in control_surface_dictionary:
262             control_surface_deflection = control_surface_dictionary[surface.identifier]
263         else:
264             control_surface_deflection = 0
265
266         apply_torsion = False
267
268         # Generates surface mesh, with torsion and dihedral
269         surface_mesh_xx, surface_mesh_yy, surface_mesh_zz = surface.generate_aero_mesh(
270             n_span_panels,
271             n_chord_panels,
272             control_surface_deflection,
273             chord_discretization,
274             span_discretization,
275             apply_torsion,
276             torsion_function,
277             torsion_center,
278         )
279
280         # When the surface is not at the root
281         if i != 0:
282
283             last_surface = connected_grids[i - 1]
284             shape = np.shape(last_surface["xx"])
285
286             # Get tip line segment from last surface
287             tip_xx = last_surface["xx"][:, shape[1] - 1]
288             tip_yy = last_surface["yy"][:, shape[1] - 1]
289             tip_zz = last_surface["zz"][:, shape[1] - 1]
290
291             tip_lead_edge = np.array([tip_xx[0], tip_yy[0], tip_zz[0]])
292             tip_trai_edge = np.array([tip_xx[1], tip_yy[1], tip_zz[1]])
293
294             tip_vector = tip_trai_edge - tip_lead_edge
295
296             # Translate surface so it's root leading edge contacts the tip leading edge of the
297             last
298                 # surface
299                 final_point = tip_lead_edge
300                 surface_mesh_xx, surface_mesh_yy, surface_mesh_zz = translate_grid(
301                     surface_mesh_xx, surface_mesh_yy, surface_mesh_zz, final_point
302                 )
303
304             # Get root line segment from current surface
305             root_xx = surface_mesh_xx[:, 0]
306             root_yy = surface_mesh_yy[:, 0]
307             root_zz = surface_mesh_zz[:, 0]
308
308             root_lead_edge = np.array([root_xx[0], root_yy[0], root_zz[0]])
309             root_trai_edge = np.array([root_xx[1], root_yy[1], root_zz[1]])
310
311             root_vector = root_trai_edge - root_lead_edge
312
313             # use cross vector to find rot axis

```

```

314         rot_axis = m.cross(root_vector, tip_vector)
315
316         # find by which angle the surface needs to be rotated
317         rot_angle = angle_between(root_vector, tip_vector)
318
319         # Rotates surface around tip_lead_edge so tip_vector and root_vector match
320         rot_center = tip_lead_edge
321
322         surface_mesh_xx, surface_mesh_yy, surface_mesh_zz = rotate_grid(
323             surface_mesh_xx,
324             surface_mesh_yy,
325             surface_mesh_zz,
326             rot_axis,
327             rot_center,
328             rot_angle,
329         )
330
331     else:
332         # Translates grid to correct position
333         final_point = macro_surface_position
334         surface_mesh_xx, surface_mesh_yy, surface_mesh_zz = translate_grid(
335             surface_mesh_xx, surface_mesh_yy, surface_mesh_zz, final_point
336         )
337
338         # Apply macro surface incidence angle, other surfaces will automatically have this
339         # incidence
340         rot_axis = np.array([0, 1, 0]) # Y axis
341         rot_center = macro_surface_position
342         rot_angle = macro_surface_incidence
343         surface_mesh_xx, surface_mesh_yy, surface_mesh_zz = rotate_grid(
344             surface_mesh_xx,
345             surface_mesh_yy,
346             surface_mesh_zz,
347             rot_axis,
348             rot_center,
349             rot_angle,
350         )
351
352         connected_grids.append(
353             {"xx": surface_mesh_xx, "yy": surface_mesh_yy, "zz": surface_mesh_zz}
354         )
355
356     return connected_grids
357
358 #
-----
```

```

359
360
361 def connect_surface_nodes(
362     surface_list,
363     n_elements_list,
364     position,
365     incidence_angle,
366     torsion_center,
367     mirror=False,
368 ):
369
370     connected_nodes = []
371
372     for i, surface in enumerate(surface_list):
373
374         n_elements = n_elements_list[i]
375         n_nodes = n_elements + 1
376
377         # Generates surface mesh, with torsion and dihedral
```

```

378         surface_nodes = surface.generate_structure_nodes(
379             n_nodes, torsion_center, mirror
380         )
381
382         # When the surface is not at the root
383         if i != 0:
384
385             last_surface_nodes = connected_nodes[i - 1]
386             tip_node = last_surface_nodes[len(last_surface_nodes) - 1]
387
388             # Translate nodes so it's root leading edge contacts the tip leading edge of the
389             # last surface
390             translation_vector = tip_node.xyz - surface_nodes[0].xyz
391
392             for j, node in enumerate(surface_nodes):
393
394                 surface_nodes[j] = node.translate(translation_vector)
395
396             # use cross vector to find rot axis between tip node and root node z axis
397             rot_axis = m.cross(surface_nodes[0].z_axis, tip_node.z_axis)
398
399             # find by which angle the nodes needs to be rotates
400             rot_angle = angle_between(surface_nodes[0].z_axis, tip_node.z_axis)
401
402             # Rotates surface around tip_lead_edge so tip z vector and root z vector match
403
404             rot_center = tip_node.xyz
405             rot_quaternion = Quaternion(axis=rot_axis, angle=rot_angle)
406
407             for j, node in enumerate(surface_nodes):
408
409                 surface_nodes[j] = node.rotate(rot_quaternion, rot_center)
410
411         else:
412             # Translates grid to correct position
413             final_point = position
414
415             for j, node in enumerate(surface_nodes):
416
417                 surface_nodes[j] = node.translate(position)
418
419             # Apply macro surface incidence angle, other surfaces will automatically have this
420             # incidence
421             rot_axis = np.array([0, 1, 0]) # Y axis
422             rot_center = position
423             rot_angle = incidence_angle
424             rot_quaternion = Quaternion(axis=rot_axis, angle=rot_angle)
425
426             for j, node in enumerate(surface_nodes):
427                 surface_nodes[j] = node.rotate(rot_quaternion, rot_center)
428
429             connected_nodes.append(surface_nodes)
430
431
432 #
-----
```

```

433
434
435 def velocity_field_function_generator(
436     velocity_vector, rotation_vector, attitude_vector, center
437 ):
438
439     # This is a horrible hack
440
```

```
441     v_x = velocity_vector[0]
442     v_y = velocity_vector[0]
443     v_z = velocity_vector[0]
444
445     r_x = rotation_vector[0]
446     r_y = rotation_vector[1]
447     r_z = rotation_vector[2]
448
449     alpha = attitude_vector[0]
450     beta = attitude_vector[1]
451     gamma = attitude_vector[2]
452
453     x_axis = np.array([1.0, 0.0, 0.0])
454     y_axis = np.array([0.0, 1.0, 0.0])
455     z_axis = np.array([0.0, 0.0, 1.0])
456     origin = np.array([0.0, 0.0, 0.0])
457
458     true_airspeed = velocity_vector[0]
459
460     cg_velocity = np.array([true_airspeed, 0, 0])[np.newaxis].transpose()
461
462     # Rotate around y for alfa
463     cg_velocity = rotate_point(cg_velocity, y_axis, origin, -alpha, degrees=True)
464
465     # Rotate around z for beta
466     cg_velocity = rotate_point(cg_velocity, z_axis, origin, -beta, degrees=True)
467
468     # Rotate around x for gamma
469     cg_velocity = rotate_point(cg_velocity, x_axis, origin, -gamma, degrees=True)
470
471     cg_velocity = cg_velocity.transpose()[0]
472
473     def velocity_field_function(point_location):
474
475         r = point_location - center
476         tangential_velocity = -m.cross(rotation_vector, r)
477
478         flow_velocity = cg_velocity + tangential_velocity
479
480         return flow_velocity
481
482     return velocity_field_function
483
484
485 #
-----
```



```
486
487
488     def angle_between(vector_1, vector_2):
489
490         cos_theta = m.dot(vector_1, vector_2) / (m.norm(vector_1) * m.norm(vector_2))
491         theta = np.arccos(cos_theta)
492
493         return theta
494
495
496 #
-----
```



```
497
498
499     def cos_between(vector_1, vector_2):
500
501         cos_theta = m.dot(vector_1, vector_2) / (m.norm(vector_1) * m.norm(vector_2))
502
503         return cos_theta
```

```
504
505
506 #
-----#
507
508
509 def interpolate_nodes(node_1, node_2, n_nodes):
510
511     node_list = []
512
513     for i, quaternion in enumerate(
514         Quaternion.intermediates(
515             node_1.quaternion, node_2.quaternion, (n_nodes - 2), include_endpoints=True
516         )
517     ):
518
519         vector = node_2.xyz - node_1.xyz
520
521         node_xyz = node_1.xyz + i * vector / (n_nodes - 1)
522         node_quaternion = quaternion
523
524         node_list.append([node_xyz, node_quaternion])
525
526     return node_list
527
528
529 #
-----#
530
531
532 def change_coord_sys(vector, X, Y, Z):
533
534     base_matrix = np.array([[X[0], Y[0], Z[0]], [X[1], Y[1], Z[1]], [X[2], Y[2], Z[2]]])
535
536     transformation_matrix = np.linalg.inv(base_matrix)
537
538     new_vector = transformation_matrix @ vector
539
540     return new_vector
541
542
543 #
-----#
544
545
546 def decompose_rotation(rotation_axis, rotation_angle, axis_1, axis_2, axis_3):
547
548     transformed_rotation_axis = change_coord_sys(rotation_axis, axis_1, axis_2, axis_3)
549
550     rotation_quaternion = Quaternion(
551         axis=transformed_rotation_axis, angle=rotation_angle
552     )
553
554     yaw_pitch_roll = rotation_quaternion.yaw_pitch_roll
555
556     return yaw_pitch_roll
557
558
559 #
-----#
560
561
562 def apply_torsion_to_grid(grid_dict, torsion_center, torsion_function, surface):
```

```

563
564     xx = grid_dict["xx"]
565     yy = grid_dict["yy"]
566     zz = grid_dict["zz"]
567
568     n_span_points = len(xx[0, :])
569     n_chord_points = len(xx[:, 0])
570
571     grid_points_xx = np.zeros(np.shape(xx))
572     grid_points_yy = np.zeros(np.shape(yy))
573     grid_points_zz = np.zeros(np.shape(zz))
574
575     for i in range(n_span_points):
576         # Extract section points from grid
577         section_points_x = xx[:, i]
578         section_points_y = yy[:, i]
579         section_points_z = zz[:, i]
580
581         # Convert points from grid to list
582         section_points = grid_to_vector(
583             section_points_x, section_points_y, section_points_z
584         )
585
586         # Calculate rotation characteristics and apply rotation
587         span_position = abs(section_points_y[0]) / (
588             surface.length * np.cos(surface.dihedral_angle_rad)
589         )
590         rot_angle = torsion_function(span_position)
591         rot_axis = np.array([0, 1, 0]) # Y axis
592
593         # Calculate Rotation center
594         section_point_1 = section_points[:, 0]
595         section_point_2 = section_points[:, 1]
596         local_chord = surface.root_chord + span_position * (
597             surface.tip_chord - surface.root_chord
598         )
599
600         section_vector = m.normalize(section_point_2 - section_point_1)
601         rot_center = section_point_1 + torsion_center * section_vector * local_chord
602
603         rot_section_points = rotate_point(
604             section_points, rot_axis, rot_center, rot_angle
605         )
606
607         # Convert section points from list to grid
608         shape = (n_chord_points, 1)
609         rot_section_points_x, rot_section_points_y, rot_section_points_z = vector_to_grid(
610             rot_section_points, shape
611         )
612
613         # Paste rotated section into grid
614         grid_points_xx[:, i] = rot_section_points_x[:, 0]
615         grid_points_yy[:, i] = rot_section_points_y[:, 0]
616         grid_points_zz[:, i] = rot_section_points_z[:, 0]
617
618     return {"xx": grid_points_xx, "yy": grid_points_yy, "zz": grid_points_zz}
619
620
621 #
-----
```

```

622
623
624 def apply_torsion_to_nodes(nodes_list, torsion_center, torsion_function, surface):
625
626     n_span_points = len(nodes_list)
627

```

```

628     rot_nodes_prop_list = []
629
630     for i, node in enumerate(nodes_list):
631
632         # Calculate wing properties at node location
633         span_position = abs(node.xyz[1]) / (
634             surface.length * np.cos(surface.dihedral_angle_rad)
635         )
636
637         local_chord = surface.root_chord + span_position * (
638             surface.tip_chord - surface.root_chord
639         )
640
641         # Calculate position of the torsion center and rotation properties
642         leading_edge_x = (
643             span_position * surface.length * tan(surface.leading_edge_sweep_angle_rad)
644         )
645         leading_edge_y = span_position * surface.span
646         leading_edge_z = span_position * surface.span * tan(surface.dihedral_angle_rad)
647
648         rot_angle = torsion_function(span_position)
649         rot_axis = np.array([0, 1, 0]) # Y axis
650         rot_center = np.array(
651             [
652                 leading_edge_x + local_chord * torsion_center,
653                 leading_edge_y,
654                 leading_edge_z,
655             ]
656         )
657         rot_quaternion = Quaternion(axis=rot_axis, angle=rot_angle)
658
659         # Rotate Node around torsion center
660
661         node_location = node.rotate(
662             rotation_quaternion=rot_quaternion, rotation_center=rot_center
663         )
664
665         yaw_pitch_row = decompose_rotation(
666             rot_axis, rot_angle, node.x_axis, node.y_axis, node.z_axis
667         )
668
669         rot_quaternion = Quaternion(axis=node.x_axis, angle=yaw_pitch_row[2])
670         rot_center = node.xyz
671
672         node_rotation = node.rotate(
673             rotation_quaternion=rot_quaternion, rotation_center=rot_center
674         )
675
676         rot_node_prop = [node_location.xyz, node_rotation.quaternion]
677
678         rot_nodes_prop_list.append(rot_node_prop)
679
680     return rot_nodes_prop_list
681
682
683 #
-----
```

```

684
685
686 def create_structure_node_vector(structure_struct_grid):
687
688     node_vector = []
689
690
691     # Add all nodes to a vector and sort then by node number and remove duplicates
692
```

```
693     for component_grid in structure_struct_grid:
694         node_vector += component_grid
695
696     # Sort the vector
697     node_vector.sort(key=lambda x: x.number)
698
699     last_node_number = None
700     remove_queue = []
701
702     # Find nodes with the same node number
703     for i, node in enumerate(node_vector):
704
705         if node.number == last_node_number:
706             remove_queue.append(i)
707         else:
708             last_node_number = node.number
709
710     # Delete duplicate nodes
711     # Iterate backwards so indice number don't change
712     for i in reversed(remove_queue):
713         del node_vector[i]
714
715     return node_vector
716
717 #
-----
```



```
718
719
720 def macrosurface_aero_grid_to_single_grid(macro_surface_mesh):
721
722     n_span_points = 0
723     n_chord_points = 0
724
725     # Count number of chord and spam points
726     for surface_mesh in macro_surface_mesh:
727         i, j = np.shape(surface_mesh["xx"])
728         n_chord_points = i
729         n_span_points += j
730
731     # Initialize single grid
732     single_grid_xx = np.zeros((n_chord_points, n_span_points))
733     single_grid_yy = np.zeros((n_chord_points, n_span_points))
734     single_grid_zz = np.zeros((n_chord_points, n_span_points))
735
736     # Populate Single Grid
737     span_index = 0
738     for surface_mesh in macro_surface_mesh:
739         n_x, n_y = np.shape(surface_mesh["xx"])
740
741         for i in range(n_x):
742             for j in range(n_y):
743                 single_grid_xx[i][j + span_index] = surface_mesh["xx"][i, j]
744                 single_grid_yy[i][j + span_index] = surface_mesh["yy"][i, j]
745                 single_grid_zz[i][j + span_index] = surface_mesh["zz"][i, j]
746
747             span_index += n_y
748
749     single_grid = {"xx":single_grid_xx, "yy":single_grid_yy, "zz":single_grid_zz}
750
751     return single_grid
752
753 #
```

A.5.3 objects.py

```
1 import numpy as np
2 import scipy as sc
3
4 from numpy import sin, cos, tan, pi
5 from pyquaternion import Quaternion
6
7 from . import functions as f
8 from .. import mathematics as m
9
10 #
=====

11 # OBJECTS
12
13
14 class Aircraft(object):
15
16     def __init__(self,
17                  name,
18                  macrosurfaces,
19                  ref_area,
20                  mean_aero_chord,
21                  beams=None,
22                  engines=None,
23                  inertial_properties=None,
24                  connections=None,
25                  ):
26
27
28         self.name = name
29         self.macrosurfaces = macrosurfaces
30         self.beams = beams
31         self.engines = engines
32         self.inertial_properties = inertial_properties
33         self.connections = connections
34         self.ref_area = ref_area
35         self.mean_aero_chord = mean_aero_chord
36
37
38 #
=====

39
40
41 class Engine(object):
42     def __init__(self,
43                  identifier,
44                  position,
45                  orientation_quaternion,
46                  inertial_properties,
47                  thrust_function,
48                  ):
49
50         self.identifier = identifier
51         self.orientation_quaternion = orientation_quaternion
52         self.position = position
53         self.orientation_quaternion = orientation_quaternion
54         self.node = Node(position, orientation_quaternion)
55         self.inertial_properties = inertial_properties
56         self.thrust_function = thrust_function
57         self.thrust_vector = self.orientation_quaternion.rotate(np.array([1, 0, 0]))
58
59     def thrust(self, throttle, parameters):
60         thrust_force = self.thrust_function(throttle, parameters) * self.thrust_vector
61
```

```
62         return thrust_force
63
64
65 # =====
66
67
68 class MaterialPoint(object):
69     def __init__(self,
70                  identifier=None,
71                  orientation_quaternion=Quaternion(),
72                  mass=0,
73                  position=np.array([0, 0, 0]),
74                  Ixx=0,
75                  Iyy=0,
76                  Izz=0,
77                  Ixy=0,
78                  Ixz=0,
79                  Iyz=0,
80                  ):
81         self.identifier = identifier
82         self.orientation_quaternion = orientation_quaternion
83         self.mass = mass
84         self.position = position
85         self.node = Node(position, orientation_quaternion)
86         self.Ixx = Ixx
87         self.Iyy = Iyy
88         self.Izz = Izz
89         self.Ixy = Ixy
90         self.Ixz = Ixz
91         self.Iyz = Iyz
92
93
94
95 #
# =====
96
97
98 class Node(object):
99     """Node object, defines a node in the FEM mesh.
100
101    Args:
102        xyz (np.array([3], dtype=float)): array with x, y and z coordinates of the node in 3D
103        space
104        quaternion (pyquaternion.Quaternion): quaternion object with node orientation
105        information
106
107    Attributes:
108        x (float): node x position in the 3D space.
109        y (float): node y position in the 3D space.
110        z (float): node z position in the 3D space.
111        xyz (np.array([3], dtype=float)): array with x, y and z coordinates of the node in 3D
112        space
113        quaternion (pyquaternion.Quaternion): quaternion object with node orientation
114        information
115        x_axis (np.array([3], dtype=float)): numpy array with the node's x_axis
116        y_axis (np.array([3], dtype=float)): numpy array with the node's y_axis
117        z_axis (np.array([3], dtype=float)): numpy array with the node's z_axis
118        x_cos (float): Direction cossine of the node's x_axis in relation to the global x_axis
119        y_cos (float): Direction cossine of the node's y_axis in relation to the global y_axis
120        z_cos (float): Direction cossine of the node's z_axis in relation to the global z_axis
121        direction_cos (np.array([3], dtype=float)): array with x_axis, y_axis and z_axis
122        direction
123        cossines
124        number (None or int): Node number, is initialized as None
```

```

120     """
121
122     def __init__(self, xyz, quaternion):
123
124         self.x = xyz[0]
125         self.y = xyz[1]
126         self.z = xyz[2]
127         self.xyz = np.array([self.x, self.y, self.z])
128         self.quaternion = quaternion
129
130         orig_x_axis = np.array([1, 0, 0])
131         orig_y_axis = np.array([0, 1, 0])
132         orig_z_axis = np.array([0, 0, 1])
133
134         self.x_axis = quaternion.rotate(orig_x_axis)
135         self.y_axis = quaternion.rotate(orig_y_axis)
136         self.z_axis = quaternion.rotate(orig_z_axis)
137
138         self.x_cos = f.cos_between(orig_x_axis, self.x_axis)
139         self.y_cos = f.cos_between(orig_y_axis, self.y_axis)
140         self.z_cos = f.cos_between(orig_z_axis, self.z_axis)
141         self.direction_cos = np.array([self.x_cos, self.y_cos, self.z_cos])
142
143         self.number = None
144
145     def translate(self, translation_vector):
146         """Applies a translation to the node, returns a new Node object with the transformed
147         coordinates
148
149         Args:
150             translation_vector (np.array([3], dtype=float)): numpy array with the translation
151             vector
152
153         Returns:
154             Node (geometry.objects.Node): Node object with a new node with the required
155             transformation
156
157         new_xyz = self.xyz + translation_vector
158
159         return Node(new_xyz, self.quaternion)
160
161     def rotate(self, rotation_quaternion, rotation_center=np.array([0, 0, 0])):
162         """Applies a rotation to the node, returns a new Node object with the transformed
163         coordinates
164
165         Args:
166             rotation_quaternion (pyquaternion.Quaternion): quaternion object with the rotation
167             to be
168                 applied
169             rotation_center (np.array([3], dtype=float)): point around which the node will be
170             rotated
171
172         Returns:
173             Node (geometry.objects.Node): Node object with a new node with the required
174             transformation
175
176         # Coordinate transformation to move rotation center to origin
177         temp_xyz = self.xyz - rotation_center
178
179         # Rotate Node
180         new_quaternion = rotation_quaternion * self.quaternion
181         rot_xyz = rotation_quaternion.rotate(temp_xyz)
182
183         # Coordinate transformation to return rotation center to it's original position
184         new_xyz = rot_xyz + rotation_center

```

```
184         return Node(new_xyz, new_quaternion)
185
186
187
188 #
=====

189
190
191 class Beam:
192     def __init__(self, identifier, root_point, tip_point, orientation_vector, ElementProperty):
193
194         self.identifier = identifier
195         self.root_point = root_point
196         self.tip_point = tip_point
197         self.orientation_vector = orientation_vector
198         self.ElementProperty = ElementProperty
199         self.vector = m.normalize(self.tip_point - self.root_point)
200         self.L = m.norm(self.tip_point - self.root_point)
201
202     def create_grid(self, n_elements):
203
204         n_nodes = n_elements + 1
205
206         # Calculate rotation quaternion
207         node = Node(np.array([0.0, 0.0, 0.0]), Quaternion())
208         x_axis = np.array([1, 0, 0])
209         y_axis = np.array([0, 1, 0])
210         z_axis = np.array([0, 0, 1])
211
212         # Rotate to align x axis with beam vector
213
214         rot_axis = m.cross(x_axis, self.vector)
215
216         if np.array_equal(rot_axis, np.zeros(3)):
217             rot_axis = x_axis
218
219         rot_angle = f.angle_between(x_axis, self.vector)
220         rot_quaternion_1 = Quaternion(axis=rot_axis, angle=rot_angle)
221
222         node = node.rotate(rot_quaternion_1, rotation_center=np.zeros(3))
223
224         # Calculate z axis based on orientation axis
225         node_z_axis = m.cross(node.x_axis, self.orientation_vector)
226
227         # Rotate to align node z axis with calculated z axis
228         rot_axis = m.cross(node.z_axis, node_z_axis)
229         rot_angle = f.angle_between(node.z_axis, node_z_axis)
230
231         if np.array_equal(rot_axis, np.zeros(3)):
232             rot_axis = z_axis
233
234         rot_quaternion_2 = Quaternion(axis=rot_axis, angle=rot_angle)
235
236         # Multiply both quaternions to generate final rot_quaternion
237
238         rot_quaternion = rot_quaternion_2 * rot_quaternion_1
239
240         # Calculate Root and Tip nodes
241         root_node = Node(self.root_point, rot_quaternion)
242         tip_node = Node(self.tip_point, rot_quaternion)
243
244         # Interpolate Nodes to generate grid
245         nodes_prop = f.interpolate_nodes(root_node, tip_node, n_nodes)
246
247         nodes_grid = []
248
```

```
249         for node_prop in nodes_prop:
250             nodes_grid.append(Node(node_prop[0], node_prop[1]))
251
252     return nodes_grid
253
254
255 #
=====
256
257
258 class Airfoil(object):
259     def __init__(self, upper_spline, lower_spline, cl_alpha, cd_alpha, cm_alpha):
260         self.upper_spline = upper_spline
261         self.lower_spline = lower_spline
262         self.cl_alpha_spline = cl_alpha
263         self.cd_alpha_spline = cd_alpha
264         self.cm_alpha_spline = cm_alpha
265
266
267 #
=====
268
269
270 class Section(object):
271     def __init__(self, identifier, material, area, Iyy, Izz, J, shear_center):
272         self.identifier = identifier
273         self.material = material
274         self.area = area
275         self.Iyy = Iyy
276         self.Izz = Izz
277         self.J = J
278         self.shear_center = shear_center
279
280
281 #
=====
282
283
284 class Panel(object):
285     """Panel object"""
286
287     def __init__(self, xx, yy, zz):
288         """Args:
289             xx [[float]] = grid with panel points x coordinates
290             yy [[float]] = grid with panel points x coordinates
291             zz [[float]] = grid with panel points x coordinates
292         """
293         self.xx = xx
294         self.yy = yy
295         self.zz = zz
296         self.A = np.array([xx[1][0], yy[1][0], zz[1][0]])
297         self.B = np.array([xx[0][0], yy[0][0], zz[0][0]])
298         self.C = np.array([xx[0][1], yy[0][1], zz[0][1]])
299         self.D = np.array([xx[1][1], yy[1][1], zz[1][1]])
300         self.AC = self.C - self.A
301         self.BD = self.D - self.B
302
303         self.l_chord = self.A - self.B
304         self.l_chord_1_4 = self.B + 0.25 * self.l_chord
305         self.l_chord_3_4 = self.B + 0.75 * self.l_chord
306
307         self.r_chord = self.D - self.C
308         self.r_chord_1_4 = self.C + 0.25 * self.r_chord
309         self.r_chord_3_4 = self.C + 0.75 * self.r_chord
```

```

310
311     self.l_edge = self.C - self.B
312     self.l_edge_1_2 = self.B + 0.5 * self.l_edge
313
314     self.t_edge = self.D - self.A
315     self.t_edge_1_2 = self.A + 0.5 * self.t_edge
316
317     self.col_point = 0.75 * (self.t_edge_1_2 - self.l_edge_1_2) + self.l_edge_1_2
318     self.aero_center = 0.25 * (self.t_edge_1_2 - self.l_edge_1_2) + self.l_edge_1_2
319
320     self.span = m.dot(self.l_edge, np.array([0, 1, 0]))
321     self.n = m.normalize(m.cross(self.BD, self.AC))
322     self.area = m.dot(self.n, m.cross(self.BD, self.AC)) / 2
323
324     infinity = 100000
325     hs_A = np.array(
326         [self.l_chord_1_4[0] + infinity, self.l_chord_1_4[1], self.l_chord_1_4[2]])
327     )
328     hs_B = np.array([self.l_chord_1_4[0], self.l_chord_1_4[1], self.l_chord_1_4[2]])
329     hs_C = np.array([self.r_chord_1_4[0], self.r_chord_1_4[1], self.r_chord_1_4[2]])
330     hs_D = np.array(
331         [self.r_chord_1_4[0] + infinity, self.r_chord_1_4[1], self.r_chord_1_4[2]])
332     )
333     hs_A = hs_A[np.newaxis]
334     hs_B = hs_B[np.newaxis]
335     hs_C = hs_C[np.newaxis]
336     hs_D = hs_D[np.newaxis]
337
338
339 #
=====

340
341
342 class Surface(object):
343     """The surface object is a wing (or similar structure) section.
344
345     Args:
346         identifier (string): Name of the surface
347         root_chord (float): length of the root chord of the surface [m]
348         root_section (geometry.Section): the section object that describes the root section of
349             the
350                 surface
351         tip_chord (float): length of the tip chord of the surface [m]
352         tip_section (geometry.Section): the section object that describes the root section of
353             the
354                 surface
355         length (float): the length of the surface [m]
356         leading_edge_sweep_angle_deg (float): the sweep angle of the leading edge of the surface
357             [z]
358         dihedral_angle_deg (float): the surface dihedral angle [z]
359         tip_torsion_angle_deg (float): the tip torsion angle in relation to the root [z]
360         control_surface_hinge_position (float): relative position of control surface in the
361             chord,
362                 must be between 0 and 1.
363
364     Attributes:
365         identifier (string): Name of the surface
366         root_chord (float): length of the root chord of the surface [m]
367         root_section (geometry.Section): the section object that describes the root section of
368             the
369                 surface
370         tip_chord (float): length of the tip chord of the surface [m]
371         tip_section (geometry.Section): the section object that describes the root section of
372             the
373                 surface
374         length (float): the length of the surface [m]

```

```

369     leading_edge_sweep_angle_deg (float): the sweep angle of the leading edge of the surface
370     [ž]
371     leading_edge_sweep_angle_rad (float): the sweep angle of the leading edge of the surface
372     [rad]
373     quarter_chord_sweep_angle_deg (float): the sweep angle of the surface at quarter chord [ž]
374     quarter_chord_sweep_angle_rad (float): the sweep angle of the surface at quarter chord [rad]
375     dihedral_angle_deg (float): the surface dihedral angle [ž]
376     dihedral_angle_rad (float): the surface dihedral angle [rad]
377     tip_torsion_angle_deg (float): the tip torsion angle in relation to the root [ž]
378     tip_torsion_angle_rad (float): the tip torsion angle in relation to the root [rad]
379     control_surface_hinge_position (float): relative position of control surface in the
380     chord,
381     must be between 0 and 1.
382     span (float): projection of the surface length in the XY plane [m]
383     ref_area (float): area of the surface projection in the XY plane [m^2]
384     true_area (float): real area of the surface [m^2]
385     taper_ratio (float): surface's taper ration
386     aspect_ratio (float): surface's aspect ratio
387     """
388     def __init__(self,
389         identifier,
390         root_chord,
391         root_section,
392         tip_chord,
393         tip_section,
394         length,
395         leading_edge_sweep_angle_deg,
396         dihedral_angle_deg,
397         tip_torsion_angle_deg,
398         control_surface_hinge_position=None,
399     ):
400
401         self.identifier = identifier
402         self.root_section = root_section
403         self.root_chord = float(root_chord)
404         self.tip_section = tip_section
405         self.tip_chord = float(tip_chord)
406         self.tip_section = tip_section
407         self.length = float(length)
408         self.leading_edge_sweep_angle_deg = float(leading_edge_sweep_angle_deg)
409         self.dihedral_angle_deg = float(dihedral_angle_deg)
410         self.tip_torsion_angle_deg = float(tip_torsion_angle_deg)
411         self.control_surface_hinge_position = control_surface_hinge_position
412
413         self.leading_edge_sweep_angle_rad = np.radians(leading_edge_sweep_angle_deg)
414         self.dihedral_angle_rad = np.radians(dihedral_angle_deg)
415         self.tip_torsion_angle_rad = np.radians(tip_torsion_angle_deg)
416
417         # Calculation of the sweep angle at the surface's quarter chord, the formula fails if
418         # the
419         # wing's taper ratio is equal to 1 so an if clause is used
420         if root_chord == tip_chord:
421             # the wing is a parallelogram, so the sweep is the same for the whole chord
422             self.quarter_chord_sweep_ang_rad = self.leading_edge_sweep_angle_rad
423
424         else:
425             self.quarter_chord_sweep_ang_rad = np.arctan(
426                 length
427                 / (
428                     length * tan(self.leading_edge_sweep_angle_rad)
429                     + 0.25 * tip_chord
430                     - 0.25 * root_chord
431                 )
432             )

```

```

431         )
432
433         self.quarter_chord_sweep_ang_deg = np.degrees(self.quarter_chord_sweep_ang_rad)
434
435         self.span = length * cos(self.dihedral_angle_rad)
436         self.ref_area = self.span * (root_chord + tip_chord) / 2
437         self.true_area = length * (root_chord + tip_chord) / 2
438         self.taper_ratio = tip_chord / root_chord
439         self.aspect_ratio = (self.span ** 2) / self.ref_area
440
441     #
442
443     def generate_aero_grid(
444         self,
445         n_span_panels,
446         n_chord_panels,
447         apply_torsion=True,
448         torsion_center=0.0,
449         torsion_function="linear",
450         mirror=False,
451         control_surface_deflection=0,
452         chord_discretization="linear",
453         span_discretization="linear",
454     ):
455         """Generates the surface's aerodynamic and structural grids.
456
457             The aerodynamic grid is returned as a dictionary with three keywords: xx, yy
458             and zz
459             they each contains a 2 dimensional array with the x, y or z coordinates of each of
460             the
461             grid points arranged in space, for example, the upper right element in the xx matrix
462             contains the x coordinate of the leading edge of the wing tip. The easiest way to
463             think
464             about this is: as if you are looking at the surface from above imagine that the a
465             propertie of each point in the grid is written in the surface, than copy this
466             information column by column in a matrix.
467
468             The structure grid is returned as a list of node objects, with the first node being
469             the node at the root of the surface and so on. The difference between a node and a
470             point
471             in space is that the node has an orientation, this is needed for the contruction of
472             the beam finite element.
473
474             The surface is defined as if the root leading edge is located in the origin of the
475             coordinate system and the wing tip is located in the POSITIVE side of the y axis. If
476             the
477             mirror option is set to TRUE the wing tip will be located at the NEGATIVE side of
478             the
479             y axis.
480
481             The structure nodes are oriented so that its x axis points from the root of the
482             surface
483             to the tip and its y axis is parallel to the section chord.
484
485             Args:
486                 n_span_panels (int): number of panels to be created in the surface span direction
487                 n_chord_panels (int): number of panels to be created in the surface chord direction,
488                         must be equal or greater than 2 if a control surface is
489                         present
490                 n_beam_elements (int): number of beam elements to be created in the surface
491                 apply_torsion (bool): if True geometrical torsion will be applied to the surface, if
492                         False only the dihedral will be applied
493                 torsion_center (float): position relative to the chord around which the section will
494                         be
495                         rotated, must be a number between 0 and 1

```

```

487         torsion_function (function): a function that receives a span position, between 0 and
488         1,
489                     and returns a number between 0 and 1 that will be
490                     multiplied by the tip rotation to calculate the section
491                     rotation, if "linear" is supplied a linear function will
492                     be
493                     created and applied.
494         mirror (bool): If false wing tip will be in the positive side of the y axis, if True
495                     wing tip will be in the negative side of the y axis.
496         control_surface_deflection (float): deflection of the control surface, positive in
497                     the
498                     root->tip direction, negative if Mirror is True [z]
499
500         """
501         # Corrects input types
502         n_span_panels = int(n_span_panels)
503         n_chord_panels = int(n_chord_panels)
504         apply_torsion = bool(apply_torsion)
505         torsion_center = float(torsion_center)
506         mirror = bool(mirror)
507         control_surface_deflection = float(control_surface_deflection)
508
509         # Checks if the number of chord panels is valid, if it isn't fix it
510         if (self.control_surface_hinge_position is not None) and (n_chord_panels < 2):
511             n_chord_panels = 2
512             print(
513                 "WARNING: Invalid number of chord panels, number of chord panels set to 2."
514             )
515
516         # Calculate number of grid points
517         n_chord_points = n_chord_panels + 1
518         n_span_points = n_span_panels + 1
519
520         # Generate discretization of chord, a list of numbers from 0 to 1
521         chord_points, hinge_index = f.discretization(
522             chord_discretization, n_chord_points, self.control_surface_hinge_position
523         )
524
525         # Generate a torsion linear torsion function when not supplied with one
526         if torsion_function == "linear":
527             torsion_function = (
528                 lambda span_position: span_position * self.tip_torsion_angle_rad
529             )
530
531         # Find root points by scaling by the root chord
532         root_chord_points_x = chord_points * self.root_chord
533         root_chord_points_y = np.repeat(0, n_chord_points)
534
535         # Find tip points by scaling and translation
536         tip_chord_points_x = chord_points * self.tip_chord + self.length * tan(
537             self.leading_edge_sweep_angle_rad
538         )
539         tip_chord_points_y = np.repeat(self.length, n_chord_points)
540
541         # Find span points by scaling by the surface's length
542         span_points, _ = f.discretization(span_discretization, n_span_points)
543         span_points_y = self.length * span_points
544
545         # Generate root and tip grids for simple calculation of the planar mesh points
546         root_points_xx = np.repeat(
547             root_chord_points_x[np.newaxis].transpose(), n_span_points, axis=1
548         )
549         root_points_yy = np.repeat(

```

```

550         root_chord_points_y[np.newaxis].transpose(), n_span_points, axis=1
551     )
552     tip_points_xx = np.repeat(
553         tip_chord_points_x[np.newaxis].transpose(), n_span_points, axis=1
554     )
555     tip_points_yy = np.repeat(
556         tip_chord_points_y[np.newaxis].transpose(), n_span_points, axis=1
557     )
558
559     # Calculate planar mesh points
560     planar_mesh_points_yy = np.repeat(
561         span_points_y[np.newaxis], n_chord_points, axis=0
562     )
563     planar_mesh_points_xx = root_points_xx + (tip_points_xx - root_points_xx) * (
564         planar_mesh_points_yy - root_points_yy
565     ) / (tip_points_yy - root_points_yy)
566     planar_mesh_points_zz = np.zeros((n_chord_points, n_span_points))
567
568     # Apply control surface rotation
569     if self.control_surface_hinge_position is not None:
570
571         # Hinge Vector and point
572         control_surface_hinge_axis = m.normalize(
573             np.array(
574                 [
575                     tip_chord_points_x[hinge_index]
576                     - root_chord_points_x[hinge_index],
577                     self.length,
578                     0,
579                 ]
580             )
581         )
582         # Hinge Point
583         hinge_point = np.array([root_chord_points_x[hinge_index], 0, 0])
584
585         # Slicing control surface grid
586         control_surface_points_xx = planar_mesh_points_xx[(hinge_index + 1) :, :]
587         control_surface_points_yy = planar_mesh_points_yy[(hinge_index + 1) :, :]
588         control_surface_points_zz = planar_mesh_points_zz[(hinge_index + 1) :, :]
589
590         # Converting grid to points list
591         control_surface_points = f.grid_to_vector(
592             control_surface_points_xx,
593             control_surface_points_yy,
594             control_surface_points_zz,
595         )
596
597         # Rotate control surface points around hinge axis
598         rot_control_surface_points = f.rotate_point(
599             control_surface_points,
600             control_surface_hinge_axis,
601             hinge_point,
602             control_surface_deflection,
603             degrees=True,
604         )
605
606         # Converting points list do grid
607         shape = np.shape(control_surface_points_xx)
608         control_surface_points_xx, control_surface_points_yy, control_surface_points_zz = f.
609         vector_to_grid(
610             rot_control_surface_points, shape
611         )
612
613         # Replacing planar points by rotate control surface points
614         planar_mesh_points_xx[(hinge_index + 1) :, :] = control_surface_points_xx
615         planar_mesh_points_yy[(hinge_index + 1) :, :] = control_surface_points_yy
616         planar_mesh_points_zz[(hinge_index + 1) :, :] = control_surface_points_zz

```

```

616
617     # Apply wing dihedral
618
619     # Convert grid to list
620     mesh_points = f.grid_to_vector(
621         planar_mesh_points_xx, planar_mesh_points_yy, planar_mesh_points_zz
622     )
623
624     # Calculate rotation characteristics and apply rotation
625     rot_angle = self.dihedral_angle_rad
626     rot_axis = np.array([1, 0, 0]) # X axis
627     rot_center = np.array([0, 0, 0])
628
629     rot_mesh_points = f.rotate_point(mesh_points, rot_axis, rot_center, rot_angle)
630
631     # Convert mesh_points from list to grid
632     shape = (n_chord_points, n_span_points)
633     mesh_points_xx, mesh_points_yy, mesh_points_zz = f.vector_to_grid(
634         rot_mesh_points, shape
635     )
636
637     grid_dict = {"xx": mesh_points_xx, "yy": mesh_points_yy, "zz": mesh_points_zz}
638
639     # Apply torsion to surface grid
640     if apply_torsion:
641
642         grid_dict = f.apply_torsion_to_grid(
643             grid_dict, torsion_center, torsion_function, self
644         )
645
646     # Mirror grid if needed
647     if mirror:
648
649         xx, yy, zz = f.mirror_grid(
650             grid_dict["xx"], grid_dict["yy"], grid_dict["zz"], "XZ"
651         )
652         grid_dict = {"xx": xx, "yy": yy, "zz": zz}
653
654     return grid_dict
655
656 #
-----
```

```

657
658     def generate_structure_nodes(
659         self,
660         n_beam_elements,
661         apply_torsion=True,
662         torsion_center=0.0,
663         torsion_function="linear",
664         mirror=False,
665     ):
666         """Generates the surface's structural grids.
667
668         The structure grid is returned as a list of node objects, with the first node being
669         the node at the root of the surface and so on. The difference between a node and a point
670         in space is that the node has an orientation, this is needed for the construction of
671         the beam finite element.
672
673         The surface is defined as if the root leading edge is located in the origin of the
674         coordinate system and the wing tip is located in the POSITIVE side of the y axis. If the
675         mirror option is set to TRUE the wing tip will be located at the NEGATIVE side of the
676         y axis.
677
678         The structure nodes are oriented so that its x axis points from the root of the surface
679         to the tip and its y axis is parallel to the section chord.
680     """

```

```

681     Args:
682         n_beam_elements (int): number of beam elements to be created in the surface
683         apply_torsion (bool): if True geometrical torsion will be applied to the surface, if
684                               False only the dihedral will be applied
685         torsion_center (float): position relative to the chord around which the section will
686                               be
687                               rotated, must be a number between 0 and 1
688         torsion_function (function): a function that receives a span position, between 0 and
689                               1,
690                               and returns a rotation angle in radians
691         mirror (bool): If false wing tip will be in the positive side of the y axis, if True
692                               wing tip will be in the negative side of the y axis.
693     """
694
695     n_nodes = n_beam_elements + 1
696
697     if torsion_function == "linear":
698         torsion_function = (
699             lambda span_position: span_position * self.tip_torsion_angle_rad
700         )
701
702     # Find the positions of the root and tip nodes in the planform
703     root_node_xyz = np.array(
704         [self.root_chord * self.root_section.shear_center, 0, 0]
705     )
706
707     tip_x_position = self.length * tan(self.leading_edge_sweep_angle_rad)
708
709     tip_node_xyz = np.array(
710         [
711             tip_x_position + self.tip_chord * self.tip_section.shear_center,
712             self.length,
713             0,
714         ]
715     )
716
717     if mirror:
718         tip_node_xyz[1] = -tip_node_xyz[1]
719
720     # Calculate rotation due to wing sweep
721     z_rotation = 0.5 * np.pi - np.arctan(
722         (tip_node_xyz - root_node_xyz)[0] / self.length
723     )
724
725     if mirror:
726         #z_rotation = np.pi - z_rotation
727         z_rotation = -z_rotation
728
729     root_quaternion = Quaternion(axis=[0, 0, 1], angle=z_rotation)
730     tip_quaternion = Quaternion(axis=[0, 0, 1], angle=z_rotation)
731
732     # Create nodes in the planform
733     root_node = Node(root_node_xyz, root_quaternion)
734     tip_node = Node(tip_node_xyz, tip_quaternion)
735
736     # Apply dihedral angle, rotate around wing root in the x axis
737     rotation_center = np.array([0, 0, 0])
738     x_rotation = self.dihedral_angle_rad
739
740     if mirror:
741         x_rotation = -x_rotation
742
743     rotation_quaternion = Quaternion(axis=[1, 0, 0], angle=x_rotation)
744
745     root_node = root_node.rotate(
746         rotation_quaternion=rotation_quaternion, rotation_center=rotation_center
747     )

```

```

746     tip_node = tip_node.rotate(
747         rotation_quaternion=rotation_quaternion, rotation_center=rotation_center
748     )
749
750     # Interpolate root and tip nodes to create grid
751     structure_nodes_props = f.interpolate_nodes(root_node, tip_node, n_nodes)
752
753     structure_nodes = []
754
755     for node_prop in structure_nodes_props:
756         structure_nodes.append(Node(node_prop[0], node_prop[1]))
757
758     # Apply Node Rotation
759     if apply_torsion:
760         structure_nodes_props = f.apply_torsion_to_nodes(
761             structure_nodes, torsion_center, torsion_function, self
762         )
763
764     structure_nodes = []
765
766     for node_prop in structure_nodes_props:
767         structure_nodes.append(Node(node_prop[0], node_prop[1]))
768
769     return structure_nodes
770
771
772 #
=====

773
774
775 class MacroSurface(object):
776     """Defines a macrosurface composed of surfaces, is used to create wings, horizontal and
777     vertical
778     stabilizers.
779
780     Args:
781         position (float): position of the leading edge of the macrosurface root [m]
782         incidence (float): angle of incidence of the macrosurface [ $\hat{z}$ ]
783         surface_list (list[surface]): list of surface objects tha compose the macrosurface,
784         ordered
785             from left to right
786         symmetry_plane (string): plane of symmetry of the surface, for example XZ for a wing,
787         none if
788             the surface is not symmetrical
789         torsion_center (float): position in the chord around wich geometric torsion is applied
790
791     Attributes:
792         position (np.array(3)): x, y, z position of the leading edge of the macrosurface root [m]
793
794         incidence_degrees (float): angle of incidence of the macrosurface [ $\hat{z}$ ]
795         incidence_rad (float): angle of incidence of the macrosurface [rad]
796         surface_list (list[surface]): list of surface objects tha compose the macrosurface,
797         ordered
798             from left to right
799         symmetry_plane (string): plane of symmetry of the surface, for example XZ for a wing,
800         none if
801             the surface is not symmetrical
802         torsion_center (float): position in the chord around wich geometric torsion is applied
803         control_surfaces (list[string]): list with the identifiers of all control surfaces
804         present
805             in the macro surface.
806
807     """
808
809     def __init__(
810         self,
811         position,
812

```

```

804     incidence,
805     surface_list,
806     symmetry_plane=None,
807     torsion_center=0.25,
808     ):
809
810     self.position = position
811     self.incidence_degrees = float(incidence)
812     self.incidence_rad = np.radians(incidence)
813     self.surface_list = surface_list
814     self.symmetry_plane = symmetry_plane
815     self.torsion_center = float(torsion_center)
816
817     control_surfaces = []
818
819     for surface in surface_list:
820         if surface.control_surface_hinge_position is not None:
821             control_surfaces.append(surface.identifier)
822
823     self.control_surfaces = control_surfaces
824
825     self.ref_area = 0
826     self.true_area = 0
827
828     for surface in surface_list:
829         self.ref_area += surface.ref_area
830         self.true_area += surface.true_area
831
832     self.mean_aero_chord = surface_list[0].root_chord
833
834     macrosurface_aero_grid, macrosurface_nodes_list = self.create_grids(
835         n_chord_panels=3,
836         n_span_panels_list=[1 for surface in surface_list],
837         n_beam_elements_list=[1 for surface in surface_list],
838         chord_discretization="linear",
839         span_discretization_list=["linear" for surface in surface_list],
840         torsion_function_list=["linear" for surface in surface_list],
841     )
842
843     root_nodes = []
844     tip_nodes = []
845
846     for surface_nodes in macrosurface_nodes_list:
847         root_nodes.append(surface_nodes[0])
848         tip_nodes.append(surface_nodes[-1])
849
850     self.root_nodes = root_nodes
851     self.tip_nodes = tip_nodes
852
853     #
854     -----
855
856     def create_grids(
857         self,
858         n_chord_panels,
859         n_span_panels_list,
860         n_beam_elements_list,
861         chord_discretization,
862         span_discretization_list,
863         torsion_function_list,
864         control_surface_deflection_dict=dict(),
865     ):
866
867         macrosurface_aero_grid = []
868         macrosurface_nodes_list = []
869
870         if self.symmetry_plane == "XZ" or self.symmetry_plane == "xz":

```

```
869         middle_index = int(len(self.surface_list) / 2)
870
871     else:
872         middle_index = 0
873
874     right_side = self.surface_list[middle_index:]
875
876     translation_vector_list = [self.position]
877     incidence_angle_list = [self.incidence_rad]
878
879     for i, surface in enumerate(right_side):
880
881         leading_edge_x = surface.length * tan(surface.leading_edge_sweep_angle_rad)
882         leading_edge_y = surface.span
883         leading_edge_z = surface.span * tan(surface.dihedral_angle_rad)
884
885         position = translation_vector_list[i] + np.array(
886             [leading_edge_x, leading_edge_y, leading_edge_z]
887         )
888
889         incidence = incidence_angle_list[i] + surface.tip_torsion_angle_rad
890
891     if i < len(right_side) - 1:
892         translation_vector_list.append(position)
893         incidence_angle_list.append(incidence)
894
895     if self.symmetry_plane == "XZ" or self.symmetry_plane == "xz":
896
897         # Create a mirror image of the translation_vector_list
898         mirror_translation_vector_list = np.flip(translation_vector_list, axis=0)
899
900         for vector in mirror_translation_vector_list:
901             vector[1] = -abs(vector[1] - self.position[1]) + self.position[1]
902
903         translation_vector_list = np.concatenate(
904             [mirror_translation_vector_list, translation_vector_list]
905         )
906         incidence_angle_list = np.concatenate(
907             [np.flip(incidence_angle_list), incidence_angle_list]
908         )
909
910     for i, surface in enumerate(self.surface_list):
911
912         if i < middle_index:
913             mirror = True
914         else:
915             mirror = False
916
917         if surface.identifier in control_surface_deflection_dict:
918             control_surface_deflection = control_surface_deflection_dict[
919                 surface.identifier
920             ]
921         else:
922             control_surface_deflection = 0
923
924         # Generate planar mesh
925         aero_grid_dict = surface.generate_aero_grid(
926             n_span_panels=n_span_panels_list[i],
927             n_chord_panels=n_chord_panels,
928             apply_torsion=False,
929             mirror=mirror,
930             control_surface_deflection=control_surface_deflection,
931             chord_discretization=chord_discretization,
932             span_discretization=span_discretization_list[i],
933         )
934
935
```

```

936     # Generate planar nodes
937     surface_nodes_list = surface.generate_structure_nodes(
938         n_beam_elements=n_beam_elements_list[i],
939         apply_torsion=False,
940         mirror=mirror,
941     )
942
943     # Create torsion function
944     torsion_function = (
945         lambda span_position: incidence_angle_list[i]
946         + span_position * surface.tip_torsion_angle_rad
947     )
948
949     # Apply torsion to aero grid and nodes
950     aero_grid_dict = f.apply_torsion_to_grid(
951         aero_grid_dict, self.torsion_center, torsion_function, surface
952     )
953
954     surface_nodes_list_prop = f.apply_torsion_to_nodes(
955         surface_nodes_list, self.torsion_center, torsion_function, surface
956     )
957
958     surface_nodes_list = []
959
960     for node_prop in surface_nodes_list_prop:
961         surface_nodes_list.append(Node(node_prop[0], node_prop[1]))
962
963     # Translate grid
964     xx, yy, zz = f.translate_grid(
965         aero_grid_dict["xx"],
966         aero_grid_dict["yy"],
967         aero_grid_dict["zz"],
968         translation_vector_list[i],
969         start_point=np.array([0, 0, 0]),
970     )
971     aero_grid_dict = {"xx": xx, "yy": yy, "zz": zz}
972
973     for j, node in enumerate(surface_nodes_list):
974         surface_nodes_list[j] = node.translate(translation_vector_list[i])
975
976     macrosurface_aero_grid.append(aero_grid_dict)
977     macrosurface_nodes_list.append(surface_nodes_list)
978
979 return macrosurface_aero_grid, macrosurface_nodes_list
980
981 #
-----
```

A.6 *loads*

A.6.1 *__init__.py*

```
1 from . import functions
```

A.6.2 *functions.py*

```
1 # =====
2 # IMPORTS
3 import numpy as np
4 import scipy as sc
5
6 from numpy import sin, cos, tan, pi
7 from numba import jit
8 from pyquaternion import Quaternion
9
10 from .. import aerodynamics as aero
11 from .. import mathematics as m
12 from .. import geometry as geo
13
14 #
15 # Functions
16
17
18 def calc_aero_loads_at_point(point, aircraft_force_grid, aircraft_panel_grid):
19
20     macrosurfaces_aero_loads = []
21
22     for macrosurface_force_grid, macrosurface_panel_grid in zip(
23         aircraft_force_grid, aircraft_panel_grid
24     ):
25
26         # Transform into a vector
27         macrosurface_force_vector = np.copy(
28             np.reshape(macrosurface_force_grid, np.size(macrosurface_force_grid))
29         )
30         macrosurface_panel_vector = np.copy(
31             np.reshape(macrosurface_panel_grid, np.size(macrosurface_panel_grid))
32         )
33
34         # Compute forces
35         aero_forces = np.zeros(3)
36         aero_moments = np.zeros(3)
37
38         for force, panel in zip(macrosurface_force_vector, macrosurface_panel_vector):
39             aero_forces += force
40
41             # Moment lever calculation
42             force_application_point = panel.aero_center
43             lever_arm = point - force_application_point
44
45             aero_moments += m.cross(lever_arm, force)
46
47         macrosurfaces_aero_loads.append([np.copy(aero_forces), np.copy(aero_moments)])
48
49     total_aero_force = np.zeros(3)
50     total_aero_moment = np.zeros(3)
51
52     for component in macrosurfaces_aero_loads:
53
54         total_aero_force += component[0]
55         total_aero_moment += component[1]
56
57     return total_aero_force, total_aero_moment, macrosurfaces_aero_loads
58
59
60 # -----
```

```
61
62
63 def calc_engine_loads_at_point(aircraft, point, throttle_list, parameters_list):
64
65     engine_loads = []
66     engine_force = np.zeros(3)
67     engine_moment = np.zeros(3)
68
69     for i, engine in enumerate(aircraft.engines):
70
71         # Thrust calculation
72         thrust = engine.thrust(throttle_list[i], parameters_list[i])
73
74         # Moment calculation
75         lever_arm = point - engine.position
76
77         moment = m.cross(lever_arm, thrust)
78
79         engine_loads.append([np.copy(thrust), np.copy(moment)])
80
81         engine_force += thrust
82         engine_moment += moment
83
84     return engine_force, engine_moment, engine_loads
85
86
87 #
-----
```



```
88
89
90 def calc_lift_drag(
91     aircraft,
92     point,
93     speed,
94     altitude,
95     attitude_vector,
96     aircraft_force_grid,
97     aircraft_panel_grid,
98 ):
99     """Calculates the lift and drag of the aircraft
100    """
101
102     # Calculate wind coord system in relation to aircraft coordinate system
103
104     alpha = np.radians(attitude_vector[0])
105     beta = np.radians(attitude_vector[1])
106     gamma = np.radians(attitude_vector[2])
107
108     wind_coord_sys = geo.objects.Node(xyz=np.array([0, 0, 0]), quaternion=Quaternion())
109
110     # Apply YAW rotation
111     wind_coord_sys = wind_coord_sys.rotate(
112         rotation_quaternion=Quaternion(axis=wind_coord_sys.z_axis, angle=-beta)
113     )
114
115     # Apply PITCH rotation
116     wind_coord_sys = wind_coord_sys.rotate(
117         rotation_quaternion=Quaternion(axis=wind_coord_sys.y_axis, angle=-alpha)
118     )
119
120     # Apply ROLL rotation
121     wind_coord_sys = wind_coord_sys.rotate(
122         rotation_quaternion=Quaternion(axis=wind_coord_sys.x_axis, angle=-gamma)
123     )
124
125     # Calculate aerodynamic forces in the aircraft coordinates system
```

```

126     total_cg_aero_force, total_cg_aero_moment, component_cg_aero_loads =
127         calc_aero_loads_at_point(
128             point, aircraft_force_grid, aircraft_panel_grid
129         )
130
131     # Transform aerodynamic forces from aircraft coordinate system to wind coordinate system
132     aero_forces = geo.functions.change_coord_sys(
133         total_cg_aero_force,
134         wind_coord_sys.x_axis,
135         wind_coord_sys.y_axis,
136         wind_coord_sys.z_axis,
137     )
138
139     lift = aero_forces[2]
140     drag = aero_forces[0]
141     sideforce = aero_forces[1]
142
143     density, pressure, temperature = aero.functions.ISA(altitude)
144     Cl = lift / (0.5 * density * (speed ** 2) * aircraft.ref_area)
145     Cd = drag / (0.5 * density * (speed ** 2) * aircraft.ref_area)
146
147     aero_moments = geo.functions.change_coord_sys(
148         total_cg_aero_moment,
149         wind_coord_sys.x_axis,
150         wind_coord_sys.y_axis,
151         wind_coord_sys.z_axis,
152     )
153
154     roll_moment = aero_moments[0]
155     pitch_moment = aero_moments[1]
156     yaw_moment = aero_moments[2]
157
158     Cm = pitch_moment / (
159         0.5 * density * (speed ** 2) * aircraft.ref_area * aircraft.mean_aero_chord
160     )
161
162     # Prepare Output
163     forces = {"lift": lift, "drag": drag, "sideforce": sideforce}
164     moments = {
165         "roll_moment": roll_moment,
166         "pitch_moment": pitch_moment,
167         "yaw_moment": yaw_moment,
168     }
169     coefficients = {"Cl": Cl, "Cd": Cd, "Cm": Cm}
170
171     return forces, moments, coefficients
172
173 #
-----
```

```

174
175
176 def calc_load_distribution(
177     aircraft_force_grid, aircraft_gamma_grid, aircraft_panel_grid, attitude_vector, altitude,
178     speed
179 ):
180
181     density, pressure, temperature = aero.functions.ISA(altitude)
182
183     components_loads = []
184
185     for component_force_grid, component_gamma_grid, component_panel_grid in zip(
186         aircraft_force_grid, aircraft_gamma_grid, aircraft_panel_grid
187     ):
188
189         # Calculate wind coord system in relation to aircraft coodinate system

```

```

189
190     alpha = np.radians(attitude_vector[0])
191     beta = np.radians(attitude_vector[1])
192     gamma = np.radians(attitude_vector[2])
193
194     wind_coord_sys = geo.objects.Node(
195         xyz=np.array([0, 0, 0]), quaternion=Quaternion()
196     )
197
198     # Apply YAW rotation
199     wind_coord_sys = wind_coord_sys.rotate(
200         rotation_quaternion=Quaternion(axis=wind_coord_sys.z_axis, angle=-beta)
201     )
202
203     # Apply PITCH rotation
204     wind_coord_sys = wind_coord_sys.rotate(
205         rotation_quaternion=Quaternion(axis=wind_coord_sys.y_axis, angle=-alpha)
206     )
207
208     # Apply ROLL rotation
209     wind_coord_sys = wind_coord_sys.rotate(
210         rotation_quaternion=Quaternion(axis=wind_coord_sys.x_axis, angle=-gamma)
211     )
212
213     n_chord_panels, n_span_panels = np.shape(component_force_grid)
214
215     y_values = np.zeros(n_span_panels)
216
217     x_force = np.zeros(n_span_panels)
218     y_force = np.zeros(n_span_panels)
219     z_force = np.zeros(n_span_panels)
220
221     lift = np.zeros(n_span_panels)
222     drag = np.zeros(n_span_panels)
223     side = np.zeros(n_span_panels)
224     Cl = np.zeros(n_span_panels)
225     Cd = np.zeros(n_span_panels)
226
227     for i in range(n_span_panels):
228
229         force_section = component_force_grid[:, i]
230         gamma_section = component_gamma_grid[:, i]
231         panels_section = np.array(component_panel_grid)[:, i]
232
233         section_total_force = force_section.sum()
234         section_total_gamma = gamma_section.sum()
235
236         # Calculate section area and chord
237         section_area = 0
238         section_chord = 0
239
240         for panel in panels_section:
241             section_area += panel.area
242             section_chord += m.norm((panel.l_chord + panel.r_chord) / 2))
243
244         section_span = panel.span
245
246         # Calculate section chord
247
248         x_force[i] = section_total_force[0]
249         y_force[i] = section_total_force[1]
250         z_force[i] = section_total_force[2]
251
252         # Transform aerodynamic forces from aircraft coordinate system to wind coordinate
253         # system
254         section_aero_forces = geo.functions.change_coord_sys(
255             section_total_force,

```

```

255             wind_coord_sys.x_axis,
256             wind_coord_sys.y_axis,
257             wind_coord_sys.z_axis,
258         )
259
260     lift[i] = section_aero_forces[2] / section_span
261     drag[i] = section_aero_forces[0] / section_span
262     side[i] = section_aero_forces[1] / section_span
263
264     y_values[i] = panels_section[0].aero_center[1]
265
266     Cl[i] = section_aero_forces[2] / (0.5 * density * (speed ** 2) * section_area)
267     Cd[i] = drag[i] / (0.5 * density * (speed ** 2) * section_area)
268
269     loads = {
270         "x_force": x_force,
271         "y_force": y_force,
272         "z_force": z_force,
273         "lift": lift,
274         "drag": drag,
275         "side": side,
276         "Cl": Cl,
277         "Cd": Cd,
278         "y_values": y_values,
279     }
280
281     components_loads.append(loads)
282
283     return components_loads
284
285
286 #
-----
```

```

287
288
289 def calculate_surface_panels_loads(surface_panel_grid, surface_force_grid):
290
291     n_chord_panels = np.shape(surface_panel_grid)[0]
292     n_spam_panels = np.shape(surface_panel_grid)[1]
293
294     delta_p_grid = np.zeros(np.shape(surface_panel_grid))
295     force_magnitude_grid = np.zeros(np.shape(surface_panel_grid))
296     force_x_grid = np.zeros(np.shape(surface_panel_grid))
297     force_y_grid = np.zeros(np.shape(surface_panel_grid))
298     force_z_grid = np.zeros(np.shape(surface_panel_grid))
299
300     for i in range(n_chord_panels):
301         for j in range(n_spam_panels):
302             force_x_grid[i][j] = surface_force_grid[i][j][0]
303             force_y_grid[i][j] = surface_force_grid[i][j][1]
304             force_z_grid[i][j] = surface_force_grid[i][j][2]
305
306             force_magnitude_grid[i][j] = m.norm(surface_force_grid[i][j])
307             delta_p_grid[i][j] =
308                 force_magnitude_grid[i][j] / surface_panel_grid[i][j].area
309
310
311     return {
312         "delta_p_grid": delta_p_grid,
313         "force_magnitude_grid": force_magnitude_grid,
314         "force_x_grid": force_x_grid,
315         "force_y_grid": force_y_grid,
316         "force_z_grid": force_z_grid,
317     }
318
319
```

```

320 #
321 -----
322
323 def calculate_aircraft_panel_loads(aircraft_panel_grids, aircraft_force_grids):
324
325     aircraft_panel_loads = []
326
327     for component_panel_grid, component_force_grid in zip(
328         aircraft_panel_grids, aircraft_force_grids
329     ):
330
331         component_panel_loads = calculate_surface_panels_loads(
332             component_panel_grid, component_force_grid
333         )
334
335     aircraft_panel_loads.append(component_panel_loads)
336
337     return aircraft_panel_loads

```

A.7 *structures*

A.7.1 *__init__.py*

```

1 from . import objects
2 from . import functions
3 from . import fem

```

A.7.2 *fem.py*

```

1 import numpy as np
2 import scipy as sc
3
4 from numpy import sin, cos, tan, pi
5 from pyquaternion import Quaternion
6
7 from .. import geometry as geo
8 from . import functions as f
9 from . import objects as o
10 from .. import mathematics as m
11
12
13 def number_nodes(components_list, components_nodes_list, connections_list):
14     """This function modifies the number attribute in the nodes of the componentes nodes lists
15     """
16
17     # Generate the connection matrix
18     connection_matrix = []
19     for connection in connections_list:
20         connection_matrix.append(connection.descriptor)
21
22     node_counter = 0
23
24     for i, component in enumerate(components_list):
25
26         for j, node in enumerate(components_nodes_list[i]):
27

```

```

28         # If the node is the root of the component check if it is connected to a node that
29         has
30         # already been numbered
31         if j == 0:
32             # Check if node is part of a connection
33             node_identifier = component.identifier + "-ROOT"
34             connections = check_connections(
35                 connections_list, components_list, node_identifier
36             )
37
38         # Run through the connections and selects connected nodes
39         connected_nodes = []
40         for connection in connections:
41
42             connected_comp_index = components_list.index(connection[0])
43             connected_comp_nodes = components_nodes_list[connected_comp_index]
44
45             if connection[1] == "ROOT":
46                 connected_node = connected_comp_nodes[0]
47
48             elif connection[1] == "TIP":
49                 last_index = len(connected_comp_nodes) - 1
50                 connected_node = connected_comp_nodes[last_index]
51
52             connected_nodes.append(connected_node)
53
54         # Run through the connected nodes and checks if one of the is numbered
55         node_number = None
56         for connected_node in connected_nodes:
57             if connected_node.number is not None:
58                 node_number = connected_node.number
59
60             if node_number is not None:
61                 node.number = node_number
62             else:
63                 node.number = node_counter
64                 node_counter += 1
65
66         # If the node is the tip of the component check if it is connected to a node tha has
67         # already been numbered, same as the root
68         elif j == len(components_nodes_list[i]) - 1:
69             # Check if node is part of a connection
70             node_identifier = component.identifier + "-TIP"
71             connections = check_connections(
72                 connections_list, components_list, node_identifier
73             )
74
75         # Run through the connections and selects connected nodes
76         connected_nodes = []
77         for connection in connections:
78
79             connected_comp_index = components_list.index(connection[0])
80             connected_comp_nodes = components_nodes_list[connected_comp_index]
81
82             if connection[1] == "ROOT":
83                 connected_node = connected_comp_nodes[0]
84
85             elif connection[1] == "TIP":
86                 last_index = len(connected_comp_nodes) - 1
87                 connected_node = connected_comp_nodes[last_index]
88
89             connected_nodes.append(connected_node)
90
91         # Run through the connected nodes and checks if one of the is numbered
92         node_number = None
93         for connected_node in connected_nodes:
94             if connected_node.number is not None:

```

```

94             node_number = connected_node.number
95
96         if node_number is not None:
97             node.number = node_number
98         else:
99             node.number = node_counter
100            node_counter += 1
101
102     # If the node is not at the ROOT or the TIP give it the next avaivable number:
103     else:
104         node.number = node_counter
105         node_counter += 1
106
107
108 #
=====

109
110
111 def check_connections(connections_list, components_list, node_identifier):
112     """This function returns all the connections to a node
113     """
114     connections = []
115
116     # Create a list with all the connection identifiers
117     connection_matrix = []
118     for connection in connections_list:
119         connection_matrix.append(connection.descriptor)
120
121     # Run through the connections
122     for i, connection in enumerate(connection_matrix):
123
124         # Finds a connection that contains the node identifier
125         if node_identifier in connection:
126
127             # Connections contains two components, index tells if it is the first or second
128             # component
129             index = connection.index(node_identifier)
130             connected_component_index = abs(index - 1)
131
132             if connected_component_index == 0:
133                 connected_component = connections_list[i].component1
134                 connected_node = connections_list[i].component1_node
135
136             elif connected_component_index == 1:
137                 connected_component = connections_list[i].component2
138                 connected_node = connections_list[i].component2_node
139
140             connections.append([connected_component, connected_node])
141
142     return connections
143
144
145 #
=====

146
147
148 def create_macrosurface_connections(macrosurface):
149
150     if macrosurface.symmetry_plane == "XZ" or macrosurface.symmetry_plane == "xz":
151
152         middle_index = int(len(macrosurface.surface_list) / 2)
153
154     else:
155         middle_index = 0
156

```

```

157     connections_list = []
158
159     if middle_index == 0:
160
161         for i in range(len(macrosurface.surface_list) - 1):
162
163             connection = o.Connection(
164                 component1=macrosurface.surface_list[i],
165                 component1_node="TIP",
166                 component2=macrosurface.surface_list[i + 1],
167                 component2_node="ROOT",
168             )
169
170             connections_list.append(connection)
171
172     else:
173         for i in range(len(macrosurface.surface_list) - 1):
174
175             if i >= middle_index:
176                 connection = o.Connection(
177                     component1=macrosurface.surface_list[i],
178                     component1_node="TIP",
179                     component2=macrosurface.surface_list[i + 1],
180                     component2_node="ROOT",
181                 )
182
183             elif i == middle_index - 1:
184                 connection = o.Connection(
185                     component1=macrosurface.surface_list[i],
186                     component1_node="ROOT",
187                     component2=macrosurface.surface_list[i + 1],
188                     component2_node="ROOT",
189                 )
190
191     else:
192         connection = o.Connection(
193             component1=macrosurface.surface_list[i],
194             component1_node="ROOT",
195             component2=macrosurface.surface_list[i + 1],
196             component2_node="TIP",
197         )
198
199     connections_list.append(connection)
200
201     return connections_list
202
203
204 #
=====

205
206
207 def generate_macrosurface_fem_elements(
208     macrosurface, macrosurface_nodes_list, prop_choice="MIDDLE"
209 ):
210     """Generates the beam finite elements of a macrosurface object.
211
212     Args
213
214     Returns
215     """
216
217     macrosurface_elements = []
218
219     for surface, surface_nodes_list in zip(
220         macrosurface.surface_list, macrosurface_nodes_list
221     ):

```

```

222
223     n_nodes = len(surface_nodes_list)
224
225     # Calculates the equivalent square section for the root and tip properties
226     root_A = surface.root_section.area
227     root_Iyy = surface.root_section.Iyy
228     root_Izz = surface.root_section.Izz
229     root_J = surface.root_section.J
230     root_E = surface.root_section.material.elasticity_modulus
231     root_G = surface.root_section.material.rigidity_modulus
232
233     root_A_sqside = np.sqrt(root_A)
234     root_Iyy_sqside = (12 * root_Iyy) ** (1 / 4)
235     root_Izz_sqside = (12 * root_Izz) ** (1 / 4)
236     root_J_sqside = (6 * root_J) ** (1 / 4)
237
238     tip_A = surface.tip_section.area
239     tip_Iyy = surface.tip_section.Iyy
240     tip_Izz = surface.tip_section.Izz
241     tip_J = surface.tip_section.J
242     tip_E = surface.root_section.material.elasticity_modulus
243     tip_G = surface.root_section.material.rigidity_modulus
244
245     tip_A_sqside = np.sqrt(tip_A)
246     tip_Iyy_sqside = (12 * tip_Iyy) ** (1 / 4)
247     tip_Izz_sqside = (12 * tip_Izz) ** (1 / 4)
248     tip_J_sqside = (6 * tip_J) ** (1 / 4)
249
250     # Linear interpolation of the equivalent square section for the node properties
251     As = np.zeros(n_nodes)
252     Iyys = np.zeros(n_nodes)
253     Izzs = np.zeros(n_nodes)
254     Js = np.zeros(n_nodes)
255     Es = np.zeros(n_nodes)
256     Gs = np.zeros(n_nodes)
257
258     root_y = surface_nodes_list[0].y
259     tip_y = surface_nodes_list[-1].y
260
261     interpolation = lambda root_prop, tip_prop, node_y: (
262         np.interp([node_y], [root_y, tip_y], [root_prop, tip_prop]))
263     )
264
265     # Calculation of the interpolated properties for each of the nodes
266     for i, node in enumerate(surface_nodes_list):
267
268         node_y = node.y
269         A_sqside = interpolation(root_A_sqside, tip_A_sqside, node_y)
270         Iyy_sqside = interpolation(root_Iyy_sqside, tip_Iyy_sqside, node_y)
271         Izz_sqside = interpolation(root_Izz_sqside, tip_Izz_sqside, node_y)
272         J_sqside = interpolation(root_J_sqside, tip_J_sqside, node_y)
273
274         node_E = interpolation(root_E, tip_E, node_y)
275         node_G = interpolation(root_G, tip_G, node_y)
276         node_A = A_sqside ** 2
277         node_Iyy = (Iyy_sqside ** 4) / 12
278         node_Izz = (Izz_sqside ** 4) / 12
279         node_J = (J_sqside ** 4) / 6
280
281         As[i] = node_A
282         Iyys[i] = node_Iyy
283         Izzs[i] = node_Izz
284         Js[i] = node_J
285         Es[i] = node_E
286         Gs[i] = node_G
287
288     # Create FEM elements

```

```

289     surface_elements = []
290
291     for i in range(n_nodes - 1):
292
293         if prop_choice == "ROOT":
294             A = As[i]
295             Iyy = Iyys[i]
296             Izz = Izzs[i]
297             J = Js[i]
298             E = Es[i]
299             G = Gs[i]
300
301         elif prop_choice == "TIP":
302             A = As[i + 1]
303             Iyy = Iyys[i + 1]
304             Izz = Izzs[i + 1]
305             J = Js[i + 1]
306             E = Es[i + 1]
307             G = Gs[i + 1]
308
309         elif prop_choice == "MIDDLE":
310             A = (As[i] + As[i + 1]) / 2
311             Iyy = (Iyys[i] + Iyys[i + 1]) / 2
312             Izz = (Izzs[i] + Izzs[i + 1]) / 2
313             J = (Js[i] + Js[i + 1]) / 2
314             E = (Es[i] + Es[i + 1]) / 2
315             G = (Gs[i] + Gs[i + 1]) / 2
316
317         else:
318             print(
319                 "ERROR: Invalid prop_choice, options are 'ROOT', 'TIP' and 'MIDDLE'"
320             )
321
322         element = o.EulerBeamElement(
323             node_A=surface_nodes_list[i],
324             node_B=surface_nodes_list[i + 1],
325             A=A,
326             Iyy=Iyy,
327             Izz=Izz,
328             J=J,
329             E=E,
330             G=G,
331             prop_choice=prop_choice,
332         )
333
334         surface_elements.append(element)
335
336     macrosurface_elements.append(surface_elements)
337
338     return macrosurface_elements
339
340
341
342 #
=====

343
344
345 def generate_beam_fem_elements(beam, beam_nodes_list, prop_choice="MIDDLE"):
346
347     n_nodes = len(beam_nodes_list)
348
349     beam_elements = []
350
351     for i in range(n_nodes - 1):
352
353         A = beam.ElementProperty.A

```

```
354     Iyy = beam.ElementProperty.Iyy
355     Izz = beam.ElementProperty.Izz
356     J = beam.ElementProperty.J
357     E = beam.ElementProperty.E
358     G = beam.ElementProperty.G
359
360     element = o.EulerBeamElement(
361         node_A=beam_nodes_list[i],
362         node_B=beam_nodes_list[i + 1],
363         A=A,
364         Iyy=Iyy,
365         Izz=Izz,
366         J=J,
367         E=E,
368         G=G,
369         prop_choice=prop_choice,
370     )
371
372     beam_elements.append(element)
373
374     return beam_elements
375
376
377 #
=====

378
379
380 def generate_aircraft_fem_elements(
381     aircraft,
382     aircraft_grids,
383     prop_choice="ROOT",
384 ):
385
386     aircraft_macrosurfaces_struct_grids = aircraft_grids["macrosurfaces_struct_grids"]
387     aircraft_beams_struct_grids = aircraft_grids["beams_struct_grids"]
388
389     aircraft_macrosurfaces_fem_elements = []
390
391     for macrosurface, macrosurface_struct_grid in zip(
392         aircraft.macrosurfaces, aircraft_macrosurfaces_struct_grids
393     ):
394
395         macrosurface_fem_elements = generate_macrosurface_fem_elements(
396             macrosurface, macrosurface_struct_grid, prop_choice
397         )
398
399         aircraft_macrosurfaces_fem_elements.append(macrosurface_fem_elements)
400
401     aircraft_beams_fem_elements = []
402
403     if aircraft.beams:
404         for beam, beam_struct_grid in zip(aircraft.beams, aircraft_beams_struct_grids):
405
406             beam_elements = generate_beam_fem_elements(
407                 beam, beam_struct_grid, prop_choice
408             )
409             aircraft_beams_fem_elements.append(beam_elements)
410
411     if aircraft.beams:
412         aircraft_fem_elements = {
413             "macrosurfaces_fem_elements":aircraft_macrosurfaces_fem_elements,
414             "beams_fem_elements":aircraft_beams_fem_elements,
415         }
416
417     else:
418         aircraft_fem_elements = {
```

```
419         "macrosurfaces_fem_elements":aircraft_macrosurfaces_fem_elements,
420         "beams_fem_elements":None,
421     }
422
423     return aircraft_fem_elements
424
425 #
=====
426
427
428 def structural_solver(struct_grid, struct_elements, struct_loads, struct_constraints):
429
430     node_vector = geo.functions.create_structure_node_vector(struct_grid)
431
432     elements_vector = []
433     # Add all elements to a vector
434     for component_elements in struct_elements:
435         elements_vector += component_elements
436
437     K_global, F_global = create_global_FEM_matrices(
438         node_vector, elements_vector, struct_loads
439     )
440
441     X_global = FEM_solver(K_global, F_global, struct_constraints)
442
443     # Find support reactions
444     force_vector = K_global @ X_global
445
446     # Deformed grid
447     deformations = np.reshape(X_global, (len(node_vector), 6))
448     internal_loads = np.reshape(force_vector, (len(node_vector), 6))
449
450     deformed_grid = []
451     for i, node in enumerate(node_vector):
452         deformed_grid.append(
453             [
454                 node.x + deformations[i][0],
455                 node.y + deformations[i][1],
456                 node.z + deformations[i][2],
457                 deformations[i][3],
458                 deformations[i][4],
459                 deformations[i][5],
460             ]
461         )
462
463     deformed_struct_grid = []
464     struct_internal_loads = []
465     struct_strains = []
466
467     for component_grid in struct_grid:
468
469         deformed_component_grid = []
470         component_internal_loads = []
471         component_strains = []
472
473         for node in component_grid:
474
475             # Create deformed grid
476             x_delta = deformations[node.number][0]
477             y_delta = deformations[node.number][1]
478             z_delta = deformations[node.number][2]
479             rx_delta = deformations[node.number][3]
480             ry_delta = deformations[node.number][4]
481             rz_delta = deformations[node.number][5]
482
483             x_axis = np.array([1.0, 0.0, 0.0])
```

```

484         y_axis = np.array([0.0, 1.0, 0.0])
485         z_axis = np.array([0.0, 0.0, 1.0])
486
487         # Apply translation
488         translation_vector = np.array([x_delta, y_delta, z_delta])
489         deformed_node = node.translate(translation_vector)
490
491         # Apply rotation
492         rot_quaternion = Quaternion(axis=x_axis, angle=rx_delta)
493         deformed_node = node.rotate(rot_quaternion)
494
495         rot_quaternion = Quaternion(axis=y_axis, angle=ry_delta)
496         deformed_node = node.rotate(rot_quaternion)
497
498         rot_quaternion = Quaternion(axis=z_axis, angle=rz_delta)
499         deformed_node = node.rotate(rot_quaternion)
500
501     deformed_component_grid.append(deformed_node)
502
503     # Create internal forces vector
504     node_internal_load = internal_loads[node.number]
505     component_internal_loads.append(node_internal_load)
506
507     # Create struct strains vector
508     node_strain = deformations[node.number]
509     component_strains.append(node_strain)
510
511     deformed_struct_grid.append(deformed_component_grid)
512     struct_internal_loads.append(component_internal_loads)
513     struct_strains.append(component_strains)
514
515     return deformations, internal_loads
516     #return deformed_struct_grid, struct_internal_loads, struct_strains, node_vector,
517     #deformations
518 #
519 =====
520
521 def create_global_FEM_matrices(nodes, fem_elements, loads):
522
523     n_nodes = len(nodes)
524
525     # Generate global stiffness matrix
526     K_global = np.zeros((n_nodes * 6, n_nodes * 6))
527     F_global = np.zeros((n_nodes * 6, 1))
528
529     for fem_element in fem_elements:
530         K_element = fem_element.calc_K_global()
531         correlation_vector = fem_element.correlation_vector
532
533         for i in range(len(correlation_vector)):
534             for j in range(len(correlation_vector)):
535                 K_global[correlation_vector[i]][correlation_vector[j]] += K_element[i][
536                     j]
537
538     # Generate Force Matrix
539     for load in loads:
540         node_index = load.application_node.number * 6
541         correlation_vector = [
542             node_index,
543             node_index + 1,
544             node_index + 2,
545             node_index + 3,
546             node_index + 4,
547

```

```

548         node_index + 5,
549     ]
550
551     for i in range(len(correlation_vector)):
552         F_global[correlation_vector[i]] += load.load[i]
553
554     return K_global, F_global
555
556
557 #
=====

558
559
560 def FEM_solver(K_global, F_global, constraints):
561
562     n_dof = len(F_global)
563     X_global = np.zeros((n_dof, 1))
564
565     # Find constrained degrees of freedom
566     constrained_dof = [False for i in range(n_dof)]
567
568     for constraint in constraints:
569         node_index = constraint.application_node.number * 6
570         correlation_vector = [
571             node_index,
572             node_index + 1,
573             node_index + 2,
574             node_index + 3,
575             node_index + 4,
576             node_index + 5,
577         ]
578
579         for i in range(len(correlation_vector)):
580             if constraint.dof_constraints[i] is not None:
581                 constrained_dof[correlation_vector[i]] = True
582                 X_global[correlation_vector[i]] += constraint.dof_constraints[i]
583
584     # Created reduced stiffness and force matrices
585     red_K_global = np.copy(K_global)
586     red_F_global = np.copy(F_global)
587
588     dof_to_delete = []
589     for i, dof in enumerate(constrained_dof):
590         if dof:
591             dof_to_delete.append(i)
592
593     red_F_global = np.delete(red_F_global, dof_to_delete, 0)
594     red_K_global = np.delete(red_K_global, dof_to_delete, 0)
595     red_K_global = np.delete(red_K_global, dof_to_delete, 1)
596
597     # Solve linear System
598     red_X_global = np.linalg.solve(red_K_global, red_F_global)
599
600     # Copy results do deformation vector
601     counter = 0
602     for i, dof in enumerate(constrained_dof):
603         if not dof:
604             X_global[i] = red_X_global[counter]
605             counter += 1
606
607     return X_global

```

A.7.3 functions.py

```

1 """
2 """
3
4 import numpy as np
5 import scipy as sc
6
7 from numpy import sin, cos, tan, pi
8 from pyquaternion import Quaternion
9
10 from . import functions as f
11 from .. import mathematics as m
12
13
14 #
-----
```

```

15
16
17 def euler_beam_stiff(E, A, L, G, J, Iyy, Izz):
18     """ Calculates the local stiffness matrix of a beam finite element.
19
20     Args:
21         A (float): Area of the section
22         J (float): Polar moment of inertia of the section
23         Iyy (float): Moment of inertia of the section in the Y axis
24         Izz (float): Moment of inertia of the section in the Z axis
25         E (float): Elastic modulus of the material
26         G (float): Shear modulus of the material
27
28     Returns:
29         K (np.array(dtype=float)): local stiffness matrix of a beam element
30     """
31
32     K = np.zeros((12, 12))
33
34     K[0][0] = E * A / L
35     K[0][6] = -E * A / L
36     K[6][0] = K[0][6]
37
38     K[1][1] = 12 * E * Izz / L ** 3
39     K[1][5] = 6 * E * Izz / L ** 2
40     K[5][1] = K[1][5]
41     K[1][7] = -12 * E * Izz / L ** 3
42     K[7][1] = K[1][7]
43     K[1][11] = 6 * E * Izz / L ** 2
44     K[11][1] = K[1][11]
45
46     K[2][2] = 12 * E * Iyy / L ** 3
47     K[2][4] = -6 * E * Iyy / L ** 2
48     K[4][2] = K[2][4]
49     K[2][8] = -12 * E * Iyy / L ** 3
50     K[8][2] = K[2][8]
51     K[2][10] = -6 * E * Iyy / L ** 2
52     K[10][2] = K[2][10]
53
54     K[3][3] = G * J / L
55     K[3][9] = -G * J / L
56     K[9][3] = K[3][9]
57
58     K[4][4] = 4 * E * Iyy / L
59     K[4][8] = 6 * E * Iyy / L ** 2
60     K[8][4] = K[4][8]
61     K[4][10] = 2 * E * Iyy / L
62     K[10][4] = K[4][10]
63
```

```

64     K[5][5] = 4 * E * Izz / L
65     K[5][7] = -6 * E * Izz / L ** 2
66     K[7][5] = K[5][7]
67     K[5][11] = 2 * E * Izz / L
68     K[11][5] = K[5][11]
69
70     K[6][6] = E * A / L
71
72     K[7][7] = 12 * E * Izz / L ** 3
73     K[7][11] = -6 * E * Izz / L ** 2
74     K[11][7] = K[7][11]
75
76     K[8][8] = 12 * E * Iyy / L ** 3
77     K[8][10] = 6 * E * Iyy / L ** 2
78     K[10][8] = K[8][10]
79
80     K[9][9] = G * J / L
81
82     K[10][10] = 4 * E * Iyy / L
83
84     K[11][11] = 4 * E * Izz / L
85
86     return K

```

A.7.4 objects.py

```

1 """
2 DOCSTRING
3 """
4
5 import numpy as np
6 import scipy as sc
7
8 from numpy import sin, cos, tan, pi
9 from pyquaternion import Quaternion
10
11 from . import functions as f
12 from .. import mathematics as m
13 from .. import geometry as geo
14
15 #
=====

16 # Objects
17
18
19 class Material:
20     """Defines a material and it's properties
21
22     Args:
23         name (string) = material's name
24         density (float) = density of the material [kg/m§]
25         elasticity_modulus (float) = elastic modulus of the material [Pa]
26         rigidity_modulus (float) = rigidity modulus of the material [Pa]
27         poisson_ratio (float) = poisson's ratio of the material
28         yield_tensile_stress (float) = yield stress of the material in tension [Pa]
29         ultimate_tensile_stress (float) = ultimate stress of the material in tension [Pa]
30         yield_shear_stress (float) = yield stress of the material in shear [Pa]
31         ultimate_shear_stress (float) = ultimate stress of the material in shear [Pa]
32
33     Attributes:
34         name (string) = material's name
35         density (float) = density of the material [kg/m§]
36         elasticity_modulus (float) = elastic modulus of the material [Pa]

```

```
37     rigidity_modulus (float) = rigidity modulus of the material [Pa]
38     poisson_ratio (float) = poisson's ratio of the material
39     yield_tensile_stress (float) = yield stress of the material in tension [Pa]
40     ultimate_tensile_stress (float) = ultimate stress of the material in tension [Pa]
41     yield_shear_stress (float) = yield stress of the material in shear [Pa]
42     ultimate_shear_stress (float) = ultimate stress of the material in shear [Pa]
43 """
44
45     def __init__(self,
46                  name,
47                  density,
48                  elasticity_modulus,
49                  rigidity_modulus,
50                  poisson_ratio=0,
51                  yield_tensile_stress=0,
52                  ultimate_tensile_stress=0,
53                  yield_shear_stress=0,
54                  ultimate_shear_stress=0,
55                  ):
56
57         self.name = name
58         self.density = density
59         self.elasticity_modulus = elasticity_modulus
60         self.rigidity_modulus = rigidity_modulus
61         self.poisson_ratio = poisson_ratio
62         self.yield_tensile_stress = yield_tensile_stress
63         self.ultimate_tensile_stress = ultimate_tensile_stress
64         self.yield_shear_stress = yield_shear_stress
65         self.ultimate_shear_stress = ultimate_shear_stress
66
67
68 #
69 """
=====

70
71
72 class ElementProperty(object):
73     """ Property stores the properties of a beam finite element
74
75     Args:
76         section (object): A Section object from the Geometry module.
77         material (object): A Material object
78
79     Attributes:
80         section (object): A Section object from the Geometry module.
81         material (object): A Material object
82         A (float): Area of the section
83         J (float): Polar moment of inertia of the section
84         Iyy (float): Moment of inertia of the section in the Y axis
85         Izz (float): Moment of inertia of the section in the Z axis
86         E (float): Elastic modulus of the material
87         G (float): Shear modulus of the material
88 """
89
90     def __init__(self, section, material):
91
92         self.section = section
93         self.material = material
94         self.A = section.area
95         self.J = section.J
96         self.Iyy = section.Iyy
97         self.Izz = section.Izz
98         self.E = material.elasticity_modulus
99         self.G = material.rigidity_modulus
100
101
```

```

102 #
=====
103
104
105 class RigidConnection(object):
106     """ Property stores the properties of a rigid connection element
107
108     Attributes:
109         section (object): A Section object from the Geometry module.
110         material (object): A Material object
111         A (float): Area of the section
112         J (float): Polar moment of inertia of the section
113         Iyy (float): Moment of inertia of the section in the Y axis
114         Izz (float): Moment of inertia of the section in the Z axis
115         E (float): Elastic modulus of the material
116         G (float): Shear modulus of the material
117     """
118
119     def __init__(self):
120
121         self.section = "rigid_connection"
122         self.material = "rigid_connection"
123         self.A = 1
124         self.J = 1
125         self.Iyy = 1
126         self.Izz = 1
127         self.E = 1e100
128         self.G = 1e100
129
130
131 #
=====

132
133
134 class EulerBeamElement(object):
135     """ Defines a beam finite element, saves it's properties and calculates it's stiffness
136     matrices
137
138     Args:
139         node_A_index (int): index of the first node of the element in the FEM grid
140         node_B_index (int): index of the second node of the element in the FEM grid
141         rotation (float): rotation of the element section in relation to it's own axis
142         propertie (object): propertie object containing section and material characteristics
143
144     Attributes:
145         node_A_index (int): index of the first node of the element in the FEM grid
146         node_B_index (int): index of the second node of the element in the FEM grid
147         A (float): Area of the section
148         J (float): Polar moment of inertia of the section
149         Iyy (float): Moment of inertia of the section in the Y axis
150         Izz (float): Moment of inertia of the section in the Z axis
151         E (float): Elastic modulus of the material
152         G (float): Shear modulus of the material
153         correlation_vector (np.array): Vector with the indexes of the global DOF that correspond
154             to the element local DOF
155     """
156
157     def __init__(self, node_A, node_B, A, Iyy, Izz, J, E, G, prop_choice="MIDDLE"):
158
159         self.node_A = node_A
160         self.node_B = node_B
161         self.A = A
162         self.Iyy = Iyy
163         self.Izz = Izz
164         self.J = J

```

```

164     self.E = E
165     self.G = G
166     self.L = m.norm(node_B.xyz - node_A.xyz)
167     self.prop_choice = prop_choice
168
169     A_index = 6 * self.node_A.number
170     B_index = 6 * self.node_B.number
171     self.correlation_vector = np.array(
172         [
173             A_index,
174             A_index + 1,
175             A_index + 2,
176             A_index + 3,
177             A_index + 4,
178             A_index + 5,
179             B_index,
180             B_index + 1,
181             B_index + 2,
182             B_index + 3,
183             B_index + 4,
184             B_index + 5,
185         ]
186     )
187
188     def calc_rotation_matrix(self):
189         """ Calculates the rotation matrix of the element.
190
191         Args:
192             grid (list): list with all the node elements of the FEM mesh
193
194         Returns:
195             rotation_matrix (np.array): transformation matrix from the local coordinate system
196                                         to the global coordinate system.
197         """
198
199         if self.prop_choice == "ROOT":
200
201             orientation_node = self.node_A
202
203         elif self.prop_choice == "TIP":
204
205             orientation_node = self.node_B
206
207         elif self.prop_choice == "MIDDLE":
208
209             # Interpolate root and tip nodes to find a node in the middle
210             nodes_prop = geo.functions.interpolate_nodes(self.node_A, self.node_B, 3)
211             middle_node_prop = nodes_prop[1]
212             orientation_node = geo.objects.Node(
213                 middle_node_prop[0], middle_node_prop[1]
214             )
215
216             # Global Coordinate System
217             x_global = np.array([1.0, 0.0, 0.0])
218             y_global = np.array([0.0, 1.0, 0.0])
219             z_global = np.array([0.0, 0.0, 1.0])
220
221             # Calculate rotation matrix
222             zero = np.zeros((3, 3))
223             r = np.zeros((3, 3))
224
225             for i, local_axis in enumerate([
226                 orientation_node.x_axis,
227                 orientation_node.y_axis,
228                 orientation_node.z_axis,
229             ]):
230                 for j, global_axis in enumerate([x_global, y_global, z_global]):

```

```

231         r[i][j] = geo.functions.cos_between(local_axis, global_axis)
232
233     rotation_matrix = np.block(
234         [
235             [r, zero, zero, zero],
236             [zero, r, zero, zero],
237             [zero, zero, r, zero],
238             [zero, zero, zero, r],
239         ]
240     )
241
242     return rotation_matrix
243
244 def calc_K_local(self):
245     """ Calculates the local stiffness matrix of the element.
246
247     Args:
248         grid (list): list with all the node elements of the FEM mesh
249
250     Returns:
251         K_local (np.array(dtype=float)): element local stiffness matrix
252     """
253
254     # Calculate Local Stiffness Matrix
255     K_local = f.euler_beam_stiff(self.E, self.A, self.L, self.G, self.J, self.Iyy, self.Izz)
256
257     return K_local
258
259 def calc_K_global(self):
260     """ Apply the rotation matrix to the local stiffness matrix and calculate the element
261     global stiffness matrix.
262
263     Args:
264         grid (list): list with all the node elements of the FEM mesh
265
266     Returns:
267         K_global (np.array(dtype=float)): element global stiffness matrix
268     """
269
270     rotation_matrix = self.calc_rotation_matrix()
271     K_local = self.calc_K_local()
272
273     # Global Stiffness Matrix
274     K_global = rotation_matrix.transpose() @ (K_local @ rotation_matrix)
275
276     return K_global
277
278
279 #
=====

280
281
282 class Structure:
283     def __init__(self, points, beams):
284
285         self.points = points
286         self.beams = beams
287
288
289 #
=====

290
291
292 #
=====
```

```
293
294
295 class Section:
296     def __init__(self, area, rotation, m_inertia_y, m_inertia_z, polar_moment):
297
298         self.area = area
299         self.rotation = rotation
300         self.m_inertia_y = m_inertia_y
301         self.m_inertia_z = m_inertia_z
302         self.polar_moment = polar_moment
303
304
305 #
=====
306
307
308 class Load:
309     def __init__(self, application_node, load):
310
311         self.application_node = application_node
312         self.load = load
313
314
315
316 #
=====
317
318
319 class Constraint:
320     def __init__(self, application_node, dof_constraints):
321
322         self.application_node = application_node
323         self.dof_constraints = dof_constraints
324
325
326 #
=====
327
328
329 class Connection(object):
330     """Describes a connection between two components, in other words, set their connecting node
331     as a single node in the global matrix.
332
333     Args:
334         component1 (object): an object able to generate nodes
335         component1_node (string): "ROOT" or "TIP"
336         component2 (object): an object able to generate nodes
337         component2_node (string): "ROOT" or "TIP"
338
339     Attributes:
340         descriptor (list(string)): a list with two strings describing the connection, the strings
341             are
342                 composed by concatenating the component identifier with the
343                 node
344                     string with an hifen between
345     """
346
347     def __init__(self, component1, component1_node, component2, component2_node):
348
349         self.component1 = component1
350         self.component1_node = component1_node
351         self.component2 = component2
352         self.component2_node = component2_node
```

```

351
352     self.descriptor = [
353         component1.identifier + "-" + component1_node,
354         component2.identifier + "-" + component2_node,
355     ]

```

A.8 *visualization*

A.8.1 `__init__.py`

```

1 from . import plot_3D
2 from . import plot_3D2

```

A.8.2 `plot_2D.py`

```

1 """
2 visualization.py
3
4 Routines for results visualization
5
6 Author: João Paulo Monteiro Cruvinel da Costa
7 email: joaopaulomcc@gmail.com / joao.cruvinel@embraer.com.br
8 github: joaopaulomcc
9 """
10 #
=====

11 # IMPORTS
12 import numpy as np
13 import scipy as sc
14 import matplotlib
15 import matplotlib.pyplot as plt
16
17 from mpl_toolkits.mplot3d import axes3d
18 from numpy import sin, cos, tan, pi
19
20 from .. import mathematics as m
21
22 #
=====

23 # FUNCTIONS
24
25 def plot_deformation(
26     struct_elements,
27     struct_deformations,
28     ax,
29     plot_axis="Y",
30     scale_factor=1,
31     show_nodes=True,
32     line_color="k",
33     alpha=1,
34 ):
35
36     for component_elements in struct_elements:
37         for beam_element in component_elements:
38
39             node_A = beam_element.node_A

```

```

40         node_B = beam_element.node_B
41
42         point_A = [
43             node_A.x + scale_factor * struct_deformations[node_A.number][0],
44             node_A.y + scale_factor * struct_deformations[node_A.number][1],
45             node_A.z + scale_factor * struct_deformations[node_A.number][2],
46         ]
47
48         point_B = [
49             node_B.x + scale_factor * struct_deformations[node_B.number][0],
50             node_B.y + scale_factor * struct_deformations[node_B.number][1],
51             node_B.z + scale_factor * struct_deformations[node_B.number][2],
52         ]
53
54         x = [point_A[0], point_B[0]]
55         y = [point_A[1], point_B[1]]
56         z = [point_A[2], point_B[2]]
57
58     if show_nodes:
59         ax.plot(
60             x, y, z, color=line_color, marker="o", markersize=2, alpha=alpha
61         )
62     else:
63         ax.plot(x, y, z, color=line_color, alpha=alpha)
64
65     return ax

```

A.8.3 plot_3D.py

```

1 #
=====
2 # IMPORTS
3 import numpy as np
4 import scipy as sc
5 import matplotlib
6 import matplotlib.pyplot as plt
7
8 from mpl_toolkits.mplot3d import axes3d
9 from numpy import sin, cos, tan, pi
10
11 from .. import mathematics as m
12
13 #
=====

14 # FUNCTIONS
15
16
17 def set_axes_equal(ax):
18     """Make axes of 3D plot have equal scale so that spheres appear as spheres,
19     cubes as cubes, etc.. This is one possible solution to Matplotlib's
20     ax.set_aspect('equal') and ax.axis('equal') not working for 3D.
21
22     Source:
23     https://stackoverflow.com/questions/13685386/matplotlib-equal-unit-length-with-equal-aspect-
24     ratio-z-axis-is-not-equal-to
25     Input
26         ax: a matplotlib axis, e.g., as output from plt.gca().
27
28     x_limits = ax.get_xlim3d()
29     y_limits = ax.get_ylim3d()
30     z_limits = ax.get_zlim3d()

```

```

31
32     x_range = abs(x_limits[1] - x_limits[0])
33     x_middle = np.mean(x_limits)
34     y_range = abs(y_limits[1] - y_limits[0])
35     y_middle = np.mean(y_limits)
36     z_range = abs(z_limits[1] - z_limits[0])
37     z_middle = np.mean(z_limits)
38
39     # The plot bounding box is a sphere in the sense of the infinity
40     # norm, hence I call half the max range the plot radius.
41     plot_radius = 0.5 * max([x_range, y_range, z_range])
42
43     ax.set_xlim3d([x_middle - plot_radius, x_middle + plot_radius])
44     ax.set_ylim3d([y_middle - plot_radius, y_middle + plot_radius])
45     ax.set_zlim3d([z_middle - plot_radius, z_middle + plot_radius])
46
47
48 #
-----
```

```

49
50
51 def generate_blank_3D_plot(title=None, show_origin=True):
52
53     # Create figure and apply defaults
54     fig = plt.figure()
55     ax = fig.add_subplot(111, projection="3d", proj_type="persp")
56
57     # Add labels
58     ax.set_xlabel("X axis")
59     ax.set_ylabel("Y axis")
60     ax.set_zlabel("Z axis")
61     ax.set_title(title)
62
63     if show_origin:
64
65         # Plot coordinate system
66         ax.quiver(
67             [0, 0, 0],
68             [0, 0, 0],
69             [0, 0, 0],
70             [1, 0, 0],
71             [0, 1, 0],
72             [0, 0, 1],
73             color="black",
74         )
75         ax.scatter([0], [0], [0], color="red")
76
77     return ax, fig
78
79
80 #
-----
```

```

81
82
83 def plot_macrosurface(
84     macrosurface_aero_grid, ax, color="tab:blue", alpha=0.85, shade=False
85 ):
86
87     for surface_aero_grid in macrosurface_aero_grid:
88         xx = surface_aero_grid["xx"]
89         yy = surface_aero_grid["yy"]
90         zz = surface_aero_grid["zz"]
91         ax.plot_surface(xx, yy, zz, color=color, shade=shade, alpha=alpha)
92
93     return ax

```

```
94
95
96 # -----
97
98
99 def plot_macrosurface_aero_grid(
100     macrosurface_aero_grid, ax, line_color="darkblue", alpha=1
101 ):
102
103     for surface_aero_grid in macrosurface_aero_grid:
104         xx = surface_aero_grid["xx"]
105         yy = surface_aero_grid["yy"]
106         zz = surface_aero_grid["zz"]
107         ax.plot_wireframe(xx, yy, zz, color=line_color, alpha=alpha)
108
109     return ax
110
111
112 #
113
114
115 def plot_structure(struct_elements, ax, show_nodes=True, line_color="k", alpha=1):
116
117     for component_elements in struct_elements:
118         for beam_element in component_elements:
119
120             point_A = [
121                 beam_element.node_A.x,
122                 beam_element.node_A.y,
123                 beam_element.node_A.z,
124             ]
125             point_B = [
126                 beam_element.node_B.x,
127                 beam_element.node_B.y,
128                 beam_element.node_B.z,
129             ]
130             x = [point_A[0], point_B[0]]
131             y = [point_A[1], point_B[1]]
132             z = [point_A[2], point_B[2]]
133
134             if show_nodes:
135                 ax.plot(
136                     x, y, z, color=line_color, marker="o", markersize=2, alpha=alpha
137                 )
138             else:
139                 ax.plot(x, y, z, color=line_color, alpha=alpha)
140
141     return ax
142
143
144 #
145
146
147 def plot_deformed_structure(
148     struct_elements,
149     struct_deformations,
150     ax,
151     scale_factor=1,
152     show_nodes=True,
153     line_color="k",
154     alpha=1,
```

```
155 ):  
156  
157     for component_elements in struct_elements:  
158         for beam_element in component_elements:  
159  
160             node_A = beam_element.node_A  
161             node_B = beam_element.node_B  
162  
163             point_A = [  
164                 node_A.x + scale_factor * struct_deformations[node_A.number][0],  
165                 node_A.y + scale_factor * struct_deformations[node_A.number][1],  
166                 node_A.z + scale_factor * struct_deformations[node_A.number][2],  
167             ]  
168  
169             point_B = [  
170                 node_B.x + scale_factor * struct_deformations[node_B.number][0],  
171                 node_B.y + scale_factor * struct_deformations[node_B.number][1],  
172                 node_B.z + scale_factor * struct_deformations[node_B.number][2],  
173             ]  
174  
175             x = [point_A[0], point_B[0]]  
176             y = [point_A[1], point_B[1]]  
177             z = [point_A[2], point_B[2]]  
178  
179             if show_nodes:  
180                 ax.plot(  
181                     x, y, z, color=line_color, marker="o", markersize=2, alpha=alpha  
182                 )  
183             else:  
184                 ax.plot(x, y, z, color=line_color, alpha=alpha)  
185  
186     return ax  
187  
188  
189 #-----  
190  
191  
192 def generate_aircraft_grids_plot(  
193     aircraft_macrosurfaces_aero_grids,  
194     aircraft_struct_fem_elements=None,  
195     title=None,  
196     ax=None,  
197     fig=None,  
198     show_origin=True,  
199     show_nodes=False,  
200     line_color="k",  
201     alpha=1,  
202 ):  
203  
204     if ax is None:  
205         ax, fig = generate_blank_3D_plot(title, show_origin)  
206  
207     for macrosurface_aero_grid in aircraft_macrosurfaces_aero_grids:  
208         ax = plot_macrosurface_aero_grid(  
209             macrosurface_aero_grid, ax, line_color=line_color, alpha=alpha  
210         )  
211  
212     if aircraft_struct_fem_elements:  
213         for macrosurface_struct_fem_elements in aircraft_struct_fem_elements[  
214             "macrosurfaces_fem_elements"  
215         ]:  
216             ax = plot_structure(  
217                 macrosurface_struct_fem_elements,  
218                 ax,  
219                 show_nodes=show_nodes,
```

```
220             line_color=line_color,
221             alpha=alpha,
222         )
223
224     if aircraft_struct_fem_elements["beams_fem_elements"]:
225         ax = plot_structure(
226             aircraft_struct_fem_elements["beams_fem_elements"],
227             ax,
228             show_nodes=show_nodes,
229             line_color=line_color,
230             alpha=alpha,
231         )
232
233     set_axes_equal(ax)
234
235     return ax, fig
236
237
238 #
-----#
239
240
241 def generate_deformed_aircraft_grids_plot(
242     aircraft_deformed_macrosurfaces_aero_grids,
243     aircraft_struct_fem_elements,
244     aircraft_struct_deformations,
245     title=None,
246     ax=None,
247     fig=None,
248     show_origin=True,
249     show_nodes=False,
250     line_color="k",
251     alpha=1,
252 ):
253
254     if ax is None:
255         ax, fig = generate_blank_3D_plot(title, show_origin)
256
257     for macrosurface_aero_grid in aircraft_deformed_macrosurfaces_aero_grids:
258         ax = plot_macrosurface_aero_grid(
259             macrosurface_aero_grid, ax, line_color=line_color, alpha=alpha
260         )
261
262     for macrosurface_struct_fem_elements in aircraft_struct_fem_elements[
263         "macrosurfaces_fem_elements"
264     ]:
265
266         ax = plot_deformed_structure(
267             macrosurface_struct_fem_elements,
268             aircraft_struct_deformations,
269             ax,
270             scale_factor=1,
271             show_nodes=show_nodes,
272             line_color=line_color,
273             alpha=alpha,
274         )
275
276     if aircraft_struct_fem_elements["beams_fem_elements"]:
277
278         ax = plot_deformed_structure(
279             aircraft_struct_fem_elements["beams_fem_elements"],
280             aircraft_struct_deformations,
281             ax,
282             scale_factor=1,
283             show_nodes=show_nodes,
284             line_color=line_color,
```

```
285         alpha=alpha ,
286     )
287
288     set_axes_equal(ax)
289
290     return ax, fig
291
292
293 # -----
294
295
296 def generate_results_plot(
297     aircraft_deformed_macrosurfaces_aero_grids ,
298     aircraft_panel_loads ,
299     aircraft_struct_fem_elements=None ,
300     aircraft_struct_deformations=None ,
301     results_string="delta_p_grid",
302     title=None ,
303     colorbar_label="Delta Pressure [Pa]" ,
304     ax=None ,
305     fig=None ,
306     show_origin=True ,
307     colormap="coolwarm" ,
308 ):
309
310     if ax is None:
311         ax, fig = generate_blank_3D_plot(title, show_origin)
312
313     # Configure Color Map
314     min_result_value = 0
315     max_result_value = 0
316
317     for component_panel_loads in aircraft_panel_loads:
318
319         grid_min = component_panel_loads[results_string].min()
320         grid_max = component_panel_loads[results_string].max()
321
322         if grid_min < min_result_value:
323             min_result_value = grid_min
324
325         if grid_max > max_result_value:
326             max_result_value = grid_max
327
328     norm = matplotlib.colors.Normalize(min_result_value, max_result_value)
329     m = plt.cm.ScalarMappable(norm=norm, cmap=colormap)
330     m.set_array([])
331
332     fig.colorbar(m, shrink=0.5, aspect=5, label=colorbar_label)
333
334     # Plot Aircraft Surfaces
335     for i, deformed_macrosurfaces_aero_grid in enumerate(
336         aircraft_deformed_macrosurfaces_aero_grids
337     ):
338
339         results = aircraft_panel_loads[i][results_string]
340
341         index = 0
342
343         for deformed_surface_aero_grid in deformed_macrosurfaces_aero_grid:
344             n_chord_panels = np.shape(deformed_surface_aero_grid["xx"])[0] - 1
345             n_span_panels = np.shape(deformed_surface_aero_grid["xx"])[1] - 1
346
347             results_slice = results[:, index : (index + n_span_panels)]
348             index += n_span_panels
349
```

```

350     xx = deformed_surface_aero_grid["xx"]
351     yy = deformed_surface_aero_grid["yy"]
352     zz = deformed_surface_aero_grid["zz"]
353     fcolors = m.to_rgba(results_slice)
354     surf = ax.plot_surface(
355         xx,
356         yy,
357         zz,
358         facecolors=fcolors,
359         vmin=min_result_value,
360         vmax=max_result_value,
361         shade=False,
362         linewidth=0.5,
363         antialiased=False,
364     )
365
366 # Plot Aircraft Deformed Structure
367 if aircraft_struct_fem_elements:
368     for macrosurface_struct_fem_elements in aircraft_struct_fem_elements[
369         "macrosurfaces_fem_elements"
370     ]:
371
372         ax = plot_deformed_structure(
373             macrosurface_struct_fem_elements,
374             aircraft_struct_deformations,
375             ax,
376             scale_factor=1,
377             show_nodes=False,
378             line_color="k",
379             alpha=1,
380         )
381
382     if aircraft_struct_fem_elements["beams_fem_elements"]:
383         ax = plot_deformed_structure(
384             aircraft_struct_fem_elements["beams_fem_elements"],
385             aircraft_struct_deformations,
386             ax,
387             scale_factor=1,
388             show_nodes=False,
389             line_color="k",
390             alpha=1,
391         )
392
393     set_axes_equal(ax)
394
395     return ax, fig
396
397
398 #
-----
```

```

399
400
401 def generate_aircraft_plot(aircraft, title=None, ax=None, fig=None, show_origin=True):
402
403     if ax is None:
404         ax, fig = generate_blank_3D_plot(title, show_origin)
405
406     # Define color map to be used. With this is possible to plot each component in a different
407     # color
408     color_pallet = [
409         "tab:blue",
410         "tab:orange",
411         "tab:green",
412         "tab:red",
413         "tab:purple",
414         "tab:brown",
415     ]

```

```

414         "tab:pink",
415         "tab:gray",
416         "tab:olive",
417         "tab:cyan",
418     ]
419
420     # Plot CG
421     if aircraft.inertial_properties:
422         cg_x = aircraft.inertial_properties.position[0]
423         cg_y = aircraft.inertial_properties.position[1]
424         cg_z = aircraft.inertial_properties.position[2]
425
426         ax.scatter([cg_x], [cg_y], [cg_z], marker="D", color="black", s=25)
427
428     # Plot Engine and Thrust Vector
429     if aircraft.engines:
430         for engine in aircraft.engines:
431             eng_x = engine.position[0]
432             eng_y = engine.position[1]
433             eng_z = engine.position[2]
434
435             eng_t_x = engine.thrust_vector[0]
436             eng_t_y = engine.thrust_vector[1]
437             eng_t_z = engine.thrust_vector[2]
438
439             ax.scatter([eng_x], [eng_y], [eng_z], marker="P", color="red", s=50)
440             ax.quiver(
441                 [eng_x], [eng_y], [eng_z], [eng_t_x], [eng_t_y], [eng_t_z], color="red"
442             )
443
444     # Plot Aircraf Components
445     for i, macrosurface in enumerate(aircraft.macrosurfaces):
446
447         # Generate component mesh
448         n_chord_panels = 3
449         n_span_panels_list = [1 for i in range(len(macrosurface.surface_list))]
450         n_beam_elements_list = [1 for i in range(len(macrosurface.surface_list))]
451         chord_discretization = "linear"
452         span_discretization_list = [
453             "linear" for i in range(len(macrosurface.surface_list))
454         ]
455         torsion_function_list = [
456             "linear" for i in range(len(macrosurface.surface_list))
457         ]
458
459         macrosurface_aero_grid, macrosurface_nodes_list = macrosurface.create_grids(
460             n_chord_panels,
461             n_span_panels_list,
462             n_beam_elements_list,
463             chord_discretization,
464             span_discretization_list,
465             torsion_function_list,
466         )
467
468         plot_macrosurface(
469             macrosurface_aero_grid, ax, color=color_pallet[i], alpha=0.85, shade=False
470         )
471
472         for surface_node_list in macrosurface_nodes_list:
473             x = []
474             y = []
475             z = []
476
477             for node in surface_node_list:
478                 x.append(node.xyz[0])
479                 y.append(node.xyz[1])
480                 z.append(node.xyz[2])

```

```
481         ax.plot(x, y, z, c="black")
482
483
484     # Plot aircraft beams
485     if aircraft.beams:
486         for beam in aircraft.beams:
487             x = np.array([beam.root_point[0], beam.tip_point[0]])
488             y = np.array([beam.root_point[1], beam.tip_point[1]])
489             z = np.array([beam.root_point[2], beam.tip_point[2]])
490
491             if beam.ElementProperty.material == "rigid_connection":
492                 ax.plot(x, y, z, c="blue", ls="--")
493             else:
494                 ax.plot(x, y, z, c="black")
495
496     # Fix axes scale
497     set_axes_equal(ax)
498
499     return ax, fig
```

Apêndice B - Exemplos de Aplicação

B.1 Asa Smith

B.1.1 smith_wing_data.py

```
1 """
2 =====
3 CFD-Based Analysis of Nonlinear Aeroelastic Behavior of High-Aspect Ratio Wings
4
5 M. J. Smith, M. J. Patil, D. H. Hodges
6
7 Georgia Institute fo Technology, Atlanta
8
9 =====
10
11 Comparisson of the results obtained by in the paper above with those generated by the tool
12 developed
13 in this work
14 Author: João Paulo Monteiro Cruvinel da Costa
15 """
16
17 #
18 # IMPORTS
19
20 import numpy as np
21
22 # Import code sub packages
23 from context import flyingcircus
24 from flyingcircus import geometry as geo
25 from flyingcircus import structures as struct
26
27 #
28 # GEOMETRY DEFINITION
29
30 # Wing section
31
32 # Aifoil name, only informative
33 naca0012 = "NACA 0012"
34
```

```
35 # Material properties, all equal to one as Smith et al only provides the stiffness
    characteristics
36 # of the wing
37 MATERIAL = struct.objects.Material(
38     name="material",
39     density=0.75,
40     elasticity_modulus=1,
41     rigidity_modulus=1,
42     poisson_ratio=1,
43     yield_tensile_stress=1,
44     ultimate_tensile_stress=1,
45     yield_shear_stress=1,
46     ultimate_shear_stress=1,
47 )
48
49 # Wing section properties
50 WING_SECTION = geo.objects.Section(
51     identifier=naca0012,
52     material=MATERIAL,
53     area=2e6,
54     Iyy=2e4,
55     Izz=5e6,
56     J=1e4,
57     shear_center=0.5,
58 )
59
60 #
-----
```



```
61
62 # Wing surface
63
64 WING_ROOT_CHORD = 1
65 WING_TIP_CHORD = 1
66 SEMI_WING_LENGTH = 16
67 WING_SWEEP_ANGLE = 0
68 WING_DIHEDRAL = 0
69 WING_TIP_TORSION_ANGLE = 0
70
71
72 # Definition of the wing planform
73 left_wing_surface = geo.objects.Surface(
74     identifier="left_wing",
75     root_chord=WING_ROOT_CHORD,
76     root_section=WING_SECTION,
77     tip_chord=WING_TIP_CHORD,
78     tip_section=WING_SECTION,
79     length=SEMI_WING_LENGTH,
80     leading_edge_sweep_angle_deg=WING_SWEEP_ANGLE,
81     dihedral_angle_deg=WING_DIHEDRAL,
82     tip_torsion_angle_deg=WING_TIP_TORSION_ANGLE,
83     control_surface_hinge_position=None,
84 )
85
86 right_wing_surface = geo.objects.Surface(
87     identifier="right_wing",
88     root_chord=WING_ROOT_CHORD,
89     root_section=WING_SECTION,
90     tip_chord=WING_TIP_CHORD,
91     tip_section=WING_SECTION,
92     length=SEMI_WING_LENGTH,
93     leading_edge_sweep_angle_deg=WING_SWEEP_ANGLE,
94     dihedral_angle_deg=WING_DIHEDRAL,
95     tip_torsion_angle_deg=WING_TIP_TORSION_ANGLE,
96     control_surface_hinge_position=None,
97 )
98
```

```

99 # Creation of the wing macrosurface
100 wing = geo.objects.MacroSurface(
101     position=np.array([0, 0, 0]),
102     incidence=0,
103     surface_list=[left_wing_surface, right_wing_surface],
104     symmetry_plane="XZ",
105     torsion_center=0.5,
106 )
107
108 #
-----
```

109

110 # Aircraft definition

111

```

112 smith_wing = geo.objects.Aircraft(
113     name="Smith Wing",
114     macrosurfaces=[wing],
115     inertial_properties=geo.objects.MaterialPoint(position=np.array([0.25, 0, 0])),
116     ref_area=32,
117     mean_aero_chord=1,
118 )
```

B.1.2 smith_wing_simulation.py

```

1 """
2 =====
3 CFD-Based Analysis of Nonlinear Aeroelastic Behavior of High-Aspect Ratio Wings
4
5 M. J. Smith, M. J. Patil, D. H. Hodges
6
7 Georgia Institute fo Technology, Atlanta
8
9 =====
```

10

11 Comparisson of the results obtained by in the paper above with those generated by the tool

 developed

12 in this work

13

14 Author: João Paulo Monteiro Cruvinel da Costa

15 """
16
17 #

=====

18 # IMPORTS
19
20 # Import python scientific libraries
21 import matplotlib.pyplot as plt
22 import numpy as np
23 import scipy as sc
24 import sys
25
26 # Import code sub packages
27 from context import flyingcircus
28 from flyingcircus import aerodynamics as aero
29 from flyingcircus import aeroelasticity as aelast
30 from flyingcircus import control
31 from flyingcircus import flight_mechanics as flmec
32 from flyingcircus import geometry as geo
33 from flyingcircus import loads
34 from flyingcircus import structures as struct

```
35 from flyingcircus import visualization as vis
36
37 #
=====

38 # print()
39 # print("====")
40 # print("= VALIDATION OF AEROELASTIC CALCULATION      =")
41 # print("= VALIDATION CASE: CFD-Based Analysis of Nonlinear      =")
42 # print("= Aeroelastic Behavior of High-Aspect Ratio Wings      =")
43 # print("= AUTHORS: M. J. Smith, M. J. Patil, D. H. Hodges      =")
44 # print("====")
45 #
=====

46 # GEOMETRY DEFINITION
47
48 print("# Importing geometric data...")
49 from smith_wing_data import smith_wing
50
51 vis.plot_3D.plot_aircraft(smith_wing, title="Smith Wing")
52
53
54 #
=====

55 # GRID CREATION
56
57
58 # Number of panels and finite elements
59 N_CHORD_PANELS = 5
60 N_SPAN_PANELS = 40
61 N_BEAM_ELEMENTS = 2 * N_SPAN_PANELS
62 CHORD_DISCRETIZATION = "linear"
63 SPAN_DISCRETIZATION = "linear"
64 TORSION_FUNCTION = "linear"
65 CONTROL_SURFACE_DEFLECTION_DICT = dict()
66
67 wing_grid_data = {
68     "n_chord_panels": N_CHORD_PANELS,
69     "n_span_panels_list": [N_SPAN_PANELS, N_SPAN_PANELS],
70     "n_beam_elements_list": [N_BEAM_ELEMENTS, N_BEAM_ELEMENTS],
71     "chord_discretization": CHORD_DISCRETIZATION,
72     "span_discretization_list": [SPAN_DISCRETIZATION, SPAN_DISCRETIZATION],
73     "torsion_function_list": [TORSION_FUNCTION, TORSION_FUNCTION],
74     "control_surface_deflection_dict": CONTROL_SURFACE_DEFLECTION_DICT,
75 }
76
77 smith_wing_grid_data = {
78     "macrosurfaces_grid_data": [wing_grid_data],
79     "beams_grid_data": None,
80 }
81
82 # Creation of the smith wing grids
83
84 smith_wing_grids = aerlast.functions.generate_aircraft_grids(
85     aircraft_object=smith_wing, aircraft_grid_data=smith_wing_grid_data
86 )
87
88 #
=====

89 # STRUCTURE DEFINITION
90
91 # Create wing finite elements
92
93 smith_wing_fem_elements = struct.fem.generate_aircraft_fem_elements(
```

```

94     aircraft=smith_wing, aircraft_grids=smith_wing_grids, prop_choice="ROOT"
95 )
96
97 # Generate Constraints
98 wing_fixation = {
99     "component_identifier": "left_wing",
100    "fixation_point": "ROOT",
101    "dof_constraints": np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0]),
102 }
103
104 smith_wing_constraint_data = [wing_fixation]
105
106 smith_wing_constraints = aelast.functions.generate_aircraft_constraints(
107     aircraft=smith_wing,
108     aircraft_grids=smith_wing_grids,
109     constraints_data_list=smith_wing_constraint_data,
110 )
111
112 # ax, fig = vis.plot_3D.plot_aircraft_grids(smith_wing_grids, smith_wing_fem_elements, title="Smith Wing Grids")
113
114 # fig.show()
115
116 #
=====

117 # AERODYNAMIC LOADS CALCULATION - CASE 1 - ALPHA 2ž - Rigid Wing
118 print()
119 print("# CASE 000:")
120 print(f" - Altitude: 20000m")
121 print(f" - True Airspeed: 25m/s")
122 print(f" - Alpha: 2ž")
123 print(f" - Rigid Wing")
124 print()
125
126 # Flight Conditions definition
127
128 # Translation velocities
129 V_X = 25
130 V_Y = 0
131 V_Z = 0
132
133 # Rotation velocities
134 R_X = 0
135 R_Y = 0
136 R_Z = 0
137
138 # Aircraft Attitude in relation to the wind axis, in degrees
139 ALPHA = 2 # Pitch angle
140 BETA = 0 # Yaw angle
141 GAMMA = 0 # Roll angle
142
143 # Center of rotation, usually the aircraft CG position
144 CENTER_OF_ROTATION = np.array([0, 0, 0])
145
146 # Flight altitude, used to calculate atmospheric conditions, in meters
147 ALTITUDE = 20000
148
149 # Atmospheric turbulence, function that calculates the air speeds in relation to the ground,
150 # given
151
152
153 def ATM_TURBULENCE_FUNCTION(point_coordinates):
154
155     return np.zeros(3)
156

```

```

157
158 FLIGHT_CONDITIONS_DATA = {
159     "translation_velocity": np.array([V_X, V_Y, V_Z]),
160     "rotation_velocity": np.array([R_X, R_Y, R_Z]),
161     "attitude_angles_deg": np.array([ALPHA, BETA, GAMMA]),
162     "center_of_rotation": CENTER_OF_ROTATION,
163     "altitude": ALTITUDE,
164     "atm_turbulence_function": ATM_TURBULENCE_FUNCTION,
165     "center_of_rotation": CENTER_OF_ROTATION,
166 }
167
168 SIMULATION_OPTIONS = {
169     "flexible_aircraft": False,
170     "status_messages": True,
171     "control_node_string": "left_wing-TIP",
172     "max_iterations": 100,
173     "bending_convergence_criteria": 0.01,
174     "torsion_convergence_criteria": 0.01,
175     "fem_prop_choice": "ROOT",
176     "interaction_algorithm": "closest",
177     "output_iteration_results": True,
178 }
179
180 results_case0 = aelast.functions.calculate_aircraft_loads(
181     aircraft_object=smith_wing,
182     aircraft_grid_data=smith_wing_grid_data,
183     aircraft_constraints_data=smith_wing_constraints_data,
184     flight_condition_data=FLIGHT_CONDITIONS_DATA,
185     simulation_options=SIMULATION_OPTIONS,
186     influence_coef_matrix=None,
187 )
188
189 #
=====

190 # AERODYNAMIC LOADS CALCULATION - CASE 1 - ALPHA 2ž - Rigid Wing
191 print()
192 print("# CASE 001:")
193 print(f"    - Altitude: 20000m")
194 print(f"    - True Airspeed: 25m/s")
195 print(f"    - Alpha: 4ž")
196 print(f"    - Rigid Wing")
197 print()
198
199 # Flight Conditions definition
200
201 # Translation velocities
202 V_X = 25
203 V_Y = 0
204 V_Z = 0
205
206 # Rotation velocities
207 R_X = 0
208 R_Y = 0
209 R_Z = 0
210
211 # Aircraft Attitude in relation to the wind axis, in degrees
212 ALPHA = 4 # Pitch angle
213 BETA = 0 # Yaw angle
214 GAMMA = 0 # Roll angle
215
216 # Center of rotation, usually the aircraft CG position
217 CENTER_OF_ROTATION = np.array([0, 0, 0])
218
219 # Flight altitude, used to calculate atmospheric conditions, in meters
220 ALTITUDE = 20000
221

```

```

222 # Atmospheric turbulence , function that calculates the air speeds in relation to the ground ,
223 # given
224 # a point coordinates
225
226 def ATM_TURBULENCE_FUNCTION(point_coordinates):
227
228     return np.zeros(3)
229
230
231 FLIGHT_CONDITIONS_DATA = {
232     "translation_velocity": np.array([V_X, V_Y, V_Z]),
233     "rotation_velocity": np.array([R_X, R_Y, R_Z]),
234     "attitude_angles_deg": np.array([ALPHA, BETA, GAMMA]),
235     "center_of_rotation": CENTER_OF_ROTATION,
236     "altitude": ALTITUDE,
237     "atm_turbulence_function": ATM_TURBULENCE_FUNCTION,
238     "center_of_rotation": CENTER_OF_ROTATION,
239 }
240
241 SIMULATION_OPTIONS = {
242     "flexible_aircraft": False,
243     "status_messages": True,
244     "control_node_string": "left_wing-TIP",
245     "max_iterations": 100,
246     "bending_convergence_criteria": 0.01,
247     "torsion_convergence_criteria": 0.01,
248     "fem_prop_choice": "ROOT",
249     "interaction_algorithm": "closest",
250     "output_iteration_results": True,
251 }
252
253 results_case1 = aelast.functions.calculate_aircraft_loads(
254     aircraft_object=smith_wing,
255     aircraft_grid_data=smith_wing_grid_data,
256     aircraft_constraints_data=smith_wing_constraints_data,
257     flight_condition_data=FLIGHT_CONDITIONS_DATA,
258     simulation_options=SIMULATION_OPTIONS,
259     influence_coef_matrix=None,
260 )
261
262 #
263 # =====
264 # AERODYNAMIC LOADS CALCULATION - CASE 2 - ALPHA 22° - Flexible Wing
265 print()
266 print("# CASE 002:")
267 print(f"    - Altitude: 20000m")
268 print(f"    - True Airspeed: 25m/s")
269 print(f"    - Alpha: 22°")
270 print(f"    - Rigid Wing")
271 print()
272 # Flight Conditions definition
273
274 # Translation velocities
275 V_X = 25
276 V_Y = 0
277 V_Z = 0
278
279 # Rotation velocities
280 R_X = 0
281 R_Y = 0
282 R_Z = 0
283
284 # Aircraft Attitude in relation to the wind axis, in degrees
285 ALPHA = 2 # Pitch angle

```

```

286 BETA = 0 # Yaw angle
287 GAMMA = 0 # Roll angle
288
289 # Center of rotation, usually the aircraft CG position
290 CENTER_OF_ROTATION = np.array([0, 0, 0])
291
292 # Flight altitude, used to calculate atmospheric conditions, in meters
293 ALTITUDE = 20000
294
295 # Atmospheric turbulence, function that calculates the air speeds in relation to the ground,
296 # given
297 # a point coordinates
298
299 #def ATM_TURBULENCE_FUNCTION(point_coordinates):
300 #
301 #     return np.zeros(3)
302
303 FLIGHT_CONDITIONS_DATA = {
304     "translation_velocity": np.array([V_X, V_Y, V_Z]),
305     "rotation_velocity": np.array([R_X, R_Y, R_Z]),
306     "attitude_angles_deg": np.array([ALPHA, BETA, GAMMA]),
307     "center_of_rotation": CENTER_OF_ROTATION,
308     "altitude": ALTITUDE,
309     "atm_turbulence_function": ATM_TURBULENCE_FUNCTION,
310     "center_of_rotation": CENTER_OF_ROTATION,
311 }
312
313 SIMULATION_OPTIONS = {
314     "flexible_aircraft": True,
315     "status_messages": True,
316     "control_node_string": "left_wing-TIP",
317     "max_iterations": 100,
318     "bending_convergence_criteria": 0.01,
319     "torsion_convergence_criteria": 0.01,
320     "fem_prop_choice": "ROOT",
321     "interaction_algorithm": "closest",
322     "output_iteration_results": True,
323 }
324
325 results_case2, iteration_results_case2 = aelast.functions.calculate_aircraft_loads(
326     aircraft_object=smith_wing,
327     aircraft_grid_data=smith_wing_grid_data,
328     aircraft_constraints_data=smith_wing_constraints_data,
329     flight_condition_data=FLIGHT_CONDITIONS_DATA,
330     simulation_options=SIMULATION_OPTIONS,
331     influence_coef_matrix=None,
332 )
333 #
334 ==
335 # AERODYNAMIC LOADS CALCULATION - CASE 3 - ALPHA 4ž - Flexible Wing
336 print()
337 print("# CASE 002:")
338 print(f"    - Altitude: 20000m")
339 print(f"    - True Airspeed: 25m/s")
340 print(f"    - Alpha: 2ž")
341 print(f"    - Rigid Wing")
342 print()
343
344 # Flight Conditions definition
345
346 # Translation velocities
347 V_X = 25
348 V_Y = 0
349 V_Z = 0

```

```

350
351 # Rotation velocities
352 R_X = 0
353 R_Y = 0
354 R_Z = 0
355
356 # Aircraft Attitude in relation to the wind axis, in degrees
357 ALPHA = 4 # Pitch angle
358 BETA = 0 # Yaw angle
359 GAMMA = 0 # Roll angle
360
361 # Center of rotation, usually the aircraft CG position
362 CENTER_OF_ROTATION = np.array([0, 0, 0])
363
364 # Flight altitude, used to calculate atmospheric conditions, in meters
365 ALTITUDE = 20000
366
367 # Atmospheric turbulence, function that calculates the air speeds in relation to the ground,
368 # given
369 # a point coordinates
370
371 #def ATM_TURBULENCE_FUNCTION(point_coordinates):
372 #
373 #     return np.zeros(3)
374
375 FLIGHT_CONDITIONS_DATA = {
376     "translation_velocity": np.array([V_X, V_Y, V_Z]),
377     "rotation_velocity": np.array([R_X, R_Y, R_Z]),
378     "attitude_angles_deg": np.array([ALPHA, BETA, GAMMA]),
379     "center_of_rotation": CENTER_OF_ROTATION,
380     "altitude": ALTITUDE,
381     "atm_turbulenc_function": ATM_TURBULENCE_FUNCTION,
382     "center_of_rotation": CENTER_OF_ROTATION,
383 }
384
385 SIMULATION_OPTIONS = {
386     "flexible_aircraft": True,
387     "status_messages": True,
388     "control_node_string": "left_wing-TIP",
389     "max_iterations": 100,
390     "bending_convergence_criteria": 0.01,
391     "torsion_convergence_criteria": 0.01,
392     "fem_prop_choice": "ROOT",
393     "interaction_algorithm": "closest",
394     "output_iteration_results": True,
395 }
396
397 results_case3, iteration_results_case3 = aelast.functions.calculate_aircraft_loads(
398     aircraft_object=smith_wing,
399     aircraft_grid_data=smith_wing_grid_data,
400     aircraft_constraints_data=smith_wing_constraints_data,
401     flight_condition_data=FLIGHT_CONDITIONS_DATA,
402     simulation_options=SIMULATION_OPTIONS,
403     influence_coef_matrix=None,
404 )

```

B.1.3 smith_wing_results.py

```

1 """
2 =====
3 CFD-Based Analysis of Nonlinear Aeroelastic Behavior of High-Aspect Ratio Wings
4

```

```
5 M. J. Smith, M. J. Patil, D. H. Hodges
6
7 Georgia Institute fo Technology, Atlanta
8
9 =====
10
11 Comparisson of the results obtained by in the paper above with those generated by the tool
   developed
12 in this work
13
14 Author: João Paulo Monteiro Cruvinel da Costa
15 """
16
17 #
18 # IMPORTS
19
20 # Import python scientific libraries
21 import matplotlib.pyplot as plt
22 import numpy as np
23 import scipy as sc
24 import sys
25 import pickle
26
27
28 # Import code sub packages
29 from context import flyingcircus
30 from flyingcircus import aerodynamics as aero
31 from flyingcircus import aeroelasticity as aelast
32 from flyingcircus import control
33 from flyingcircus import flight_mechanics as flmec
34 from flyingcircus import geometry as geo
35 from flyingcircus import loads
36 from flyingcircus import structures as struct
37 from flyingcircus import visualization as vis
38
39 #
40 print()
41 print("===== ")
42 print("= VALIDATION OF AEROELASTIC CALCULATION      =")
43 print("= VALIDATION CASE: CFD-Based Analysis of Nonlinear      =")
44 print("= Aeroelastic Behavior of High-Aspect Ratio Wings      =")
45 print("= AUTHORS: M. J. Smith, M. J. Patil, D. H. Hodges      =")
46 print("===== ")
47 #
48 # EXECUTE CALCULATION
49
50 from smith_wing_data import smith_wing
51
52 #from smith_wing_simulation import (
53 #     results_case1,
54 #     results_case2,
55 #     iteration_results_case2,
56 #     results_case3,
57 #     iteration_results_case3,
58 #)
59
60 #f = open("results\\smith_wing\\results.pckl", "wb")
61 #pickle.dump(
62 #     [
63 #         results_case1,
```

```
64 #         results_case2,
65 #         iteration_results_case2,
66 #         results_case3,
67 #         iteration_results_case3,
68 #     ],
69 #     f,
70 #)
71 #f.close()
72
73 f = open("results\\smith_wing\\results\\results.pckl", "rb")
74 results_case1, results_case2, iteration_results_case2, results_case3, iteration_results_case3 =
    pickle.load(
75     f
76 )
77 f.close()
78
79 # Draw Aircraft
80 aircraft_ax, aircraft_fig = vis.plot_3D2.generate_aircraft_plot(
81     smith_wing, title="Smith Wing"
82 )
83
84 #
=====

85 # PROCESSING RESULTS
86
87 # CASE 001:
88 # - Alpha: 2°
89 # - Speed: 25 m/s
90 # - Altitude: 20000 m
91 # - Rigid
92
93 # Generate Original vs Deformed Grid Plot
94
95 # Draw original grids
96 grids_ax, grids_fig = vis.plot_3D2.generate_aircraft_grids_plot(
97     aircraft_macrosurfaces_aero_grids=results_case1["aircraft_original_grids"]["
        macrosurfaces_aero_grids"],
98     aircraft_struct_fem_elements=None,
99     title="Smith Wing - Case 001 - Aerodynamic Grids",
100    ax=None,
101    show_origin=True,
102    show_nodes=False,
103    line_color="k",
104    alpha=0.5,
105 )
106
107
108 # Calculate Loads on each of the aerodynamic panels
109 aircraft_panel_loads = loads.functions.calculate_aircraft_panel_loads(
110     results_case1["aircraft_macrosurfaces_panels"], results_case1["aircraft_force_grid"]
111 )
112
113 results_ax, results_fig = vis.plot_3D2.generate_results_plot(
114     aircraft_deformed_macrosurfaces_aero_grids=results_case1["aircraft_original_grids"]["
        macrosurfaces_aero_grids"],
115     aircraft_panel_loads=aircraft_panel_loads,
116     aircraft_struct_fem_elements=None,
117     aircraft_struct_deformations=None,
118     results_string="delta_p_grid",
119     title="Smith Wing - Case 001 - Delta Pressure [Pa]",
120     colorbar_label="Delta Pressure [Pa]",
121     ax=None,
122     fig=None,
123     show_origin=True,
124     colormap="coolwarm",
125 )
```

```

126
127 interest_point = smith_wing.inertial_properties.position
128
129 # Aerodynamic forces in the aircraft coordinate system
130 total_cg_aero_force, total_cg_aero_moment, component_cg_aero_loads = loads.functions.
    calc_aero_loads_at_point(
131     interest_point,
132     results_case1["aircraft_force_grid"],
133     results_case1["aircraft_macrosurfaces_panels"],
134 )
135
136 print()
137 print("#####")
138 print("#          CASE 001 RESULTS          #")
139 print("#####")
140
141 print()
142 print(f"# Total loads at aircraft CG:")
143 print(f"  FX: {total_cg_aero_force[0]} N")
144 print(f"  FY: {total_cg_aero_force[1]} N")
145 print(f"  FZ: {total_cg_aero_force[2]} N")
146 print(f"  RX: {total_cg_aero_moment[0]} N")
147 print(f"  RY: {total_cg_aero_moment[1]} N")
148 print(f"  RZ: {total_cg_aero_moment[2]} N")
149
150 V_X = 25
151 V_Y = 0
152 V_Z = 0
153
154 # Rotation velocities
155 R_X = 0
156 R_Y = 0
157 R_Z = 0
158
159 # Aircraft Attitude in relation to the wind axis, in degrees
160 ALPHA = 2 # Pitch angle
161 BETA = 0 # Yaw angle
162 GAMMA = 0 # Roll angle
163
164 # Center of rotation, usually the aircraft CG position
165 CENTER_OF_ROTATION = smith_wing.inertial_properties.position
166
167 # Flight altitude, used to calculate atmospheric conditions, in meters
168 ALTITUDE = 20000
169
170 forces, moments, coefficients = loads.functions.calc_lift_drag(
171     aircraft=smith_wing,
172     point=interest_point,
173     speed=V_X,
174     altitude=ALTITUDE,
175     attitude_vector=np.array([ALPHA, BETA, GAMMA]),
176     aircraft_force_grid=results_case1["aircraft_force_grid"],
177     aircraft_panel_grid=results_case1["aircraft_macrosurfaces_panels"],
178 )
179
180 print()
181 print("# Aerodynamic Coeffients:")
182 print(f"  - Lift: {forces['lift']} N")
183 print(f"  - Cl: {coefficients['Cl']}")
184 print(f"  - Drag: {forces['drag']} N")
185 print(f"  - Cd: {coefficients['Cd']}")
186 print(f"  - Pitch Moment: {moments['pitch_moment']} N.m")
187 print(f"  - Cm: {coefficients['Cm']}")
188
189 # Create load distribution plots
190 components_loads = loads.functions.calc_load_distribution(
191     aircraft_force_grid=results_case1["aircraft_force_grid"],

```

```

192     aircraft_panel_grid=results_case1["aircraft_macrosurfaces_panels"],
193     aircraft_gamma_grid=results_case1["aircraft_gamma_grid"],
194     attitude_vector=np.array([ALPHA, BETA, GAMMA]),
195     altitude=ALTITUDE,
196     speed=V_X,
197 )
198
199 for component in components_loads:
200     fig = plt.figure()
201     ax1 = fig.add_subplot(3, 1, 1)
202     ax1.set_title("Smith Wing - Case 001 - Lift Distribution")
203     ax1.set_xlabel("Span Position [m]")
204     ax1.set_ylabel("Lift [N]")
205     ax1.plot(component["y_values"], component["lift"])
206     ax1.grid()
207
208     ax2 = fig.add_subplot(3, 1, 2)
209     ax2.set_title("Smith Wing - Case 001 - Cl Distribution")
210     ax2.set_xlabel("Span Position [m]")
211     ax2.set_ylabel("Cl")
212     ax2.plot(component["y_values"], component["Cl"])
213     ax2.grid()
214
215     ax3 = fig.add_subplot(3, 1, 3)
216     ax3.set_title("Smith Wing - Case 001 - Drag Distribution")
217     ax3.set_xlabel("Span Position [m]")
218     ax3.set_ylabel("Drag [N]")
219     ax3.plot(component["y_values"], component["drag"])
220     ax3.grid()
221     plt.tight_layout()
222
223 #plt.show()
224
225 #
-----
```

```

226
227 # CASE 002:
228 #   - Alpha: 2̢
229 #   - Speed: 25 m/s
230 #   - Altitude: 20000 m
231 #   - Flexible
232
233 # Generate Original vs Deformed Grid Plot
234
235 # Draw original grids
236 grids_ax, grids_fig = vis.plot_3D2.generate_aircraft_grids_plot(
237     results_case2["aircraft_original_grids"]["macrosurfaces_aero_grids"],
238     results_case2["aircraft_struct_fem_elements"],
239     title="Smith Wing - Case 002 - Original vs Deformed Grids",
240     ax=None,
241     show_origin=True,
242     show_nodes=False,
243     line_color="k",
244     alpha=0.5,
245 )
246
247
248 # Draw deformed Grids
249 grids_ax, grids_fig = vis.plot_3D2.generate_deformed_aircraft_grids_plot(
250     results_case2["aircraft_deformed_macrosurfaces_aero_grids"],
251     results_case2["aircraft_struct_fem_elements"],
252     results_case2["aircraft_struct_deformations"],
253     ax=grids_ax,
254     fig=grids_fig,
255     show_origin=True,
256     show_nodes=False,
```

```
257     line_color="r",
258     alpha=1,
259 )
260
261 # Calculate Loads on each of the aerodynamic panels
262 aircraft_panel_loads = loads.functions.calculate_aircraft_panel_loads(
263     results_case2["original_aircraft_panel_grid"], results_case2["aircraft_force_grid"]
264 )
265
266 results_ax, results_fig = vis.plot_3D2.generate_results_plot(
267     aircraft_deformed_macrosurfaces_aero_grids=results_case2[
268         "aircraft_deformed_macrosurfaces_aero_grids"],
269     aircraft_panel_loads=aircraft_panel_loads,
270     aircraft_struct_fem_elements=results_case2["aircraft_struct_fem_elements"],
271     aircraft_struct_deformations=results_case2["aircraft_struct_deformations"],
272     results_string="delta_p_grid",
273     title="Smith Wing - Case 002 - Delta Pressure [Pa]",
274     colorbar_label="Delta Pressure [Pa]",
275     ax=None,
276     fig=None,
277     show_origin=True,
278     colormap="coolwarm",
279 )
280
281
282 # Deformation plot
283
284 deformation_table = aelast.functions.calculate_deformation_table(
285     results_case2["aircraft_original_grids"],
286     results_case2["aircraft_struct_deformations"],
287 )
288
289 # sort nodes by desired column, in this case the Y coordinate
290
291 nodes = deformation_table["aircraft_macrosurfaces_deformed_nodes"][0]
292 nodes = nodes[nodes[:, 1].argsort()]
293
294 # Plot Bending
295 fig, ax = plt.subplots()
296 ax.plot(nodes[:, 1], nodes[:, 2])
297 ax.grid()
298 ax.set_title("Smith Wing - Case 002 - Bending")
299 ax.set_ylabel("Bending [m]")
300 ax.set_xlabel("Span [m]")
301
302 # Plot Torsion
303 fig, ax = plt.subplots()
304 ax.plot(nodes[:, 1], np.degrees(nodes[:, 4]))
305 ax.grid()
306 ax.set_title("Smith Wing - Case 002 - Torsion")
307 ax.set_ylabel("Torsion [degrees]")
308 ax.set_xlabel("Span [m]")
309
310 interest_point = smith_wing.inertial_properties.position
311
312 # Aerodynamic forces in the aircraft coordinate system
313 total_cg_aero_force, total_cg_aero_moment, component_cg_aero_loads = loads.functions.
314     calc_aero_loads_at_point(
315         interest_point,
316         results_case2["aircraft_force_grid"],
317         results_case2["aircraft_deformed_macrosurfaces_aero_panels"],
318     )
319 print()
320 print("#####")
321 print(" CASE 002 RESULTS      #")
```

```

322 print("#####")
323 print()
324 print(f"# Total loads at aircraft CG:")
325 print(f"    FX: {total_cg_aero_force[0]} N")
326 print(f"    FY: {total_cg_aero_force[1]} N")
327 print(f"    FZ: {total_cg_aero_force[2]} N")
328 print(f"    RX: {total_cg_aero_moment[0]} N")
329 print(f"    RY: {total_cg_aero_moment[1]} N")
330 print(f"    RZ: {total_cg_aero_moment[2]} N")
331
332 V_X = 25
333 V_Y = 0
334 V_Z = 0
335
336 # Rotation velocities
337 R_X = 0
338 R_Y = 0
339 R_Z = 0
340
341 # Aircraft Attitude in relation to the wind axis, in degrees
342 ALPHA = 2 # Pitch angle
343 BETA = 0 # Yaw angle
344 GAMMA = 0 # Roll angle
345
346 # Center of rotation, usually the aircraft CG position
347 CENTER_OF_ROTATION = smith_wing.inertial_properties.position
348
349 # Flight altitude, used to calculate atmospheric conditions, in meters
350 ALTITUDE = 20000
351
352 forces, moments, coefficients = loads.functions.calc_lift_drag(
353     aircraft=smith_wing,
354     point=interest_point,
355     speed=V_X,
356     altitude=ALTITUDE,
357     attitude_vector=np.array([ALPHA, BETA, GAMMA]),
358     aircraft_force_grid=results_case2["aircraft_force_grid"],
359     aircraft_panel_grid=results_case2["aircraft_deformed_macrosurfaces_aero_panels"],
360 )
361
362 print()
363 print("# Aerodynamic Coeffients:")
364 print(f"    - Lift: {forces['lift']} N")
365 print(f"    - Cl: {coefficients['C1']}")
366 print(f"    - Drag: {forces['drag']} N")
367 print(f"    - Cd: {coefficients['Cd']}")
368 print(f"    - Pitch Moment: {moments['pitch_moment']} N.m")
369 print(f"    - Cm: {coefficients['Cm']}")
370
371 # Create load distribution plots
372 components_loads = loads.functions.calc_load_distribution(
373     aircraft_force_grid=results_case2["aircraft_force_grid"],
374     aircraft_panel_grid=results_case2["original_aircraft_panel_grid"],
375     aircraft_gamma_grid=results_case1["aircraft_gamma_grid"],
376     attitude_vector=np.array([ALPHA, BETA, GAMMA]),
377     altitude=ALTITUDE,
378     speed=V_X,
379 )
380
381 for component in components_loads:
382     fig = plt.figure()
383
384     ax1 = fig.add_subplot(3, 1, 1)
385     ax1.set_title("Smith Wing - Case 002 - Lift Distribution")
386     ax1.set_xlabel("Span Position [m]")
387     ax1.set_ylabel("Lift [N]")
388     ax1.plot(component["y_values"], component["lift"])

```

```
389     ax1.grid()
390
391     ax2 = fig.add_subplot(3, 1, 2)
392     ax2.set_title("Smith Wing - Case 002 - Cl Distribution")
393     ax2.set_xlabel("Span Position [m]")
394     ax2.set_ylabel("Cl")
395     ax2.plot(component["y_values"], component["Cl"])
396     ax2.grid()
397
398     ax3 = fig.add_subplot(3, 1, 3)
399     ax3.set_title("Smith Wing - Case 002 - Drag Distribution")
400     ax3.set_xlabel("Span Position [m]")
401     ax3.set_ylabel("Drag [N]")
402     ax3.plot(component["y_values"], component["drag"])
403     ax3.grid()
404     plt.tight_layout()
405
406 #plt.show()
407 #
408 #
409
410 # CASE 003:
411 # - Alpha: 4°
412 # - Speed: 25 m/s
413 # - Altitude: 20000 m
414 # - Flexible
415
416 # Generate Original vs Deformed Grid Plot
417
418 # Draw original grids
419 grids_ax, grids_fig = vis.plot_3D2.generate_aircraft_grids_plot(
420     results_case3["aircraft_original_grids"]["macrosurfaces_aero_grids"],
421     results_case3["aircraft_struct_fem_elements"],
422     title="Smith Wing - Case 003 - Original vs Deformed Grids",
423     ax=None,
424     show_origin=True,
425     show_nodes=False,
426     line_color="k",
427     alpha=0.5,
428 )
429
430
431 # Draw deformed Grids
432 grids_ax, grids_fig = vis.plot_3D2.generate_deformed_aircraft_grids_plot(
433     results_case3["aircraft_deformed_macrosurfaces_aero_grids"],
434     results_case3["aircraft_struct_fem_elements"],
435     results_case3["aircraft_struct_deformations"],
436     ax=grids_ax,
437     fig=grids_fig,
438     show_origin=True,
439     show_nodes=False,
440     line_color="r",
441     alpha=1,
442 )
443
444 # Calculate Loads on each of the aerodynamic panels
445 aircraft_panel_loads = loads.functions.calculate_aircraft_panel_loads(
446     results_case3["original_aircraft_panel_grid"], results_case3["aircraft_force_grid"]
447 )
448
449 results_ax, results_fig = vis.plot_3D2.generate_results_plot(
450     aircraft_deformed_macrosurfaces_aero_grids=results_case3[
451         "aircraft_deformed_macrosurfaces_aero_grids"],
452     aircraft_panel_loads=aircraft_panel_loads,
453     aircraft_struct_fem_elements=results_case3["aircraft_struct_fem_elements"],
```

```

453     aircraft_struct_deformations=results_case3["aircraft_struct_deformations"],
454     results_string="delta_p_grid",
455     title="Smith Wing - Case 003 - Delta Pressure [Pa]",
456     colorbar_label="Delta Pressure [Pa]",
457     ax=None,
458     fig=None,
459     show_origin=True,
460     colormap="coolwarm",
461 )
462
463 # plt.show()
464
465 # Deformation plot
466
467 deformation_table = aelast.functions.calculate_deformation_table(
468     results_case3["aircraft_original_grids"],
469     results_case3["aircraft_struct_deformations"],
470 )
471
472 # sort nodes by desired column, in this case the Y coordinate
473
474 nodes = deformation_table["aircraft_macrosurfaces_deformed_nodes"] [0]
475 nodes = nodes[nodes[:, 1].argsort()]
476
477 # Plot Bending
478 fig, ax = plt.subplots()
479 ax.plot(nodes[:, 1], nodes[:, 2])
480 ax.grid()
481 ax.set_title("Smith Wing - Case 003 - Bending")
482 ax.set_ylabel("Bending [m]")
483 ax.set_xlabel("Span [m]")
484
485 # Plot Torsion
486 fig, ax = plt.subplots()
487 ax.plot(nodes[:, 1], np.degrees(nodes[:, 4]))
488 ax.grid()
489 ax.set_title("Smith Wing - Case 003 - Torsion")
490 ax.set_ylabel("Torsion [degrees]")
491 ax.set_xlabel("Span [m]")
492
493 interest_point = smith_wing.inertial_properties.position
494
495 # Aerodynamic forces in the aircraft coordinate system
496 total_cg_aero_force, total_cg_aero_moment, component_cg_aero_loads = loads.functions.
        calc_aero_loads_at_point(
497     interest_point,
498     results_case3["aircraft_force_grid"],
499     results_case3["aircraft_deformed_macrosurfaces_aero_panels"],
500 )
501
502 print()
503 print("#####")
504 print("#          CASE 003 RESULTS          #")
505 print("#####")
506 print()
507 print(f"# Total loads at aircraft CG:")
508 print(f"    FX: {total_cg_aero_force[0]} N")
509 print(f"    FY: {total_cg_aero_force[1]} N")
510 print(f"    FZ: {total_cg_aero_force[2]} N")
511 print(f"    RX: {total_cg_aero_moment[0]} N")
512 print(f"    RY: {total_cg_aero_moment[1]} N")
513 print(f"    RZ: {total_cg_aero_moment[2]} N")
514
515 V_X = 25
516 V_Y = 0
517 V_Z = 0
518

```

```

519 # Rotation velocities
520 R_X = 0
521 R_Y = 0
522 R_Z = 0
523
524 # Aircraft Attitude in relation to the wind axis, in degrees
525 ALPHA = 2 # Pitch angle
526 BETA = 0 # Yaw angle
527 GAMMA = 0 # Roll angle
528
529 # Center of rotation, usually the aircraft CG position
530 CENTER_OF_ROTATION = smith_wing.inertial_properties.position
531
532 # Flight altitude, used to calculate atmospheric conditions, in meters
533 ALTITUDE = 20000
534
535 forces, moments, coefficients = loads.functions.calc_lift_drag(
536     aircraft=smith_wing,
537     point=interest_point,
538     speed=V_X,
539     altitude=ALTITUDE,
540     attitude_vector=np.array([ALPHA, BETA, GAMMA]),
541     aircraft_force_grid=results_case3["aircraft_force_grid"],
542     aircraft_panel_grid=results_case3["aircraft_deformed_macrosurfaces_aero_panels"],
543 )
544
545 print()
546 print("# Aerodynamic Coefficients:")
547 print(f" - Lift: {forces['lift']} N")
548 print(f" - C1: {coefficients['C1']} ")
549 print(f" - Drag: {forces['drag']} N")
550 print(f" - Cd: {coefficients['Cd']} ")
551 print(f" - Pitch Moment: {moments['pitch_moment']} N.m")
552 print(f" - Cm: {coefficients['Cm']} ")
553
554 # Create load distribution plots
555 components_loads = loads.functions.calc_load_distribution(
556     aircraft_force_grid=results_case3["aircraft_force_grid"],
557     aircraft_panel_grid=results_case3["original_aircraft_panel_grid"],
558     aircraft_gamma_grid=results_case1["aircraft_gamma_grid"],
559     attitude_vector=np.array([ALPHA, BETA, GAMMA]),
560     altitude=ALTITUDE,
561     speed=V_X,
562 )
563
564 for component in components_loads:
565     fig = plt.figure()
566
567     ax1 = fig.add_subplot(3, 1, 1)
568     ax1.set_title("Smith Wing - Case 003 - Lift Distribution")
569     ax1.set_xlabel("Span Position [m]")
570     ax1.set_ylabel("Lift [N]")
571     ax1.plot(component["y_values"], component["lift"])
572     ax1.grid()
573
574     ax2 = fig.add_subplot(3, 1, 2)
575     ax2.set_title("Smith Wing - Case 003 - C1 Distribution")
576     ax2.set_xlabel("Span Position [m]")
577     ax2.set_ylabel("C1")
578     ax2.plot(component["y_values"], component["C1"])
579     ax2.grid()
580
581     ax3 = fig.add_subplot(3, 1, 3)
582     ax3.set_title("Smith Wing - Case 003 - Drag Distribution")
583     ax3.set_xlabel("Span Position [m]")
584     ax3.set_ylabel("Drag [N]")
585     ax3.plot(component["y_values"], component["drag"])

```

```

586     ax3.grid()
587     plt.tight_layout()
588
589
590 a = results_case1["aircraft_force_grid"]
591 b = results_case2["aircraft_force_grid"]
592 c = results_case3["aircraft_force_grid"]
593 plt.show()

```

B.1.4 smith_wing_mesh_sensitivity.py

```

1 """
2 =====
3 CFD-Based Analysis of Nonlinear Aeroelastic Behavior of High-Aspect Ratio Wings
4
5 M. J. Smith, M. J. Patil, D. H. Hodges
6
7 Georgia Institute fo Technology, Atlanta
8
9 =====
10
11 Comparisson of the results obtained by in the paper above with those generated by the tool
12 developed
13 in this work
14 Author: João Paulo Monteiro Cruvinel da Costa
15 """
16
17 #
18 # IMPORTS
19
20 # Import python scientific libraries
21 import matplotlib.pyplot as plt
22 import numpy as np
23 import scipy as sc
24 import sys
25 import pickle
26
27 # Import code sub packages
28 from context import flyingcircus
29 from flyingcircus import aerodynamics as aero
30 from flyingcircus import aeroelasticity as aelast
31 from flyingcircus import control
32 from flyingcircus import flight_mechanics as flmec
33 from flyingcircus import geometry as geo
34 from flyingcircus import loads
35 from flyingcircus import structures as struct
36 from flyingcircus import visualization as vis
37
38 from smith_wing_data import smith_wing
39
40 #
41 # GRID CREATION
42
43 results_list = []
44 iteration_results_list = []
45
46 for i in range(1, 11):

```

```
47
48     # Number of panels and finite elements
49     N_CHORD_PANELS = i
50     N_SPAN_PANELS = int(16 / (2 * (1 / i)))
51     N_BEAM_ELEMENTS = 2 * N_SPAN_PANELS
52     CHORD_DISCRETIZATION = "linear"
53     SPAN_DISCRETIZATION = "linear"
54     TORSION_FUNCTION = "linear"
55     CONTROL_SURFACE_DEFLECTION_DICT = dict()
56
57     wing_grid_data = {
58         "n_chord_panels": N_CHORD_PANELS,
59         "n_span_panels_list": [N_SPAN_PANELS, N_SPAN_PANELS],
60         "n_beam_elements_list": [N_BEAM_ELEMENTS, N_BEAM_ELEMENTS],
61         "chord_discretization": CHORD_DISCRETIZATION,
62         "span_discretization_list": [SPAN_DISCRETIZATION, SPAN_DISCRETIZATION],
63         "torsion_function_list": [TORSION_FUNCTION, TORSION_FUNCTION],
64         "control_surface_deflection_dict": CONTROL_SURFACE_DEFLECTION_DICT,
65     }
66
67     smith_wing_grid_data = {
68         "macrosurfaces_grid_data": [wing_grid_data],
69         "beams_grid_data": None,
70     }
71     #
72     =====
73
74     # STRUCTURE DEFINITION
75
76     # Generate Constraints
77     wing_fixation = {
78         "component_identifier": "left_wing",
79         "fixation_point": "ROOT",
80         "dof_constraints": np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0]),
81     }
82
83     smith_wing_constraint_data = [wing_fixation]
84
85     #
86     =====
87
88     # AERODYNAMIC LOADS CALCULATION - CASE 2 - ALPHA 2° - Flexible Wing
89
90     # Flight Conditions definition
91
92     # Translation velocities
93     V_X = 25
94     V_Y = 0
95     V_Z = 0
96
97     # Rotation velocities
98     R_X = 0
99     R_Y = 0
100    R_Z = 0
101
102    # Aircraft Attitude in relation to the wind axis, in degrees
103    ALPHA = 2 # Pitch angle
104    BETA = 0 # Yaw angle
105    GAMMA = 0 # Roll angle
106
107    # Center of rotation, usually the aircraft CG position
108    CENTER_OF_ROTATION = np.array([0, 0, 0])
109
110    # Flight altitude, used to calculate atmospheric conditions, in meters
111    ALTITUDE = 20000
```

```

109     # Atmospheric turbulence, function that calculates the air speeds in relation to the ground,
110     # given
111     # a point coordinates
112
113     def ATM_TURBULENCE_FUNCTION(point_coordinates):
114         return np.zeros(3)
115
116     FLIGHT_CONDITIONS_DATA = {
117         "translation_velocity": np.array([V_X, V_Y, V_Z]),
118         "rotation_velocity": np.array([R_X, R_Y, R_Z]),
119         "attitude_angles_deg": np.array([ALPHA, BETA, GAMMA]),
120         "center_of_rotation": CENTER_OF_ROTATION,
121         "altitude": ALTITUDE,
122         "atm_turbulenc_function": ATM_TURBULENCE_FUNCTION,
123         "center_of_rotation": CENTER_OF_ROTATION,
124     }
125
126     SIMULATION_OPTIONS = {
127         "flexible_aircraft": True,
128         "status_messages": True,
129         "control_node_string": "left_wing-TIP",
130         "max_iterations": 100,
131         "bending_convergence_criteria": 0.01,
132         "torsion_convergence_criteria": 0.01,
133         "fem_prop_choice": "ROOT",
134         "interaction_algorithm": "closest",
135         "output_iteration_results": True,
136     }
137
138     results, iteration_results = aelast.functions.calculate_aircraft_loads(
139         aircraft_object=smith_wing,
140         aircraft_grid_data=smith_wing_grid_data,
141         aircraft_constraints_data=smith_wing_constraints_data,
142         flight_condition_data=FLIGHT_CONDITIONS_DATA,
143         simulation_options=SIMULATION_OPTIONS,
144         influence_coef_matrix=None,
145     )
146
147     results_list.append(results)
148     iteration_results_list.append(iteration_results)
149
150     print("====")
151     print(f"Chord Panels: {N_CHORD_PANELS}, Span panels: {N_SPAN_PANELS}")
152     print("Total force:")
153     print(results["aircraft_force_grid"][0].sum())
154
155     f = open("results\\smith_wing\\mesh_sensitivity.pckl", "wb")
156     pickle.dump([results_list, iteration_results_list], f)
157     f.close()

```

B.2 Hale Aircraft

B.2.1 hale_aircraft_data.py

```

1 """
2 =====
3 Nonlinear Aeroelasticity and Flight Dynamics of High-Altitude Long-Endurance Aircraft
4
5 Mayuresh J. Patil, Dewey H. Hodges, Carlos E. S. Cesnik

```

```
6
7 =====
8
9 Comparisson of the results obtained by in the paper above with those generated by the tool
   developed
10 in this work
11
12 Author: João Paulo Monteiro Cruvinel da Costa
13 """
14
15 #
=====
16 # IMPORTS
17
18 import numpy as np
19 import matplotlib.pyplot as plt
20
21 from pyquaternion import Quaternion
22
23 # Import code sub packages
24 from context import flyingcircus
25 from flyingcircus import geometry as geo
26 from flyingcircus import structures as struct
27 from flyingcircus import visualization as vis
28
29 #
=====
30 # GEOMETRY DEFINITION
31
32 # Wing section
33
34 # Aifoil name, only informative
35 airfoil = "airfoil"
36
37 # Material properties, all equal to one as Smith et al only provies the stiffness
   characteristics
38 # of the wing
39 MATERIAL = struct.objects.Material(
40     name="material",
41     density=0.75,
42     elasticity_modulus=1,
43     rigidity_modulus=1,
44     poisson_ratio=1,
45     yield_tensile_stress=1,
46     ultimate_tensile_stress=1,
47     yield_shear_stress=1,
48     ultimate_shear_stress=1,
49 )
50
51 # Wing section properties
52 SECTION = geo.objects.Section(
53     identifier=airfoil,
54     material=MATERIAL,
55     area=1e6,
56     Iyy=2e4,
57     Izz=4e6,
58     J=1e4,
59     shear_center=0.5,
60 )
61
62 #
-----
63 # WING AND TAIL DEFINITION
```

```
64
65 WING_INCIDENCE = 0
66
67 # Stub surface data
68 STUB_ROOT_CHORD = 1
69 STUB_TIP_CHORD = 1
70 SEMI_STUB_LENGTH = 2.5
71 STUB_SWEEP_ANGLE = 0
72 STUB_DIHEDRAL_ANGLE = 0
73 STUB_TIP_TORSION_ANGLE = 0
74
75 # Wing surface data
76 WING_ROOT_CHORD = 1
77 WING_TIP_CHORD = 1
78 SEMI_WING_LENGTH = 11.5
79 WING_SWEEP_ANGLE = 0
80 WING_DIHEDRAL_ANGLE = 0
81 WING_TIP_TORSION_ANGLE = 0
82
83 # Aileron Surface data
84 AILERON_ROOT_CHORD = 1
85 AILERON_TIP_CHORD = 1
86 AILERON_LENGTH = 2
87 AILERON_SWEEP_ANGLE = 0
88 AILERON_DIHEDRAL_ANGLE = 0
89 AILERON_TIP_TORSION_ANGLE = 0
90 AILERON_CONTROL_SURFACE_HINGE_POSITION = 0.8
91
92 # Elevator Surface Data
93 ELEVATOR_ROOT_CHORD = 0.5
94 ELEVATOR_TIP_CHORD = 0.5
95 SEMI_ELEVATOR_LENGTH = 2.5
96 ELEVATOR_SWEEP_ANGLE = 0
97 ELEVATOR_DIHEDRAL_ANGLE = 0
98 ELEVATOR_TIP_TORSION_ANGLE = 0
99 ELEVATOR_CONTROL_SURFACE_HINGE_POSITION = 0.6
100
101 # Rudder Surface Data
102 RUDDER_ROOT_CHORD = 0.5
103 RUDDER_TIP_CHORD = 0.5
104 RUDDER_LENGTH = 2.5
105 RUDDER_SWEEP_ANGLE = 0
106 RUDDER_DIHEDRAL_ANGLE = 90
107 RUDDER_TIP_TORSION_ANGLE = 0
108 RUDDER_CONTROL_SURFACE_HINGE_POSITION = 0.6
109
110 # Creation of the surface objects of the wing macrosurface
111 left_stub_surface = geo.objects.Surface(
112     identifier="left_stub",
113     root_chord=STUB_ROOT_CHORD,
114     root_section=SECTION,
115     tip_chord=STUB_TIP_CHORD,
116     tip_section=SECTION,
117     length=SEMI_STUB_LENGTH,
118     leading_edge_sweep_angle_deg=STUB_SWEEP_ANGLE,
119     dihedral_angle_deg=STUB_DIHEDRAL_ANGLE,
120     tip_torsion_angle_deg=STUB_TIP_TORSION_ANGLE,
121     control_surface_hinge_position=None,
122 )
123
124 right_stub_surface = geo.objects.Surface(
125     identifier="right_stub",
126     root_chord=STUB_ROOT_CHORD,
127     root_section=SECTION,
128     tip_chord=STUB_TIP_CHORD,
129     tip_section=SECTION,
130     length=SEMI_STUB_LENGTH,
```

```
131     leading_edge_sweep_angle_deg=STUB_SWEEP_ANGLE ,
132     dihedral_angle_deg=STUB_DIHEDRAL_ANGLE ,
133     tip_torsion_angle_deg=STUB_TIP_TORSION_ANGLE ,
134     control_surface_hinge_position=None ,
135 )
136
137 left_wing_surface = geo.objects.Surface(
138     identifier="left_wing",
139     root_chord=WING_ROOT_CHORD ,
140     root_section=SECTION ,
141     tip_chord=WING_TIP_CHORD ,
142     tip_section=SECTION ,
143     length=SEMI_WING_LENGTH ,
144     leading_edge_sweep_angle_deg=WING_SWEEP_ANGLE ,
145     dihedral_angle_deg=WING_DIHEDRAL_ANGLE ,
146     tip_torsion_angle_deg=WING_TIP_TORSION_ANGLE ,
147     control_surface_hinge_position=None ,
148 )
149
150 right_wing_surface = geo.objects.Surface(
151     identifier="right_wing",
152     root_chord=WING_ROOT_CHORD ,
153     root_section=SECTION ,
154     tip_chord=WING_TIP_CHORD ,
155     tip_section=SECTION ,
156     length=SEMI_WING_LENGTH ,
157     leading_edge_sweep_angle_deg=WING_SWEEP_ANGLE ,
158     dihedral_angle_deg=WING_DIHEDRAL_ANGLE ,
159     tip_torsion_angle_deg=WING_TIP_TORSION_ANGLE ,
160     control_surface_hinge_position=None ,
161 )
162
163 left_aileron_surface = geo.objects.Surface(
164     identifier="left_aileron",
165     root_chord=AILERON_ROOT_CHORD ,
166     root_section=SECTION ,
167     tip_chord=AILERON_TIP_CHORD ,
168     tip_section=SECTION ,
169     length=AILERON_LENGTH ,
170     leading_edge_sweep_angle_deg=AILERON_SWEEP_ANGLE ,
171     dihedral_angle_deg=AILERON_DIHEDRAL_ANGLE ,
172     tip_torsion_angle_deg=AILERON_TIP_TORSION_ANGLE ,
173     control_surface_hinge_position=AILERON_CONTROL_SURFACE_HINGE_POSITION ,
174 )
175
176 right_aileron_surface = geo.objects.Surface(
177     identifier="right_aileron",
178     root_chord=AILERON_ROOT_CHORD ,
179     root_section=SECTION ,
180     tip_chord=AILERON_TIP_CHORD ,
181     tip_section=SECTION ,
182     length=AILERON_LENGTH ,
183     leading_edge_sweep_angle_deg=AILERON_SWEEP_ANGLE ,
184     dihedral_angle_deg=AILERON_DIHEDRAL_ANGLE ,
185     tip_torsion_angle_deg=AILERON_TIP_TORSION_ANGLE ,
186     control_surface_hinge_position=AILERON_CONTROL_SURFACE_HINGE_POSITION ,
187 )
188
189 # Creation of the surface objects of the horizontal tail macrosurface
190 left_elevator_surface = geo.objects.Surface(
191     identifier="left_elevator",
192     root_chord=ELEVATOR_ROOT_CHORD ,
193     root_section=SECTION ,
194     tip_chord=ELEVATOR_TIP_CHORD ,
195     tip_section=SECTION ,
196     length=SEMI_ELEVATOR_LENGTH ,
197     leading_edge_sweep_angle_deg=ELEVATOR_SWEEP_ANGLE ,
```

```
198     dihedral_angle_deg=ELEVATOR_DIHEDRAL_ANGLE,
199     tip_torsion_angle_deg=ELEVATOR_TIP_TORSION_ANGLE,
200     control_surface_hinge_position=ELEVATOR_CONTROL_SURFACE_HINGE_POSITION,
201 )
202
203 right_elevator_surface = geo.objects.Surface(
204     identifier="right_elevator",
205     root_chord=ELEVATOR_ROOT_CHORD,
206     root_section=SECTION,
207     tip_chord=ELEVATOR_TIP_CHORD,
208     tip_section=SECTION,
209     length=SEMI_ELEVATOR_LENGTH,
210     leading_edge_sweep_angle_deg=ELEVATOR_SWEEP_ANGLE,
211     dihedral_angle_deg=ELEVATOR_DIHEDRAL_ANGLE,
212     tip_torsion_angle_deg=ELEVATOR_TIP_TORSION_ANGLE,
213     control_surface_hinge_position=ELEVATOR_CONTROL_SURFACE_HINGE_POSITION,
214 )
215
216 # Creation of the surface objects of the vertical tail macrosurface
217 rudder_surface = geo.objects.Surface(
218     identifier="rudder",
219     root_chord=RUDDER_ROOT_CHORD,
220     root_section=SECTION,
221     tip_chord=RUDDER_TIP_CHORD,
222     tip_section=SECTION,
223     length=RUDDER_LENGTH,
224     leading_edge_sweep_angle_deg=RUDDER_SWEEP_ANGLE,
225     dihedral_angle_deg=RUDDER_DIHEDRAL_ANGLE,
226     tip_torsion_angle_deg=RUDDER_TIP_TORSION_ANGLE,
227     control_surface_hinge_position=RUDDER_CONTROL_SURFACE_HINGE_POSITION,
228 )
229
230 # Creation of the wing macrosurface
231 wing = geo.objects.MacroSurface(
232     position=np.array([0, 0, 0]),
233     incidence=WING_INCIDENCE,
234     surface_list=[
235         left_aileron_surface,
236         left_wing_surface,
237         left_stub_surface,
238         right_stub_surface,
239         right_wing_surface,
240         right_aileron_surface,
241     ],
242     symmetry_plane="XZ",
243     torsion_center=0.5,
244 )
245
246 # Creation of the horizontal tail macrosurface
247 htail = geo.objects.MacroSurface(
248     position=np.array([10.75, 0, 0]),
249     incidence=WING_INCIDENCE,
250     surface_list=[
251         left_elevator_surface,
252         right_elevator_surface,
253     ],
254     symmetry_plane="XZ",
255     torsion_center=0.5,
256 )
257
258 # Creation of the vertical tail macrosurface
259 vtail = geo.objects.MacroSurface(
260     position=np.array([10.75, 0, 0]),
261     incidence=WING_INCIDENCE,
262     surface_list=[
263         rudder_surface,
264     ],
```

```
265     symmetry_plane=None,
266     torsion_center=0.5,
267 )
268
269 aircraft_macrosurfaces = [wing, htail, vtail]
270
271 #
-----  

272 # FUSELAGE AND TAIL BOOM DEFINITION
273
274 POINT_1 = np.array([0.5, 0.0, 0.0])
275 POINT_2 = np.array([1.0, 0.0, 0.0])
276 POINT_3 = np.array([11.0, 0.0, 0.0])
277
278 beam_property = struct.objects.ElementProperty(section=SECTION, material=MATERIAL)
279
280 fuselage = geo.objects.Beam(
281     identifier="fuselage",
282     root_point=POINT_1,
283     tip_point=POINT_2,
284     orientation_vector=np.array([0.0, 1.0, 0.0]),
285     ElementProperty=beam_property,
286 )
287
288 tail_boom = geo.objects.Beam(
289     identifier="tail_boom",
290     root_point=POINT_2,
291     tip_point=POINT_3,
292     orientation_vector=np.array([0.0, 1.0, 0.0]),
293     ElementProperty=beam_property,
294 )
295
296 aircraft_beams = [fuselage, tail_boom]
297
298 #
-----  

299 # AIRCRAFT CG DEFINITION
300
301 AIRCRAFT_MASS = 4000
302 CG_POSITION = POINT_2
303 IXX = 1
304 IYY = 1
305 IZZ = 1
306 IXY = 1
307 IXZ = 1
308 IYZ = 1
309
310 aircraft_cg = geo.objects.MaterialPoint(
311     identifier="aircraft_cg",
312     orientation_quaternion=Quaternion(),
313     mass=AIRCRAFT_MASS,
314     position=CG_POSITION,
315     Ixx=IXX,
316     Iyy=IYY,
317     Izz=IZZ,
318     Ixy=IXY,
319     Ixz=IXZ,
320     Iyz=IYZ,
321 )
322
323 #
-----  

324 # AIRCRAFT STRUCTURE CONNECTIONS
325
```

```

326 wing_to_fuselage = struct.objects.Connection(
327     left_stub_surface, "ROOT", fuselage, "ROOT"
328 )
329
330 fuselage_to_tail_boom = struct.objects.Connection(fuselage, "TIP", tail_boom, "ROOT")
331
332
333 tail_boom_to_htail = struct.objects.Connection(
334     tail_boom, "TIP", left_elevator_surface, "ROOT"
335 )
336
337 tail_boom_to_vtail = struct.objects.Connection(
338     tail_boom, "TIP", rudder_surface, "ROOT"
339 )
340
341 htail_to_vtail = struct.objects.Connection(
342     left_elevator_surface, "ROOT", rudder_surface, "ROOT"
343 )
344
345 aircraft_struct_connections = [
346     wing_to_fuselage,
347     fuselage_to_tail_boom,
348     tail_boom_to_htail,
349     tail_boom_to_vtail,
350     htail_to_vtail,
351 ]
352
353 #
-----
```

```

354
355 # Aircraft definition
356
357 AIRCRAFT_NAME = "hale aircraft"
358 AIRCRAFT_MACROSURFACES = aircraft_macrosurfaces
359 AIRCRAFT_BEAMS = aircraft_beams
360 AIRCRAFT_INERTIAL_PROPERTIES = aircraft_cg
361 AIRCRAFT_STRUCT_CONNECTIONS = aircraft_struct_connections
362
363 hale_aircraft = geo.objects.Aircraft(
364     name=AIRCRAFT_NAME,
365     macrosurfaces=AIRCRAFT_MACROSURFACES,
366     beams=AIRCRAFT_BEAMS,
367     inertial_properties=AIRCRAFT_INERTIAL_PROPERTIES,
368     connections=AIRCRAFT_STRUCT_CONNECTIONS,
369     ref_area=32,
370     mean_aero_chord=1
371 )
```

B.2.2 hale_aircraft_simulation.py

```

1 """
2 =====
3 Comparisson between results found in the literature and those obtained by using Flying Circus.
4 Definition of the simulation data
5
6 Author: João Paulo Monteiro Cruvinel da Costa
7
8 Literature results:
9
10 NACA Technical Note No.1270 - EXPERIMENTAL AND CALCULATED CHARACTERISTICS OF SEVERALNACA 44-
11 SERIES
12 WINGS WITH ASPECT RATIOS OF 8, 10, AND 12 AND TAPER RATIOS OF 2.5 AND 3.5
```

```
12
13 Authors: Robert H. Neely, Thomas V. Bollech, Gertrude C. Westrick, and Robert R. Graham
14
15 Langley Memorial Aeronautical Laboratory
16 Langley Field, Va.
17
18 Washington, May 1947
19 =====
20 """
21
22 #
=====
23 # IMPORTS
24
25 # Import python scientific libraries
26 import matplotlib.pyplot as plt
27 import numpy as np
28 import scipy as sc
29 import sys
30 import pickle
31
32 # Import code sub packages
33 from context import flyingcircus
34 from flyingcircus import aerodynamics as aero
35 from flyingcircus import aeroelasticity as aelast
36 from flyingcircus import control
37 from flyingcircus import flight_mechanics as flmec
38 from flyingcircus import geometry as geo
39 from flyingcircus import loads
40 from flyingcircus import structures as struct
41 from flyingcircus import visualization as vis
42
43 #
=====
44 # print()
45 # print("===== ")
46 # print("= VALIDATION OF AEROELASTIC CALCULATION      =")
47 # print("= VALIDATION CASE: CFD-Based Analysis of Nonlinear      =")
48 # print("= Aeroelastic Behavior of High-Aspect Ratio Wings      =")
49 # print("= AUTHORS: M. J. Smith, M. J. Patil, D. H. Hodges      =")
50 # print("===== ")
51 #
=====
52 # GEOMETRY DEFINITION
53
54 print("# Importing geometric data...")
55 from hale_aircraft_data import hale_aircraft
56
57 # WING GRID DATA
58 N_CHORD_PANELS = 5
59
60 STUB_N_SPAN_PANELS = 5
61 STUB_N_BEAM_ELEMENTS = 2 * STUB_N_SPAN_PANELS
62
63 WING_N_SPAN_PANELS = 23
64 WING_N_BEAM_ELEMENTS = 2 * WING_N_SPAN_PANELS
65
66 AILERON_N_SPAN_PANELS = 4
67 AILERON_N_BEAM_ELEMENTS = 2 * AILERON_N_SPAN_PANELS
68
69 CHORD_DISCRETIZATION = "linear"
70 SPAN_DISCRETIZATION = "linear"
71 TORSION_FUNCTION = "linear"
```

```

72 CONTROL_SURFACE_DEFLECTION_DICT = {"left_aileron": 0, "right_aileron": 0}
73
74 wing_grid_data = {
75     "n_chord_panels": N_CHORD_PANELS,
76     "n_span_panels_list": [
77         AILERON_N_SPAN_PANELS,
78         WING_N_SPAN_PANELS,
79         STUB_N_SPAN_PANELS,
80         STUB_N_SPAN_PANELS,
81         WING_N_SPAN_PANELS,
82         AILERON_N_SPAN_PANELS,
83     ],
84     "n_beam_elements_list": [
85         AILERON_N_BEAM_ELEMENTS,
86         WING_N_BEAM_ELEMENTS,
87         STUB_N_BEAM_ELEMENTS,
88         STUB_N_BEAM_ELEMENTS,
89         WING_N_BEAM_ELEMENTS,
90         AILERON_N_BEAM_ELEMENTS,
91     ],
92     "chord_discretization": CHORD_DISCRETIZATION,
93     "span_discretization_list": [
94         SPAN_DISCRETIZATION,
95         SPAN_DISCRETIZATION,
96         SPAN_DISCRETIZATION,
97         SPAN_DISCRETIZATION,
98         SPAN_DISCRETIZATION,
99         SPAN_DISCRETIZATION,
100    ],
101    "torsion_function_list": [
102        TORSION_FUNCTION,
103        TORSION_FUNCTION,
104        TORSION_FUNCTION,
105        TORSION_FUNCTION,
106        TORSION_FUNCTION,
107        TORSION_FUNCTION,
108    ],
109    "control_surface_deflection_dict": CONTROL_SURFACE_DEFLECTION_DICT,
110 }
111
112 #
-----
```

```

113
114 # HTAIL GRID DATA
115 HTAIL_N_CHORD_PANELS = 5
116
117 HTAIL_N_SPAN_PANELS = 5
118 HTAIL_N_BEAM_ELEMENTS = 2 * HTAIL_N_SPAN_PANELS
119
120 HTAIL_CHORD_DISCRETIZATION = "linear"
121 HTAIL_SPAN_DISCRETIZATION = "linear"
122 HTAIL_TORSION_FUNCTION = "linear"
123 HTAIL_CONTROL_SURFACE_DEFLECTION_DICT = {"left_elevator": 0, "right_elevator": 0}
124
125 htail_grid_data = {
126     "n_chord_panels": HTAIL_N_CHORD_PANELS,
127     "n_span_panels_list": [
128         HTAIL_N_SPAN_PANELS,
129         HTAIL_N_SPAN_PANELS,
130     ],
131     "n_beam_elements_list": [
132         HTAIL_N_BEAM_ELEMENTS,
133         HTAIL_N_BEAM_ELEMENTS,
134     ],
135     "chord_discretization": HTAIL_CHORD_DISCRETIZATION,
136     "span_discretization_list": [
```

```

137         HTAIL_SPAN_DISCRETIZATION,
138         HTAIL_SPAN_DISCRETIZATION,
139     ],
140     "torsion_function_list": [
141         HTAIL_TORSION_FUNCTION,
142         HTAIL_TORSION_FUNCTION,
143     ],
144     "control_surface_deflection_dict": HTAIL_CONTROL_SURFACE_DEFLECTION_DICT,
145 }
146
147 #
-----
```

```

148
149 # VTAIL GRID DATA
150 VTAIL_N_CHORD_PANELS = 5
151
152 VTAIL_N_SPAN_PANELS = 5
153 VTAIL_N_BEAM_ELEMENTS = 2 * VTAIL_N_SPAN_PANELS
154
155 VTAIL_CHORD_DISCRETIZATION = "linear"
156 VTAIL_SPAN_DISCRETIZATION = "linear"
157 VTAIL_TORSION_FUNCTION = "linear"
158 VTAIL_CONTROL_SURFACE_DEFLECTION_DICT = {"rudder": 0}
159
160 vtail_grid_data = {
161     "n_chord_panels": VTAIL_N_CHORD_PANELS,
162     "n_span_panels_list": [
163         VTAIL_N_SPAN_PANELS,
164     ],
165     "n_beam_elements_list": [
166         VTAIL_N_BEAM_ELEMENTS,
167     ],
168     "chord_discretization": VTAIL_CHORD_DISCRETIZATION,
169     "span_discretization_list": [
170         VTAIL_SPAN_DISCRETIZATION,
171     ],
172     "torsion_function_list": [
173         HTAIL_TORSION_FUNCTION,
174     ],
175     "control_surface_deflection_dict": VTAIL_CONTROL_SURFACE_DEFLECTION_DICT,
176 }
177 #
-----
```

```

178 # FUSELAGE AND TAIL BOOM GRID DATA
179
180 fuselage_grid_data = {"n_elements": 2}
181
182 tail_boom_grid_data = {"n_elements": 40}
183
184 #
-----
```

```

185 # HALE AIRCRAFT GRID DATA
186
187 hale_aircraft_grid_data = {
188     "macrosurfaces_grid_data": [wing_grid_data, htail_grid_data, vtail_grid_data],
189     "beams_grid_data": [fuselage_grid_data, tail_boom_grid_data],
190 }
191
192 #
=====
```

```

193 # GRID CREATION
194
195 # Creation of the smith wing grids
```

```
196 hale_aircraft_grids = aelast.functions.generate_aircraft_grids(
197 #     aircraft_object=hale_aircraft, aircraft_grid_data=hale_aircraft_grid_data
198 #)
199
200 #
=====

201 # STRUCTURE DEFINITION
202
203 # Create wing finite elements
204
205 hale_aircraft_fem_elements = struct.fem.generate_aircraft_fem_elements(
206 #     aircraft=hale_aircraft, aircraft_grids=hale_aircraft_grids, prop_choice="ROOT"
207 #)
208
209 #grids_ax, grids_fig = vis.plot_3D2.generate_aircraft_grids_plot(
210 #     hale_aircraft_grids["macrosurfaces_aero_grids"],
211 #     hale_aircraft_fem_elements,
212 #     title="Hale Aircraft - Original vs Deformed Grids",
213 #     ax=None,
214 #     show_origin=True,
215 #     show_nodes=False,
216 #     line_color="k",
217 #     alpha=0.5,
218 #)
219 #
220 #plt.show()
221
222 # Generate Constraints
223 cg_fixation = {
224     "component_identifier": "fuselage",
225     "fixation_point": "TIP",
226     "dof_constraints": np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0]),
227 }
228
229 hale_aircraft_constraint_data = [cg_fixation]
230
231 #hale_aircraft_constraints = aelast.functions.generate_aircraft_constraints(
232 #     aircraft=hale_aircraft,
233 #     aircraft_grids=hale_aircraft_grids,
234 #     constraints_data_list=hale_aircraft_constraint_data,
235 #)
236
237 #
=====

238 # AERODYNAMIC LOADS CALCULATION - CASE 1 - ALPHA 22
239 print()
240 print("# CASE 001:")
241 print(f"    - Altitude: 20000m")
242 print(f"    - True Airspeed: 25m/s")
243 print(f"    - Alpha: 22")
244 print(f"    - Flexible Wing")
245 print()
246
247 # Flight Conditions definition
248
249 # Translation velocities
250 V_X = 25
251 V_Y = 0
252 V_Z = 0
253
254 # Rotation velocities
255 R_X = 0
256 R_Y = 0
257 R_Z = 0
258
```

```
259 # Aircraft Attitude in relation to the wind axis, in degrees
260 ALPHA = 2 # Pitch angle
261 BETA = 0 # Yaw angle
262 GAMMA = 0 # Roll angle
263
264 # Center of rotation, usually the aircraft CG position
265 CENTER_OF_ROTATION = hale_aircraft.inertial_properties.position
266
267 # Flight altitude, used to calculate atmospheric conditions, in meters
268 ALTITUDE = 20000
269
270 # Atmospheric turbulence, function that calculates the air speeds in relation to the ground,
271 # given
272 # a point coordinates
273
274 def ATM_TURBULENCE_FUNCTION(point_coordinates):
275
276     return np.zeros(3)
277
278 rig_results = []
279 flex_results = []
280 flex_iteration_results = []
281
282 #for i in range(1, 6):
283 for i in [1, 2, 3, 3.5, 4, 4.5, 5]:
284
285     ALPHA = i
286
287     FLIGHT_CONDITIONS_DATA = {
288         "translation_velocity": np.array([V_X, V_Y, V_Z]),
289         "rotation_velocity": np.array([R_X, R_Y, R_Z]),
290         "attitude_angles_deg": np.array([ALPHA, BETA, GAMMA]),
291         "center_of_rotation": CENTER_OF_ROTATION,
292         "altitude": ALTITUDE,
293         "atm_turbulence_function": ATM_TURBULENCE_FUNCTION,
294         "center_of_rotation": CENTER_OF_ROTATION,
295     }
296
297     SIMULATION_OPTIONS = {
298         "flexible_aircraft": True,
299         "status_messages": True,
300         "control_node_string": "left_elevator-TIP",
301         "max_iterations": 100,
302         "bending_convergence_criteria": 0.01,
303         "torsion_convergence_criteria": 0.01,
304         "fem_prop_choice": "ROOT",
305         "interaction_algorithm": "closest",
306         "output_iteration_results": True,
307     }
308
309     results, iteration_results = aelast.functions.calculate_aircraft_loads(
310         aircraft_object=hale_aircraft,
311         aircraft_grid_data=hale_aircraft_grid_data,
312         aircraft_constraints_data=hale_aircraft_constraints_data,
313         flight_condition_data=FLIGHT_CONDITIONS_DATA,
314         simulation_options=SIMULATION_OPTIONS,
315         influence_coef_matrix=None,
316     )
317
318     flex_results.append(results)
319     flex_iteration_results.append(iteration_results)
320
321     SIMULATION_OPTIONS = {
322         "flexible_aircraft": False,
323         "status_messages": True,
324         "control_node_string": "left_elevator-TIP",
```

```

325         "max_iterations": 100,
326         "bending_convergence_criteria": 0.01,
327         "torsion_convergence_criteria": 0.01,
328         "fem_prop_choice": "ROOT",
329         "interaction_algorithm": "closest",
330         "output_iteration_results": True,
331     }
332
333     results = aelast.functions.calculate_aircraft_loads(
334         aircraft_object=hale_aircraft,
335         aircraft_grid_data=hale_aircraft_grid_data,
336         aircraft_constraints_data=hale_aircraft_constraints_data,
337         flight_condition_data=FLIGHT_CONDITIONS_DATA,
338         simulation_options=SIMULATION_OPTIONS,
339         influence_coef_matrix=None,
340     )
341
342     rig_results.append(results)
343
344 f = open("results\\hale_aircraft\\hale_aircraft_sim.pkl", "wb")
345 pickle.dump([rig_results, flex_results, flex_iteration_results], f)
346 f.close()

```

B.2.3 hale_aircraft_results.py

```

1 """
2 =====
3 CFD-Based Analysis of Nonlinear Aeroelastic Behavior of High-Aspect Ratio Wings
4
5 M. J. Smith, M. J. Patil, D. H. Hodges
6
7 Georgia Institute fo Technology, Atlanta
8
9 =====
10
11 Comparisson of the results obtained by in the paper above with those generated by the tool
12 developed
13 in this work
14
15 """
16
17 #
18 # IMPORTS
19
20 # Import python scientific libraries
21 import matplotlib.pyplot as plt
22 import numpy as np
23 import scipy as sc
24 import sys
25
26 # Import code sub packages
27 from context import flyingcircus
28 from flyingcircus import aerodynamics as aero
29 from flyingcircus import aeroelasticity as aelast
30 from flyingcircus import control
31 from flyingcircus import flight_mechanics as flmec
32 from flyingcircus import geometry as geo
33 from flyingcircus import loads
34 from flyingcircus import structures as struct

```

```
35 from flyingcircus import visualization as vis
36
37 #
=====

38 print()
39 print("====")
40 print("= VALIDATION OF AEROELASTIC CALCULATION      =")
41 print("= VALIDATION CASE: CFD-Based Analysis of Nonlinear      =")
42 print("= Aeroelastic Behavior of High-Aspect Ratio Wings      =")
43 print("= AUTHORS: M. J. Smith, M. J. Patil, D. H. Hodges      =")
44 print("====")
45 #
=====

46 # EXECUTE CALCULATION
47
48 from hale_aircraft_data import hale_aircraft
49
50 #from hale_aircraft_simulation import results, iteration_results
51
52 import pickle
53
54 #f = open("results\\hale_aircraft\\results\\results_no_deflection.pckl", 'wb')
55 #pickle.dump([results, iteration_results], f)
56 #f.close()
57
58 f = open("results\\hale_aircraft\\hale_aircraft_sim.pckl", "rb")
59 rig_results, flex_results, flex_iteration_results = pickle.load(f)
60 f.close()
61
62 # Draw Aircraft
63 aircraft_ax, aircraft_fig = vis.plot_3D2.generate_aircraft_plot(
64     hale_aircraft, title="Hale Aircraft"
65 )
66
67 results = flex_results[0]
68
69 #
=====

70 # PROCESSING RESULTS
71
72 # CASE 001:
73 #     - Alpha: 2°
74 #     - Speed: 25 m/s
75 #     - Altitude: 20000 m
76 #     - Flexible
77
78 # Generate Original vs Deformed Grid Plot
79
80 # Draw original grids
81 grids_ax, grids_fig = vis.plot_3D2.generate_aircraft_grids_plot(
82     results["aircraft_original_grids"]["macrosurfaces_aero_grids"],
83     results["aircraft_struct_fem_elements"],
84     title="Hale Aircraft - Original vs Deformed Grids",
85     ax=None,
86     show_origin=True,
87     show_nodes=False,
88     line_color="k",
89     alpha=0.5,
90 )
91
92
93 # Draw deformed Grids
94 grids_ax, grids_fig = vis.plot_3D2.generate_deformed_aircraft_grids_plot(
95     results["aircraft_deformed_macrosurfaces_aero_grids"],
```

```

96     results["aircraft_struct_fem_elements"],
97     results["aircraft_struct_deformations"],
98     ax=grids_ax,
99     fig=grids_fig,
100    show_origin=True,
101    show_nodes=False,
102    line_color="r",
103    alpha=1,
104 )
105
106 # Calculate Loads on each of the aerodynamic panels
107 aircraft_panel_loads = loads.functions.calculate_aircraft_panel_loads(
108     results["original_aircraft_panel_grid"], results["aircraft_force_grid"]
109 )
110
111 results_ax, results_fig = vis.plot_3D2.generate_results_plot(
112     aircraft_deformed_macrosurfaces_aero_grids=results[
113         "aircraft_deformed_macrosurfaces_aero_grids"],
114     aircraft_panel_loads=aircraft_panel_loads,
115     aircraft_struct_fem_elements=results["aircraft_struct_fem_elements"],
116     aircraft_struct_deformations=results["aircraft_struct_deformations"],
117     results_string="delta_p_grid",
118     title="Hale Aircraft - Delta Pressure [Pa]",
119     colorbar_label="Delta Pressure [Pa]",
120     ax=None,
121     fig=None,
122     show_origin=True,
123     colormap="coolwarm",
124 )
125 # Deformation plot
126
127 deformation_table = aelast.functions.calculate_deformation_table(
128     results["aircraft_original_grids"],
129     results["aircraft_struct_deformations"],
130 )
131
132 # sort nodes by desired column, in this case the Y coordinate
133
134 nodes = deformation_table["aircraft_macrosurfaces_deformed_nodes"][0]
135 nodes = nodes[nodes[:, 1].argsort()]
136
137 # Plot Bending
138 fig, ax = plt.subplots()
139 ax.plot(nodes[:, 1], nodes[:, 2])
140 ax.grid()
141 ax.set_title("Hale Aircraft - Bending")
142 ax.set_ylabel("Bending [m]")
143 ax.set_xlabel("Span [m]")
144
145 # Plot Torsion
146 fig, ax = plt.subplots()
147 ax.plot(nodes[:, 1], np.degrees(nodes[:, 4]))
148 ax.grid()
149 ax.set_title("Hale Aircraft - Torsion")
150 ax.set_ylabel("Torsion [degrees]")
151 ax.set_xlabel("Span [m]")
152
153 interest_point = hale_aircraft.inertial_properties.position
154
155 # Aerodynamic forces in the aircraft coordinate system
156 total_cg_aero_force, total_cg_aero_moment, component_cg_aero_loads = loads.functions.
157     calc_aero_loads_at_point(
158     interest_point,
159     results["aircraft_force_grid"],
160     results["aircraft_deformed_macrosurfaces_aero_panels"],
161 )

```

```

161
162 print()
163 print("#####")
164 print("#          HALE AIRCRAFT RESULTS      #")
165 print("#####")
166 print()
167 print(f"# Total loads at aircraft CG:")
168 print(f"    FX: {total_cg_aero_force[0]} N")
169 print(f"    FY: {total_cg_aero_force[1]} N")
170 print(f"    FZ: {total_cg_aero_force[2]} N")
171 print(f"    RX: {total_cg_aero_moment[0]} N.m")
172 print(f"    RY: {total_cg_aero_moment[1]} N.m")
173 print(f"    RZ: {total_cg_aero_moment[2]} N.m")
174
175 V_X = 25
176 V_Y = 0
177 V_Z = 0
178
179 # Rotation velocities
180 R_X = 0
181 R_Y = 0
182 R_Z = 0
183
184 # Aircraft Attitude in relation to the wind axis, in degrees
185 ALPHA = 2 # Pitch angle
186 BETA = 0 # Yaw angle
187 GAMMA = 0 # Roll angle
188
189 # Center of rotation, usually the aircraft CG position
190 CENTER_OF_ROTATION = hale_aircraft.inertial_properties.position
191
192 # Flight altitude, used to calculate atmospheric conditions, in meters
193 ALTITUDE = 20000
194
195 forces, moments, coefficients = loads.functions.calc_lift_drag(
196     aircraft=hale_aircraft,
197     point=interest_point,
198     speed=V_X,
199     altitude=ALTITUDE,
200     attitude_vector=np.array([ALPHA, BETA, GAMMA]),
201     aircraft_force_grid=results["aircraft_force_grid"],
202     aircraft_panel_grid=results["aircraft_deformed_macrosurfaces_aero_panels"],
203 )
204
205 print()
206 print("# Aerodynamic Coeffients:")
207 print(f"    - Lift: {forces['lift']} N")
208 print(f"    - Cl: {coefficients['Cl']} ")
209 print(f"    - Drag: {forces['drag']} N")
210 print(f"    - Cd: {coefficients['Cd']} ")
211 print(f"    - Pitch Moment: {moments['pitch_moment']} N.m")
212 print(f"    - Cm: {coefficients['Cm']} ")
213
214 # Create load distribution plots
215 components_loads = loads.functions.calc_load_distribution(
216     aircraft_force_grid=results["aircraft_force_grid"],
217     aircraft_panel_grid=results["original_aircraft_panel_grid"],
218     aircraft_gamma_grid=results["aircraft_gamma_grid"],
219     attitude_vector=np.array([ALPHA, BETA, GAMMA]),
220     altitude=ALTITUDE,
221     speed=V_X,
222 )
223
224 for component in components_loads:
225     fig = plt.figure()
226
227     ax1 = fig.add_subplot(3, 1, 1)

```

```

228     ax1.set_title(f"Hale Aircraft - Lift Distribution")
229     ax1.set_xlabel("Span Position [m]")
230     ax1.set_ylabel("Lift [N]")
231     ax1.plot(component["y_values"], component["lift"])
232     ax1.grid()
233
234     ax2 = fig.add_subplot(3, 1, 2)
235     ax2.set_title(f"Hale Aircraft - Cl Distribution")
236     ax2.set_xlabel("Span Position [m]")
237     ax2.set_ylabel("Cl")
238     ax2.plot(component["y_values"], component["Cl"])
239     ax2.grid()
240
241     ax3 = fig.add_subplot(3, 1, 3)
242     ax3.set_title(f"Hale Aircraft - Drag Distribution")
243     ax3.set_xlabel("Span Position [m]")
244     ax3.set_ylabel("Drag [N]")
245     ax3.plot(component["y_values"], component["drag"])
246     ax3.grid()
247     plt.tight_layout()
248
249 plt.show()
250
251 input()

```

B.2.4 hale_aircraft_results_processing.py

```

1 """
2 =====
3 CFD-Based Analysis of Nonlinear Aeroelastic Behavior of High-Aspect Ratio Wings
4
5 M. J. Smith, M. J. Patil, D. H. Hodges
6
7 Georgia Institute fo Technology, Atlanta
8
9 =====
10
11 Comparisson of the results obtained by in the paper above with those generated by the tool
12 developed
13 in this work
14
15 Author: João Paulo Monteiro Cruvinel da Costa
16 """
17 #
18 # IMPORTS
19
20 # Import python scientific libraries
21 import matplotlib.pyplot as plt
22 import numpy as np
23 import scipy as sc
24 import sys
25 import pickle
26
27 # Import code sub packages
28 from context import flyingcircus
29 from flyingcircus import aerodynamics as aero
30 from flyingcircus import aeroelasticity as aelast
31 from flyingcircus import control
32 from flyingcircus import flight_mechanics as flmec

```

```
33 from flyingcircus import geometry as geo
34 from flyingcircus import loads
35 from flyingcircus import structures as struct
36 from flyingcircus import visualization as vis
37 from flyingcircus import mathematics as m
38
39
40 from hale_aircraft_data import hale_aircraft
41
42 #f = open("results\\hale_aircraft\\hale_aircraft_sim.pckl", "rb")
43 #rig_results, flex_results, flex_iteration_results = pickle.load(f)
44 #f.close()
45
46 f = open("results\\hale_aircraft\\hale_aircraft_final_results.pckl", "rb")
47 rig_results, flex_results = pickle.load(f)
48 f.close()
49
50
51 #f = open("results\\hale_aircraft\\hale_aircraft_final_results.pckl", "wb")
52 #pickle.dump([rig_results, flex_results], f)
53 #f.close()
54
55 # Draw Aircraft
56 aircraft_ax, aircraft_fig = vis.plot_3D2.generate_aircraft_plot(
57     hale_aircraft, title="Aeronave Hale",
58 )
59
60 #plt.show()
61
62 #
=====

63 # PROCESSING RESULTS
64
65 # CASE 001:
66 # - Alpha: 2ž
67 # - Speed: 25 m/s
68 # - Altitude: 20000 m
69 # - Flexible
70
71 # Generate Original vs Deformed Grid Plot
72
73 alphas = [1, 2, 3, 3.5, 4, 4.5, 5]
74
75 color_pallet = [
76     "tab:blue",
77     "tab:orange",
78     "tab:green",
79     "tab:red",
80     "tab:purple",
81     "tab:brown",
82     "tab:pink",
83     "tab:gray",
84     "tab:olive",
85     "tab:cyan",
86 ]
87
88
89 # Draw original grids
90 grids_ax, grids_fig = vis.plot_3D2.generate_aircraft_grids_plot(
91     flex_results[0]["aircraft_original_grids"]["macrosurfaces_aero_grids"],
92     flex_results[0]["aircraft_struct_fem_elements"],
93     title="Aeronave HALE Malhas Original e Deformadas",
94     ax=None,
95     show_origin=True,
96     show_nodes=False,
97     line_color="tab:blue",
```

```
98     alpha=0.5,
99 )
100
101 for i, results_m in enumerate(flex_results):
102
103     # Draw deformed Grids
104     grids_ax, grids_fig = vis.plot_3D2.generate_deformed_aircraft_grids_plot(
105         results_m["aircraft_deformed_macrosurfaces_aero_grids"],
106         results_m["aircraft_struct_fem_elements"],
107         results_m["aircraft_struct_deformations"],
108         ax=grids_ax,
109         fig=grids_fig,
110         show_origin=True,
111         show_nodes=False,
112         line_color=color_pallet[i+1],
113         alpha=1,
114     )
115
116 #grids_ax, grids_fig = vis.plot_3D2.generate_deformed_aircraft_grids_plot(
117 #    flex_results[1]["aircraft_deformed_macrosurfaces_aero_grids"],
118 #    flex_results[1]["aircraft_struct_fem_elements"],
119 #    flex_results[1]["aircraft_struct_deformations"],
120 #    ax=grids_ax,
121 #    fig=grids_fig,
122 #    show_origin=True,
123 #    show_nodes=False,
124 #    line_color="tab:green",
125 #    alpha=1,
126 #)
127 #
128 #grids_ax, grids_fig = vis.plot_3D2.generate_deformed_aircraft_grids_plot(
129 #    flex_results[2]["aircraft_deformed_macrosurfaces_aero_grids"],
130 #    flex_results[2]["aircraft_struct_fem_elements"],
131 #    flex_results[2]["aircraft_struct_deformations"],
132 #    ax=grids_ax,
133 #    fig=grids_fig,
134 #    show_origin=True,
135 #    show_nodes=False,
136 #    line_color="tab:red",
137 #    alpha=1,
138 #)
139 #
140 #grids_ax, grids_fig = vis.plot_3D2.generate_deformed_aircraft_grids_plot(
141 #    flex_results[3]["aircraft_deformed_macrosurfaces_aero_grids"],
142 #    flex_results[3]["aircraft_struct_fem_elements"],
143 #    flex_results[3]["aircraft_struct_deformations"],
144 #    ax=grids_ax,
145 #    fig=grids_fig,
146 #    show_origin=True,
147 #    show_nodes=False,
148 #    line_color="tab:purple",
149 #    alpha=1,
150 #)
151 #
152 #grids_ax, grids_fig = vis.plot_3D2.generate_deformed_aircraft_grids_plot(
153 #    flex_results[4]["aircraft_deformed_macrosurfaces_aero_grids"],
154 #    flex_results[4]["aircraft_struct_fem_elements"],
155 #    flex_results[4]["aircraft_struct_deformations"],
156 #    ax=grids_ax,
157 #    fig=grids_fig,
158 #    show_origin=True,
159 #    show_nodes=False,
160 #    line_color="brown",
161 #    alpha=1,
162 #)
163
164 plt.show()
```

```
165
166 # Calculate Loads on each of the aerodynamic panels
167 aircraft_panel_loads = loads.functions.calculate_aircraft_panel_loads(
168     flex_results[1]["original_aircraft_panel_grid"], flex_results[1]["aircraft_force_grid"]
169 )
170
171 results_ax, results_fig = vis.plot_3D2.generate_results_plot(
172     aircraft_deformed_macrosurfaces_aero_grids=flex_results[1]["
173         aircraft_deformed_macrosurfaces_aero_grids"],
174     aircraft_panel_loads=aircraft_panel_loads,
175     aircraft_struct_fem_elements=flex_results[1]["aircraft_struct_fem_elements"],
176     aircraft_struct_deformations=flex_results[1]["aircraft_struct_deformations"],
177     results_string="delta_p_grid",
178     title="Aeronave HALE Delta de Pressão - Alfa 2ž",
179     colorbar_label="Delta de Pressão [Pa]",
180     ax=None,
181     fig=None,
182     show_origin=True,
183     colormap="coolwarm",
184 )
185 #plt.show()
186
187 # Deformation plot
188 fig1, ax1 = plt.subplots()
189 fig2, ax2 = plt.subplots()
190
191 for i, alpha in enumerate(alphas):
192
193     deformation_table = aelast.functions.calculate_deformation_table(
194         flex_results[i]["aircraft_original_grids"],
195         flex_results[i]["aircraft_struct_deformations"],
196     )
197
198     # sort nodes by desired column, in this case the Y coordinate
199
200     nodes = deformation_table["aircraft_macrosurfaces_deformed_nodes"][0]
201     nodes = nodes[nodes[:, 1].argsort()]
202
203     ax1.plot(nodes[:, 1], nodes[:, 2], label=f"Alfa {alpha}ž", color=color_pallet[i+1])
204     ax2.plot(nodes[:, 1], np.degrees(nodes[:, 4]), label=f"Alfa {alpha}ž", color=color_pallet[i+1])
205
206 # Plot Bending
207
208
209 ax1.grid()
210 ax1.set_title("Aeronave HALE Flexão da Asa")
211 ax1.set_ylabel("Flexão [m]")
212 ax1.set_xlabel("Semi-envergadura [m]")
213 ax1.legend()
214
215 # Plot Torsion
216
217
218 ax2.grid()
219 ax2.set_title("Aeronave HALE Torção da Asa")
220 ax2.set_ylabel("Torção [graus]")
221 ax2.set_xlabel("Semi-envergadura [m]")
222 ax2.legend()
223
224 # Deformation plot
225 fig1, ax1 = plt.subplots()
226 fig2, ax2 = plt.subplots()
227
228 for i, alpha in enumerate(alphas):
229
```

```

230     deformation_table = aelast.functions.calculate_deformation_table(
231         flex_results[i]["aircraft_original_grids"],
232         flex_results[i]["aircraft_struct_deformations"],
233     )
234
235     # sort nodes by desired column, in this case the Y coordinate
236
237     nodes = deformation_table["aircraft_macrosurfaces_deformed_nodes"][:, 1]
238     nodes = nodes[nodes[:, 1].argsort()]
239
240     ax1.plot(nodes[:, 1], nodes[:, 2], label=f"Alfa {alpha}ž", color=color_pallet[i+1])
241     ax2.plot(nodes[:, 1], np.degrees(nodes[:, 4]), label=f"Alfa {alpha}ž", color=color_pallet[i+1])
242
243 # Plot Bending
244
245
246 ax1.grid()
247 ax1.set_title("Aeronave HALE Flexão da Empenagem Horizontal")
248 ax1.set_ylabel("Flexão [m]")
249 ax1.set_xlabel("Semi-envergadura [m]")
250 ax1.legend()
251
252 # Plot Torsion
253
254
255 ax2.grid()
256 ax2.set_title("Aeronave HALE Torção da Empenagem Horizontal")
257 ax2.set_ylabel("Torção [graus]")
258 ax2.set_xlabel("Semi-envergadura [m]")
259 ax2.legend()
260
261 #plt.show()
262
263 interest_point = hale_aircraft.inertial_properties.position
264
265 ## Aerodynamic forces in the aircraft coordinate system
266 total_cg_aero_force, total_cg_aero_moment, component_cg_aero_loads = loads.functions.
267     calc_aero_loads_at_point(
268     interest_point,
269     results["aircraft_force_grid"],
270     results["aircraft_deformed_macrosurfaces_aero_panels"],
271     )
272 #
273 #print()
274 ##### HALE AIRCRAFT RESULTS #####
275 ##### HALE AIRCRAFT RESULTS #####
276 #print()
277 #print(f"# Total loads at aircraft CG:")
278 #print(f"    FX: {total_cg_aero_force[0]} N")
279 #print(f"    FY: {total_cg_aero_force[1]} N")
280 #print(f"    FZ: {total_cg_aero_force[2]} N")
281 #print(f"    RX: {total_cg_aero_moment[0]} N")
282 #print(f"    RY: {total_cg_aero_moment[1]} N")
283 #print(f"    RZ: {total_cg_aero_moment[2]} N")
284
285 V_X = 25
286 V_Y = 0
287 V_Z = 0
288
289 # Rotation velocities
290 R_X = 0
291 R_Y = 0
292 R_Z = 0
293
294 # Aircraft Attitude in relation to the wind axis, in degrees

```

```
295 ALPHA = 2 # Pitch angle
296 BETA = 0 # Yaw angle
297 GAMMA = 0 # Roll angle
298
299 # Center of rotation, usually the aircraft CG position
300 CENTER_OF_ROTATION = hale_aircraft.inertial_properties.position
301
302 # Flight altitude, used to calculate atmospheric conditions, in meters
303 ALTITUDE = 20000
304
305 flex_cls = [0]
306 flex_cms = [0]
307 flex_cds = [0]
308 rig_cls = [0]
309 rig_cms = [0]
310 rig_cds = [0]
311
312 for i, alpha in enumerate(alphas):
313
314     ALPHA = alpha
315
316     forces, moments, coefficients = loads.functions.calc_lift_drag(
317         aircraft=hale_aircraft,
318         point=interest_point,
319         speed=V_X,
320         altitude=ALTITUDE,
321         attitude_vector=np.array([ALPHA, BETA, GAMMA]),
322         aircraft_force_grid=flex_results[i]["aircraft_force_grid"],
323         aircraft_panel_grid=flex_results[i]["aircraft_deformed_macrosurfaces_aero_panels"],
324     )
325     flex_cls.append(coefficients['C1'])
326     flex_cms.append(coefficients['Cm'])
327     flex_cds.append(coefficients['Cd'])
328
329     forces, moments, coefficients = loads.functions.calc_lift_drag(
330         aircraft=hale_aircraft,
331         point=interest_point,
332         speed=V_X,
333         altitude=ALTITUDE,
334         attitude_vector=np.array([ALPHA, BETA, GAMMA]),
335         aircraft_force_grid=rig_results[i]["aircraft_force_grid"],
336         aircraft_panel_grid=rig_results[i]["aircraft_macrosurfaces_panels"],
337     )
338     rig_cls.append(coefficients['C1'])
339     rig_cms.append(coefficients['Cm'])
340     rig_cds.append(coefficients['Cd'])
341
342
343 fig, ax = plt.subplots()
344 ax.plot([0] + alphas, flex_cls, label="Flexível")
345 ax.plot([0] + alphas, rig_cls, label="Rígido")
346 ax.set_xlabel("Ângulo de Ataque [Graus]")
347 ax.set_ylabel("$C_l$")
348 ax.set_title("Aeronave HALE $C_l$ vs $\alpha$")
349 ax.grid()
350 ax.legend()
351
352 fig, ax = plt.subplots()
353 ax.plot([0] + alphas, flex_cms, label="Flexível")
354 ax.plot([0] + alphas, rig_cms, label="Rígido")
355 ax.set_xlabel("Ângulo de Ataque [Graus]")
356 ax.set_ylabel("$C_m$")
357 ax.set_title("Aeronave HALE $C_m$ vs $\alpha$")
358 ax.grid()
359 ax.legend()
360
361 fig, ax = plt.subplots()
```

```
362 ax.plot([0] + alphas, flex_cds, label="Flexível")
363 ax.plot([0] + alphas, rig_cds, label="Rígido")
364 ax.set_xlabel("Ângulo de Ataque [Graus]")
365 ax.set_ylabel("$C_d$")
366 ax.set_title("Aeronave HALE $C_d$ vs $\alpha$")
367 ax.grid()
368 ax.legend()
369
370 #plt.show()
371
372 print()
373 print("# Aerodynamic Coeffients:")
374 print(f" - Lift: {forces['lift']} N")
375 print(f" - Cl: {coefficients['Cl']} ")
376 print(f" - Drag: {forces['drag']} N")
377 print(f" - Cd: {coefficients['Cd']} ")
378 print(f" - Pitch Moment: {moments['pitch_moment']} N.m")
379 print(f" - Cm: {coefficients['Cm']} ")
380
381 # Create load distribution plots
382
383 flex_comp = []
384 rig_comp = []
385
386 for i, alpha in enumerate(alphas):
387
388     ALPHA = alpha
389
390     components_loads = loads.functions.calc_load_distribution(
391         aircraft_force_grid=flex_results[i]["aircraft_force_grid"],
392         aircraft_panel_grid=flex_results[i]["original_aircraft_panel_grid"],
393         aircraft_gamma_grid=flex_results[i]["aircraft_gamma_grid"],
394         attitude_vector=np.array([ALPHA, BETA, GAMMA]),
395         altitude=ALTITUDE,
396         speed=V_X,
397     )
398
399     flex_comp.append(components_loads)
400
401     components_loads = loads.functions.calc_load_distribution(
402         aircraft_force_grid=rig_results[i]["aircraft_force_grid"],
403         aircraft_panel_grid=rig_results[i]["aircraft_macrosurfaces_panels"],
404         aircraft_gamma_grid=rig_results[i]["aircraft_gamma_grid"],
405         attitude_vector=np.array([ALPHA, BETA, GAMMA]),
406         altitude=ALTITUDE,
407         speed=V_X,
408     )
409
410     rig_comp.append(components_loads)
411
412 fig1, ax1 = plt.subplots()
413 ax1.set_title(f"Aeronave HALE - ASA - Distribuição de Sustentação")
414 ax1.set_xlabel("Envergadura [m]")
415 ax1.set_ylabel("Sustentação [N/m]")
416 ax1.grid()
417
418 fig2, ax2 = plt.subplots()
419 ax2.set_title(f"Aeronave HALE - Empenagem Horizontal - Distribuição de Sustentação")
420 ax2.set_xlabel("Envergadura [m]")
421 ax2.set_ylabel("Sustentação [N/m]")
422 ax2.grid()
423
424 fig3, ax3 = plt.subplots()
425 ax3.set_title(f"Aeronave HALE - Empenagem Vertical - Distribuição de Sustentação")
426 ax3.set_xlabel("Envergadura [m]")
427 ax3.set_ylabel("Sustentação [N/m]")
428 ax3.grid()
```

```
429
430 color_pallet = [
431     "tab:blue",
432     "tab:orange",
433     "tab:green",
434     "tab:red",
435     "tab:purple",
436     "tab:brown",
437     "tab:pink",
438     "tab:gray",
439     "tab:olive",
440     "tab:cyan",
441 ]
442
443 for i, alpha in enumerate(alphas):
444
445     flex_component = flex_comp[i]
446     rig_component = rig_comp[i]
447
448     ax1.plot(flex_component[0]["y_values"], flex_component[0]["lift"], label=f"Flex - $\backslash\alpha$ = {alpha}\z", color=color_pallet[i+1])
449     ax1.plot(rig_component[0]["y_values"], rig_component[0]["lift"], label=f"Rig - $\backslash\alpha$ = {alpha}\z", linestyle="--", color=color_pallet[i+1])
450
451     ax2.plot(flex_component[1]["y_values"], flex_component[1]["lift"], label=f"Flex - $\backslash\alpha$ = {alpha}\z", color=color_pallet[i+1])
452     ax2.plot(rig_component[1]["y_values"], rig_component[1]["lift"], label=f"Rig - $\backslash\alpha$ = {alpha}\z", linestyle="--", color=color_pallet[i+1])
453
454     ax3.plot(flex_component[2]["y_values"], flex_component[2]["lift"], label=f"Flex - $\backslash\alpha$ = {alpha}\z", color=color_pallet[i+1])
455     ax3.plot(rig_component[2]["y_values"], rig_component[2]["lift"], label=f"Rig - $\backslash\alpha$ = {alpha}\z", linestyle="--", color=color_pallet[i+1])
456
457     ax1.legend()
458     ax2.legend()
459     ax3.legend()
460
461 #for results in flex_iteration_results[3]:
462 #
463 #    print(m.norm(results["aircraft_struct_deformations"][129][:3]))
464 #
465
466
467
468 plt.show()
469
470
471
472
473 input()
```

FOLHA DE REGISTRO DO DOCUMENTO			
1. CLASSIFICAÇÃO/TIPO DP	2. DATA 15 de abril de 2019	3. DOCUMENTO N° DCTA/ITA/DP-036/2019	4. N° DE PÁGINAS 221
5. TÍTULO E SUBTÍTULO: Desenvolvimento de uma ferramenta em Python para a análise de aeroelasticidade estática em aeronaves flexíveis			
6. AUTOR(ES): João Paulo Monteiro Cruvinel da Costa			
7. INSTITUIÇÃO(ÓES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÓES): Instituto Tecnológico de Aeronáutica – ITA			
8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR: Aeroelasticidade estática; Vortex lattice; Elementos finitos; Python.			
9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO: Método de malha turbilhonar; Aeroelasticidade; Estabilidade de aeronaves; Método de elementos finitos; Engenharia aeronáutica.			
10. APRESENTAÇÃO: <input checked="" type="checkbox"/> Nacional <input type="checkbox"/> Internacional ITA, São José dos Campos. Curso de Mestrado Profissional em Engenharia Aeronáutica. Programa de Pós-Graduação em Engenharia Aeronáutica e Mecânica. Orientador: Prof. Dr. Roberto Gil Annes da Silva; coorientador: Me. Marcos Paulo Halal Lombardi. Defesa em 29/03/2019. Publicada em 2019.			
11. RESUMO: Este trabalho apresenta o desenvolvimento de uma ferramenta computacional, chamada <i>Flying Circus</i> usando a linguagem de programação <i>Python</i> e bibliotecas de código aberto para a análise de aeroelasticidade estática de aeronaves em fase de projeto conceitual. A ferramenta usa um método de <i>vortex lattice</i> para os cálculos aerodinâmicos, um método de elementos finitos baseado em elementos de viga de Euler-Bernoulli para o cálculo estrutural, um critério de proximidade para o acoplamento aerodinâmico/estrutural e um método iterativo para o cálculo aeroelástico. Os resultados obtidos por essa ferramenta foram validados contra outros resultados disponíveis na literatura. A ferramenta foi utilizada para avaliar uma aeronave UAV (<i>Unmanned Aerial Vehicle</i>) da categoria HALE (<i>High Altitude Long Endurance</i>). O código fonte da ferramenta desenvolvida foi incluído nos anexos, juntamente com o código utilizado para gerar os resultados apresentados.			
12. GRAU DE SIGILO: <input checked="" type="checkbox"/> OSTENSIVO <input type="checkbox"/> RESERVADO <input type="checkbox"/> SECRETO			