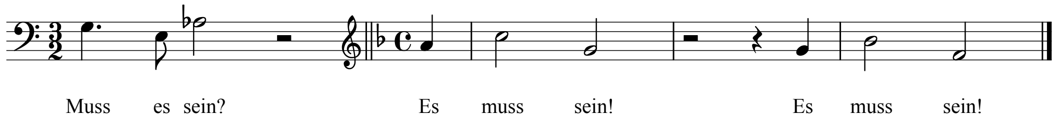
Inclusion of Geometrically Nonlinear Aeroelastic Effects into Gradient-Based Aircraft Optimization

by

Christopher A. Lupp

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Aerospace Engineering)
in the University of Michigan
2020



—Ludwig van Beethoven, String Quartet No. 16 (Op. 135)

Christopher A. Lupp

clupp@umich.edu

ORCID iD: 0000-0003-3436-549X

ACKNOWLEDGMENTS

I would like to thank my advisor Prof. Carlos Cesnik for the opportunity he gave me to complete this work, as well as for the support and advice over the years. I would also like to extend my gratitude to my committee members, Prof. Martins, Prof. Friedmann, and Prof. Maki.

I would also like to thank Cristina Riso, Divya Sanghi, Joshua Deaton (AFRL), Phil Beran (AFRL), Ziyang Pang, Jessica Jones, and Ryan Kitson for their help in the course of this work.

This work was supported in part by the NASA Advanced Air Transport Technology (AATT) High Aspect Ratio Wing (HARW) NRA Award (Karen Taminger and Carol Wieseman).

This work was supported by the U.S. Air Force Research Laboratory (AFRL) under the Michigan-AFRL Collaborative Center in Aerospace Vehicle Design (CCAVD), with Dr. Philip Beran as the task Technical Monitor. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

LIST OF ABBREVIATIONS

HALE	High Altitude Long Endurance
ISR	Intelligence, Surveillance, and Reconnaissance
FEM	Finite Element Method
UM/NAST	University of Michigan's Nonlinear Aeroelastic Simulation Toolbox
AD	Algorithmic Differentiation
A^2SRL	Active Aeroelasticity and Structures Research Laboratory
ISA	International Standard Atmosphere
UVLM	Unsteady Vortex Lattice Method
ODE	Ordinary Differential Equations
KS	Kreisselmeier-Steinhauser
MDO	Multi-disciplinary Design Optimization
EVP	eigenvalue problem
CFD	Computational Fluid Dynamics
DOE	Design of Experiments
LHS	Latin Hypercube Sampling
I/O	input/output
API	Application Programming Interface
DOF	degrees of freedom
ESP	Engineering Sketch Pad
FSI	fluid-structure interaction
BWB	Blended Wing Body

LCO	limit cycle oscillations
uCRM	undeformed Common Research Model
VLM	vortex lattice method
XDSM	extended design structure matrix
HPC	high performance computing
GVT	ground vibration testing
MAC	modal assurance criterion
IVT	Intermediate Value Theorem
MPI	Message Passing Interface
RBEs	Rigid Body Elements

LIST OF SYMBOLS

M	Mach number
AR	aspect ratio
a_∞	speed of sound, m/s
a_c	speed of sound at cruise, m/s
b	wing span, m
b_{max}	max. permissible wing span, m
b_{wing}	span of the wing section, m
C_D	aircraft drag coefficient
C_L	aircraft lift coefficient
C_f	friction coefficient
C_{Df}	friction drag coefficient
C_{Di}	induced drag coefficient
$C_{L,max}$	max. lift coefficient
c_{min}	min. permiss. wing section chord, m
c_{wing}	wing section chord, m
f_a	forces on the aerodynamic mesh
f_s	forces on the structural mesh
g	gravitational acceleration, m/s^2
$g\left(x\right)$	constraint function
h	geodetic altitude, m

h_{box}	wing box height
L	lift, N
M_c	cruise Mach number
M_{max}	max. Mach number
m	aircraft mass, kg
m_f	fuel mass, kg
m_{body}	body structural mass, kg
m_{struc}	structural mass, kg
N_x	number of variables or function inputs
q	dynamic pressure, Pa
q_F	flutter dynamic pressure
R	aircraft range, m
Re	Reynolds number
ρ	atmospheric density
S	wing surface area, m^2
S_{wet}	wing wetted area, m^2
sfc	specific fuel consumption
t_{body}	wing skin thickness, m
t_{wing}	wing skin thickness, m
u_a	displacements of the aerodynamic mesh
u_s	displacements on the structural mesh
V_{box}	wing box volume, m^3
V_{fuel}	fuel volume, m^3
v_c	cruise speed, m/s
v_{stall}	stall speed, m/s
W	aircraft weight, N
W_f	fuel weight, N

W_s	structural weight, N
w_i^δ	Euclidean weight
w_{box}	wing box width
x	function variable
x_{box}	x-location of the wing box center
y, $y\left(x\right)$	problem state variable
α	angle of attack
α_{root}	wing box skin thickness, m
δ	flap deflection
$\delta(a, b)$	distance metric between two points
η_{CS}	control surface deflections
η_{box}	wing box width relative to wing chord
κ_y	bending strain/curvature
κ_{max}	max. permissible bending strain
μ_{wing}	wing mass distribution, kg/m
ω	frequency values of the flutter solution
$\phi(x)$	linearized mode shape
$\bar{\rho}$	KS function constant
ρ_∞	atmospheric density, kg/m^3
ζ	damping values of the flutter solution
$\mathbb{R}^N$	N-dimensional real space

UM/NAST Variables

$\bar{C}_{FF}$, $\bar{C}_{FB}$, $\bar{C}_{BF}$, $\bar{C}_{BB}$	components of the damping matrix of the linearized system
c_d	drag coefficient
C_{FF}, C_{FB}, C_{BF}, C_{BB}	Components of the generalized damping matrix
$c_{l\alpha}$	lift slope
c_l	lift coefficient

$c_{m\delta}$	moment slope due to flap deflection
c_m	moment coefficient
F^{aero}, M^{aero}	aerodynamic loads in the body frame
F_1, F_2, F_3	inflow influence matrices
h	vector containing all nodal positions and rotations
h_w	vector containing the nodal positions and rotations
$J_{\theta\varepsilon}$	Jacobian relating nodal rotations to strain
$J_{\theta b}$	Jacobian relating nodal rotations to the body frame
$J_{h\varepsilon}$	Jacobian relating nodal positions and rotations to strains
J_{hb}	Jacobian relating the nodal positions and rotations to the body frame
$J_{p\varepsilon}$	Jacobian relating nodal positions to strains
J_{pb}	Jacobian relating nodal positions to the body frame
$\bar{K}_{FF}$	components of the stiffness matrix of the linearized system
K	stiffness matrix
K_{FF}	components of the generalized stiffness matrix
$c_{l\delta}$	lift slope due to flap deflection
l_{ac}, d_{ac}, m_{ac}	nodal aerodynamic loads (lift, drag, moment) evaluated at the aerodynamic center
$\bar{M}_{FF}$, $\bar{M}_{FB}$, $\bar{M}_{BF}$, $\bar{M}_{BB}$	components of the damping matrix of the linearized system
M_{FF}, M_{FB}, M_{BF}, M_{BB}	Components of the generalized mass matrix
N_{inflow}	number of inflow states
P_B	body frame offset vector
P_w	local beam frame offset vector
Q_1, Q_2, A, B	matrices of the state space equations
R_F, R_B	components of the generalized loads vector
s	beam coordinate
u	control input (in the state space equations)

v_B	translational body velocity
$\dot{y}$	state derivative
$\tilde{y}$	perturbed state vector
$\tilde{\dot{y}}$	perturbed state derivative
α_{eff}	effective angle of attack
β	body velocities
ε^{el}	element strain vector
ε_x	extensional strain
κ_x	twist curvature
κ_y	bending curvature about w_y
κ_z	bending curvature about w_z
λ	inflow states
λ_0	inflow velocities
ω_B	rotational body velocity
θ_B	orientation of the body frame

UM/NAST Reference Frames and Rotation Matrices

G	UM/NAST global frame
B	UM/NAST body frame
w	UM/NAST local beam node frame
C^{BG}	rotation matrix from the body to the global frame
C^{Bw}	rotation matrix from the body to the local beam node frame
w_x, w_y, w_z	UM/NAST local beam frame coordinate system vectors
x_B, y_B, z_B	UM/NAST body frame coordinate system vectors
x_G, y_G, z_G	UM/NAST global frame coordinate system vectors

ABSTRACT

While aircraft have largely featured flexible wings for decades, more recently, aircraft structures have rapidly become more flexible. The pursuit of longer ranges and higher efficiency through higher aspect ratio wings, as well as the introduction of modern, light-weight materials has yielded moderately and very flexible aircraft configurations. Past accidents, such as the loss of the Helios High Altitude Long Endurance (HALE) aircraft have highlighted the limitations of linear analysis methods and demonstrated the peril of neglecting nonlinear effects when designing such aircraft. In particular, accounting for geometrical nonlinearities in flutter analyses become necessary in aircraft optimization, including transport aircraft, or future aircraft may require costly modifications late in the design process to fulfill certification requirements. As a result, there is a need to account for geometrical nonlinearities earlier in the design process and integrate these analyses directly into the multi-disciplinary design optimization (MDO) problems.

This thesis investigates geometrically nonlinear flutter problems and how these should be integrated into aircraft MDO problems. First, flutter problems with and without geometrical nonlinearities are discussed and a unifying interpretation is presented. Furthermore, methods for interpreting nonlinear flutter problems are proposed and differences between linear and nonlinear flutter problem interpretation are discussed. Next, a flutter constraint formulation which accounts for geometrically nonlinear effects using beam-based analyses is presented. The resulting constraint uses a Kreisselmeier-Steinhauser aggregation function to yield a scalar constraint from flight envelope flutter damping values. While the constraint enforces feasibility over the entire flight envelope, how the flight envelope is sampled largely determines the flutter constraint's accuracy. To this end, a constrained Maximin approach, which is applicable for non-hypercube spaces, is used to sample the flight envelope and obtain a low-discrepancy sample set. The flutter constraint is then implemented using a beam-based geometrically nonlinear aeroelastic simulation code, UM/NAST.

As gradient-based optimization methods are used in MDO due to the large number of design variables in aircraft design problems, the flutter constraint requires the recovery of flutter damping sensitivities. These are obtained by applying algorithmic differentiation (AD) to the UM/NAST code base. This enables the recovery of gradients for any solution type (static, modal, dynamic, and flutter/stability) with respect to any local design variable available within UM/NAST. The performance of the gradient prediction is studied and a hybrid primal-AD scheme is developed to obtain the coupled nonlinear aeroelastic sensitivities. After verifying the accuracy and performance of the gradient evaluation, the flutter constraint was implemented in a sample optimization problem.

Finally, a roadmap for including the beam-based flutter constraint within an aircraft design problem is presented using analyses of varying fidelity. To this end, analyses of appropriate fidelity are used depending on the output of interest. While a shell-based FEM model can recover stress distributions, and is therefore well-suited for strength constraints, they are ill-suited for geometrically nonlinear flutter constraints due to their computational cost. Analyses are presented for a high aspect ratio transport aircraft configuration to illustrate the proposed approach and highlight the necessity for the inclusion of a geometrically nonlinear flutter constraint.

CHAPTER 1

Introduction

Aircraft have become progressively more flexible as wing aspect ratios increase and structural weights decrease in pursuit of higher performance. High Altitude Long Endurance (HALE) aircraft (e.g., AeroVironment's Helios, Figure 1.1a) present one class of vehicle that often exhibits very flexible wings. This results from high aspect ratio wings and long wing spans due to their mission profiles, such as Intelligence, Surveillance, and Reconnaissance (ISR) missions, and require long loiter times. HALE aircraft have long been the prototypical examples for very flexible vehicles. However, more recently, transport as well as other categories of aircraft have become more flexible as their wing span increases. Furthermore, configurations studied by NASA [1, 2] for next-generation transport aircraft feature high aspect ratios accompanied by substantial wing flexibility. Clearly, aircraft flexibility will increasingly need to be considered when designing new vehicles.

To those unfamiliar with aeroelasticity, the question may arise what advantages very flexible wings may entail. The answer may not be as simple as "none," but increasingly flexible structures certainly pose challenges to the design and certification processes. Moreover, this is true from both an aircraft performance and safety perspective, where changes in the wing deformation may yield vastly different performance than predicted, as well as concerns due to potential aeroelastic instabilities. Because of this, an increase in flexibility necessitates geometrically-nonlinear aeroe-

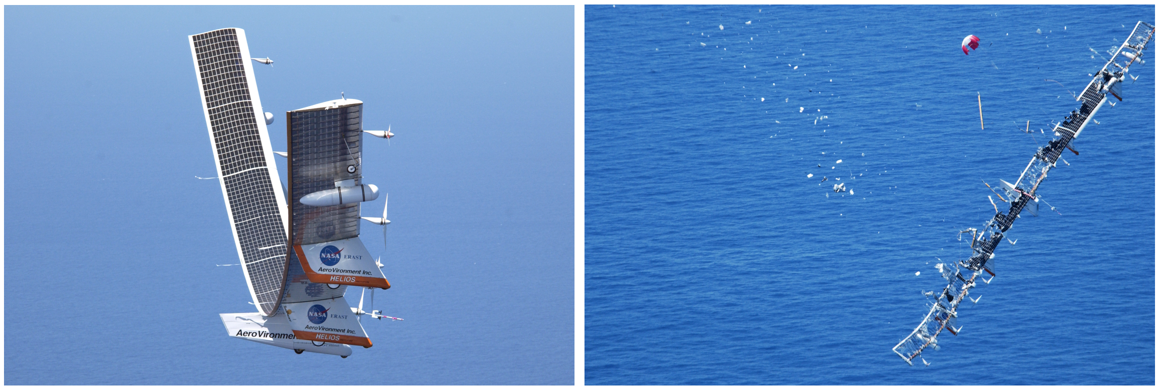

(a) Helios [7] (b) Disintegration of the Helios aircraft [7]

Figure 1.1: High performance aircraft increasingly feature very flexible wings. Aeroelastic instabilities must be properly identified and mitigated during vehicle design.

lastic analyses to ensure vehicle stability and performance.

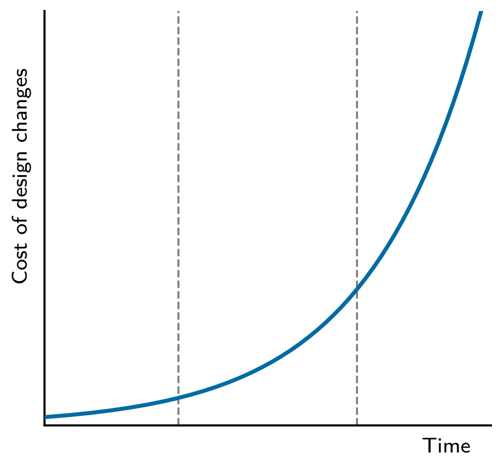
A lack of or insufficient analyses may yield catastrophic results. The Helios HALE) aircraft referenced earlier was lost (Figure 1.1b) due to gusts deforming the wing into a high dihedral configuration and an ensuing unstable pitch oscillation that ultimately led to an overspeed condition and vehicle disintegration [3]. Following the incident, an investigation [3] identified the "Lack of adequate analysis methods led to an inaccurate risk assessment of the effects of configuration changes leading to an inappropriate decision to fly an aircraft configuration highly sensitive to disturbances" as partialy responsible for the loss of the aircraft. The conclusions led to work in numerical aeroelastic analyses including geometrical nonlinearities and the inclusion of rigid body degrees of freedom [4, 5] as well as experimental studies, such as the University of Michigan's X-HALE [6].

Beyond requiring appropriate analysis tools and analyses, instabilities such as flutter must be identified early in the design process. The cost of design changes increases substantially over time (Figure 1.2a). A design change during flight testing may cost multiples of one during the preliminary or conceptual design phases. This cost may be monetary, but may also be paid in terms of a performance penalty and long-term operating costs. Depending on the severity of the penalty, this may ultimately render the design ineffective or cause the vehicle to be expensive to operate. Compounding the cost issue, the modeled detail of the vehicle behaves inversely proportional to the cost (Figure 1.2b). That is, relatively few analyses are performed during the conceptual design phases (due to a lack of information on mass, stiffness, and aerodynamics of the model) [8]. A high level of certainty in terms of analyses only is achieved during the late detailed design or even testing phase.

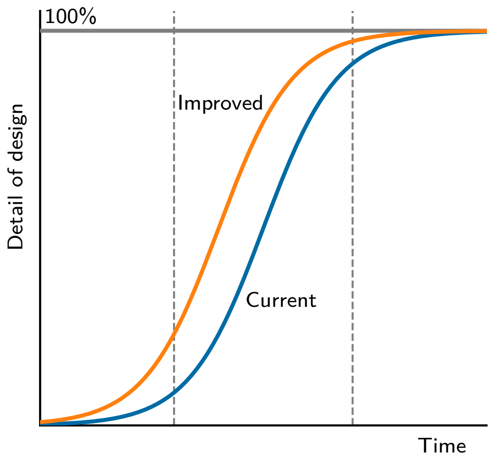
To enable higher-performance designs, it is necessary to shift the "s-curve" shown in Figure 1.2b to the left. While the additional detail in the earlier design phases may result in a cost reduction of the entire project, it also enables higher-performance designs. For example, by including a flutter constraint early in the design process together with aerodynamic and structural sizing in a Multi-disciplinary Design Optimization (MDO) problem [9, 10], the effect on the vehicle's performance may be assessed earlier with a smaller performance impact than if a costly modification is needed later. Conversely, MDO problems without the consideration of a flutter constraint may yield light-weight, yet infeasible, designs [11, 12].

One solution to shifting the s-curve is including additional analyses during conceptual design. As geometrically-nonlinear analyses may be computationally-prohibitive when conducted in high-fidelity, it may be advantageous to use a multi-fidelity approach. A multi-fidelity analysis utilizes varying levels of fidelity to account for different outputs of interest. For example, a vehicle may be modeled using shell Finite Element Method (FEM) elements for stress analyses, while beam-based methods account for flutter analyses. This mixed-fidelity approach may shift the s-curve by providing more detailed information (i.e., flutter analyses) into the design process.

However, before discussing multi-fidelity problems, an understanding of the term *fidelity* must be established. Indeed, the denotation and connotation of the word within the engineering community differ significantly. While the denotation (i.e., the



(a) Cost of design changes over time



(b) Level of design detail over time

Figure 1.2: The cost associated with design change increases substantially in later design phases. One approach to avoid costly design changes late in the process is to account for more analyses and phenomena earlier.

definition) means "accuracy in details" [13], the connotation within the community often entails "accuracy of the solution." This, in fact, may not be the case. Increasing the level of fidelity (e.g., from a beam to a shell element) may yield additional information, such as additional degrees of freedom, but the accuracy of the solution may not increase. In fact, the accuracy of the solution depends greatly on the quantity of interest. If a strength analysis is required, a shell-based analysis will likely be more applicable than a beam-based one. On the other hand, in the case of a modal frequency analysis, a beam-based analysis may yield equally accurate results as the shell model (unless local cross section effects become dominant). Choosing the appropriate fidelity for the problem, therefore, is key.

A multi-fidelity analysis or optimization problem utilizes varying levels of analysis and model fidelity and combines them to a single problem. The promise of this approach is to enable the evaluation of objectives and constraints with the appropriate level of fidelity required. For example, a fuel burn objective may require higher-fidelity CFD simulations, while a flutter constraint may be accurate when analyzed using beam-based models. Choosing the appropriate fidelity ensures that the desired quantity (objective, constraint) is accurately modeled, while the computational expense can be limited.

1.1 Previous Work

This section reviews existing literature relevant to the work presented in this dissertation. Jonsson and coworkers[1] [14] compiled a comprehensive review of flutter and post-flutter constraints in aircraft optimization problems. This section constitutes a more concise review of flutter prediction and constraints as they pertain to this work.

1.1.1 Flutter Constraints

Due to the large computational cost associated with predicting flutter in time-domain, flutter prediction is typically performed in the frequency-domain. Eigenvalue-analysis methods, such as the *k*-, *p*-, *pk*- and *g*-methods [15, 16, 17, 18, 19], are widely used and applicable to linear aeroelastic systems as well as nonlinear systems which have linearized about a state of equilibrium. While gradient-free optimization methods (e.g., Particle Swarm [20] and Genetic Algorithms [21]) require only the function evaluation of the flutter prediction, gradient-based optimization algorithms additionally require the aeroelastic sensitivities. Due to the large number of design variables in MDO problems, gradient-free methods may prove computationally expensive. Gradient-based methods provide a feasible solution to these high-dimensional optimization problems [22, 23].

While flutter constraints have been applied since the 1970's such as by Haftka [9, 24], Hajela [25], Bhatia and Rudisill [26, 27] or Gwin and Taylor [28], as well as in the early 1990's by Livne, Schmit and Friedmann [29, 30], MDO problems still do not

[1] I am a coauthor of this work

regularly account for flutter. More recently, researchers have again investigated the effect on the optimized configuration by including flutter constraints, mostly focused on geometrically linear flutter problems.

Early flutter constraints attempted to constrain the flutter speed. Bhatia and Rudisill [26] conducted a wing mass minimization subject to an unchanged flutter speed. Rudisill and Bhatia [27] later improved their method by determining second-order derivatives of the flutter speed and eigenvalues with respect to the design variables. Similarly, Gwin and Taylor [28] constrained a minimum flutter speed while conducting a mass minimization problem. While these studies used simplified structural and aerodynamic models due to limited computational resources, their constraint of the flutter speed may lead to discontinuities (a detailed explanation of this is provided in Chapter 5).

A continuous flutter (and divergence) constraint was formulated by Ringertz [31]. Ringertz constrained the damping values of the flutter problem and applied the constraint to both a mass minimization of both a rectangular wing as well as a swept and tapered wing. However, Ringertz approach resulted in a large number of damping constraints.

Stanford and coworkers [32] investigated new aeroelastic tailoring schemes for aircraft mass minimization. Similar to Rigertz' approach [31], they constrained the system damping values to lie beneath a prescribed stability boundary. They applied six different aeroelastic tailoring methods to the undeformed Common Research Model (uCRM) configuration [33] to study the difference of effectiveness between metallic thickness variations, functionally graded materials, balanced or unbalanced composite laminates, curvilinear tow steering, and distributed trailing edge control surfaces. Stanford and coworkers utilized the nonlinear higher-fidelity ZEUS code to account for steady aerodynamics, while transonic small disturbance theory was used to account for unsteady aerodynamics. This permitted them to model the aerodynamic nonlinearities associated with the transonic flight regime.

Jonsson and coworkers [34] developed a flutter constraint which could account for wing planform changes. They used a KS aggregate [35] the damping values to prevent constraint discontinuities. The gradients of the aggregated constraint values were determined using both analytical and Algorithmic Differentation (AD)-based derivatives. They studied a rectangular (flat plate) wing with respect to thickness and wing planform variables and were able to obtain a flutter-free, higher aspect ratio design.

While the previously discussed work developed geometrically linear flutter constraints (in some cases, such as Stanford et al.[32], aerodynamic nonlinearities were considered), geometrically nonlinear flutter constraints, with few exceptions, have not been rigorously studied. Xie and coworkers [36] studied the design optimization of a wind tunnel model and constraining the flutter speed of the wing. However, their approach used a gradient-free optimizer. As noted earlier, gradient-free methods may become computationally infeasible for large scale MDO problems.

Variyar and coworkers [12] investigated optimization of unconventional aircraft configurations including a flutter constraint. They coupled the aircraft design toolbox SUAVE [37] with ASWING [38] to account for a geometrically nonlinear flutter constraint. However, as was the case in earlier flutter constraint studies, they constrained the flutter speed to lie beneath a predefined boundary. As previously mentioned, this approach may lead to constraint discontinuities (see Chapter 5). Furthermore, the constraint derivatives were determined using finite differences.

Finally, Bhatia and Beran [39] conducted an optimization of a thermally stressed structure subject to a transonic flutter constraint. They modeled the plate structure using a nonlinear von Kármán strain Timoshenko beam and accounted for transonic aerodynamics using an Euler solution.

While linear flutter constraints constitute a large body of work, comparitively little research has been conducted into geometrically nonlinear flutter constraints and their effect on the optimized configuration. As future vehicles reach new degrees of structural flexibility, it will become necessary to account for coupling between geometrically nonlinear aeroelasticity and flight dynamics [4, 40].

1.1.2 Multi-Fidelity Problems

While linear flutter constraints have been implemented using higher-fidelity (e.g., shell-based structural models) methods, geometrically nonlinear flutter constraints have generally been limited to beam-based analyses due to computational expense. However, as has been noted, flutter constraints which account for geometrical nonlinearities will play an increased role. As such, including the beam-based analyses into the higher-fidelity MDO frameworks is one possible solution.

Multi-fidelity problems combine analyses of varying fidelity to form a single optimization problem. For example, Bryson and Rumpfkeil [41] investigated the multi-fidelity design of a chevron-shaped vehicle. They utilized both Euler Computational Fluid Dynamics (CFD) solutions as well as panel-based aerodynamic solutions tightly coupled with a linear FEM solution. The multi-fidelity approach resulted in a lower computational cost per iteration.

Furthermore, multi-fidelity approaches have been used to include flutter constraints into higher-fidelity optimization problems. Opgenoord and coworkers [42, 43] combined a linear flutter constraint with a lattice-based topology optimization problem. The properties for the beam-based flutter constraint were evaluated using a condensation process. This process determined equivalent beam properties from the truss-based lattice structure.

Finally, Stodieck and coworkers [44] presented a beam condensation process intended to connect lower-fidelity, nonlinear aeroelastic solutions to higher-fidelity structural models. The structural condensation process utilizes the stiffness condensation presented by Malcolm and coworkers [45] and extends it with derivatives making it suitable for gradient-based optimizaton frameworks.

1.2 Scope of this Work

This dissertation attempts to address the uncertainty of early design iterations and future, very flexible aircraft by proposing a methodology for including a geometrically nonlinear flutter constraint into aircraft optimization problems. The dissertation is divided into four parts, starting with a description of the methods used, a summary of the numerical tools used and developed within this work, numerical studies, and finally concluding statements.

Part I describes a variety of methods for determining gradients of numerical problems with their respective advantages and disadvantages. Next, I describe the theoretical formulation of the UM/NAST framework along with a new flutter analysis method introduced in this work. In Chapter 5, I formulate a flutter constraint including geometrical nonlinearities based on an existing approach proposed by Jonsson and coworkers [34], which uses a constrained flight envelope sampling (described in Chapter 6). Finally, I describe the beam condensation method used to couple the higher fidelity structural model with the beam-based (and low-fidelity aerodynamics-based) UM/NAST analyses and how gradients for this method are obtained.

Part II describes the numerical tools used in the dissertation. It begins with a description of available AD tools and details the selection process for the tool used. Chapter 9 describes the individual components of the MDO framework utilized for the numerical studies in Chapter 16. Finally, Chapter 10 describes the UM/NAST version used and details improvements provided to the framework, while Chapter 11 describes a tool designed for using UM/NAST in MDO problems.

Part III details the numerical studies conducted within this work. Chapter 13 investigates the accuracy of the new flutter algorithm presented in Chapter 4. Next, I verify the accuracy of the sensitivities obtained within UM/NAST (Chapter 14) and apply the geometrically nonlinear flutter constraint to a beam-based optimization problem in Chapter 15. Chapter 16 presents studies of the assembled multi-fidelity problem, representing a roadmap to the inclusion of the flutter constraint into higher-fidelity optimization problems.

Finally, Part IV describes conclusions, contributions of this dissertation and outlines potential areas of future work.

Part I

Methods

CHAPTER 2

Determining Gradients

Many MDO problems use gradient-based optimization due to a large number of design variables and the associated advantages of gradient-based optimization for such problems. As such, gradients and how to determine them accurately and efficiently play a key role during the development of tools intended for design applications. While many methods to determine gradients for numerical tools exist, three prevalent methods and their applicability to the problems addressed in this thesis are discussed here. And despite the obvious focus on gradient-based optimization, the methods discussed in this chapter can be used for other applications, as presented in Chapter 3.

Choosing a method for determining gradients requires evaluating the strenghts and weaknesses of each method. Some methods require little or no implementation, yet suffer from low accuracy, while others may yield the computationally fast and accurate results at the expense of implementation time and effort. The appropriate method therefore usually represents a compromise between accuracy and computational performance, while being constrained by the implementation time that may be permitted by project deadlines and times allocated for development.

2.1 Finite Difference Method

Finite difference methods are a popular perturbation-based family of methods used to numerically approximate the gradient of a function. While their popularity may be based on the relatively small amount of effort required to implement them, they suffer from accuracy and performance issues [46].

2.1.1 Derivation

While there are many finite difference methods, derived from Taylor series approximations, the term finite difference method is often used to describe the forward difference method, due to its wide-spread use. As with other finite difference methods, the forward difference method formula can be derived from the Taylor series, with a known perturbation h:

$$f\left(x+h\right) = f\left(x\right) + hf'\left(x\right) + h.o.t. \tag{2.1}$$

The forward difference formula is then determined by rearranging Equation 2.1:

$$f'\left(x\right) = \frac{f\left(x+h\right) - f\left(x\right)}{h} + \mathcal{O}\left(h\right) \tag{2.2}$$

The higher order terms can be neglected to obtain the approximation:

$$f'\left(x\right) \approx \frac{f\left(x+h\right) - f\left(x\right)}{h} \tag{2.3}$$

2.1.2 Implementation and Accuracy

Finite difference methods and the forward difference method, in particular, may enjoy wide-spread popularity due to their ease of implementation. It does not require access to or modification of the source code to be differentiated. To determine gradients a perturbation is added to an element of the function inputs and the function is evaluated. This must be conducted for every function input (a total of N_x times), each evaluation yielding a column of the Jacobian. As a result, determining the gradients for a function with a large number of inputs becomes computationally expensive.

Additionally, the finite difference method suffers from accuracy and reliability issues. While the forward difference method is $\mathcal{O}(h)$ and this would indicate that a smaller perturbation would result in more accurate results, this is not the case. As the perturbation becomes smaller, the subtraction of numbers of similar magnitude yields cancellation errors that can become substantial (Figure 2.1). This remains true for the central difference method, despite its $\mathcal{O}(h^2)$ convergence rate. Because a coarse perturbation yields inaccurate results and an excessively small one yields cancellation errors, the application of the finite difference method ideally requires a convergence study for every change in the input variables.

2.2 Complex Step Method

The complex step method is a relatively new method to determine derivatives of real functions by complex perturbations. Martins [46] describes that the method was first developed by Lyness and Moler [47] as well as Lyness [48] and later rediscovered by Squire and Trapp [49]. Furthermore, Martins and coworkers [50] presented an alternate derivation, drew a connection to AD, and demonstrated the applicability of the method to any algorithm, forming the basis of the method used in this work. Unlike the finite difference method, the complex step method does not experience cancellation errors due to subtraction and can achieve high accuracy results. Due to the use of complex numbers, this method generally requires access and some modifications to the function's source code.

2.2.1 Derivation

Like the finite difference method, the complex step formula can be derived from a Taylor series expansion, in this case for a complex function:

$$f(x+ih) = f(x) + ihf'(x) - h^2 \frac{f''(x)}{2} - \dots \tag{2.4}$$

Rearranging for the first derivative, and taking the imaginary part yields:

$$f'(x) = \frac{\operatorname{Im}(f(x+ih))}{h} + \mathcal{O}\left(h^2\right) \tag{2.5}$$

Neglecting the higher order terms results in the approximation:

$$f'(x) \approx \frac{\operatorname{Im}(f(x+ih))}{h} \tag{2.6}$$

2.2.2 Implementation and Accuracy

Unlike the finite difference implementation, the complex step method requires access to the function source code and modifications to enable a complex evaluation of the function. While this inevitably results in a more difficult process to prepare for gradient evaluation, the modifications to source code may be minimal. Martins et al. [50] provided tools and guides for easily converting source code for the complex step method[1]. With these helpers and type definitions, the effort of applying the complex step method has been greatly reduced, and in the case of templated code (e.g., in C++) may be as simple as calling the code with the predefined (*cplx*) type. Like the finite difference method, the input variables must be perturbed one at a time for a total of N_x function evaluations, with each evaluation yielding a column of the

[1]This work uses a modified version of the *complexify.h* header file [51]. I added function definitions and overloads to the *cplx* type, as the missing implementations led to compilation errors and warnings, particularly when paired with the Eigen linear algebra library.

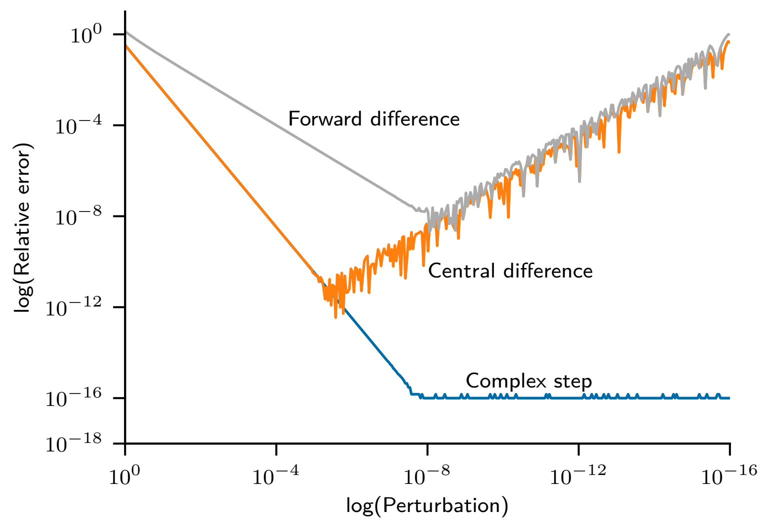


Figure 2.1: Comparison the gradient accuracy for the forward difference (gray), central difference (orange), and complex step (blue) methods for the function $f(x) = x^3$.

Jacobian. However, because of the function evaluations are conducted using complex numbers instead of real floating precision types, the memory overhead doubles (at least) and the computational performance may be reduced by a factor of two to four.

The accuracy of the complex step method, however, rewards the implementation effort compared to the finite difference method. When comparing the forward difference, central difference, and complex step methods (Figure 2.1), the advantages of the latter become clear. While the central difference and complex step methods are both $\mathcal{O}(h^2)$, the complex step method does not experience cancellation errors due to subtraction. As a result, if the step size is chosen properly, the complex step method can predict gradients to the computer's working precision.

To obtain derivative at machine precision, the size of the perturbation must be chosen appropriately. Assuming finite-precision arithmetic with a relative working precision of ε, the truncation errors of the function (from Equation 2.4) can be eliminated by choosing a perturbation such that [46]:

$$h^2 \left| \frac{f''(x)}{2} \right| < \varepsilon \left| f(x) \right| \tag{2.7}$$

To eliminate the truncation error of the derivative, a similar condition can be derived from Equation 2.5:

$$h^3 \left| \frac{f'''(x)}{6} \right| < \varepsilon \left| f'(x) \right| \tag{2.8}$$

However, Martins [46] notes that fulfilling both of these conditions may not always be possible. Nonetheless, the complex step method offers a high-accuracy method for determining gradients, despite its increased computational cost.

2.3 Semi-Analytical Methods

Semi-analytical methods require more intrusive changes to the source code than the methods presented. Moreover, they necessitate a detailed theoretical understanding of the computational problem, which the finite difference and complex step methods do not. The methods are named “semi-analytical” because the developer uses analytical derivations to reduce the gradient evaluation to a smaller problem. This smaller problem may then be evaluated using an arbitrary gradient method (e.g., complex step) to obtain the gradients of interest. The large implementation effort, if properly executed, may result in a much more efficient gradient evaluation than any of the methods presented so far in this chapter.

To apply a semi-analytical method, the function of interest, the design variables, and the problem state variables must first be identified. The function typically depends on both the design as well as the state variables:

$$f = F(x, y(x)) \tag{2.9}$$

where x is the variable the function depends on, and y is a state variable that depends on x. The computational problem is governed by the residual, which also depends on the design and state variables:

$$r = R(x, y(x)) = 0 \tag{2.10}$$

An example of a residual for a structural finite element problem with applied forces f, stiffness matrix K and displacements d is:

$$R(x, d(x)) = Kd - f = 0 \tag{2.11}$$

In this case the state variables are the displacements d, while the design variables may be the element thicknesses (which would influence the stiffness matrix K).

2.3.1 Derivation

In this section, two semi-analytical methods will be derived and presented: the direct and adjoint methods. The derivation presented here follows the typical derivation that may be found in literature [46].

The total derivative of the function with respect to the design variables may be rewritten using the chain rule:

$$\frac{df}{dx} = \frac{\partial F}{\partial x} + \frac{\partial F}{\partial y}\frac{dy}{dx} \tag{2.12}$$

Similarly, the residual equations can be rewritten:

$$\frac{dr}{dx} = \frac{\partial R}{\partial x} + \frac{\partial R}{\partial y}\frac{dy}{dx} = 0 \tag{2.13}$$

Note, that the total derivatives of the residual equations with respect to the design variables must also be zero, as the governing equations must always be fulfilled. Additionally, while the partial derivative notation $\partial(\cdot)$ represent the variation of a quantity for a fixed state y, the total derivative $d(\cdot)$ accounts for changes in the state variable so that the governing equations remain fulfilled. The residual equations (2.13) can be rewritten to obtain the total derivatives dy/dx:

$$\frac{\partial R}{\partial y}\frac{dy}{dx} = -\frac{\partial R}{\partial x} \tag{2.14}$$

Substituting Equation 2.14 into 2.12 yields:

$$\frac{df}{dx} = \frac{\partial F}{\partial x} - \underbrace{\frac{\partial F}{\partial y}\overbrace{\left[\frac{\partial R}{\partial y}\right]^{-1}}^{}}_{\frac{df}{dr}}\!\!\!\!\!\!\!\!\!\overbrace{\phantom{\left[\frac{\partial R}{\partial y}\right]^{-1}}\frac{\partial R}{\partial x}}^{-\frac{dy}{dx}} \tag{2.15}$$

There are two solution approaches to Equation 2.15: the direct (or forward) and the adjoint (or reverse) solutions. It should be noted that the inverse of $\partial R/\partial y$ is not calculated, but rather, the solution involves the solution of a linear system of equations.

The direct method is preferable when the number of functions is greater than the number of design variables. To solve for the total derivatives of the functions with respect to the design variables, Equation 2.16 is solved for dy/dx and then substituted into 2.17:

$$-\frac{\partial R}{\partial y}\frac{dy}{dx} = \frac{\partial R}{\partial x} \tag{2.16}$$

$$\frac{df}{dx} = \frac{\partial F}{\partial x} + \frac{\partial F}{\partial y}\frac{dy}{dx} \tag{2.17}$$

Similarly, the adjoint solution is obtained by first determining df/dr in Equation 2.18 and substituting into 2.19:

$$-\left[\frac{\partial R}{\partial y}\right]^T \left[\frac{dy}{dx}\right]^T = \left[\frac{\partial F}{\partial y}\right]^T \tag{2.18}$$

$$\frac{df}{dx} = \frac{\partial F}{\partial x} + \frac{df}{dr}\frac{dR}{dx} \tag{2.19}$$

2.3.2 Implementation and Accuracy

The semi-analytical methods are the most computationally efficient and accurate methods presented here. They reduce the size of the problem that needs to be differentiated numerically (e.g., using complex step), thereby increasing computational performance. However, implementation of the methods requires additional effort. Furthermore, they require access to source code, which may not be available. Finally, the implementation process is prone to error. As a result, the derivatives must be verified using alternate methods. Because of the accuracy deficiencies of the finite difference method, the complex step method or AD may be required during the verification of the semi-analytical derivatives. As those methods require implementation changes of their own, verification may require a substantial development effort in addition to the implementation of the semi-analytical method itself.

2.4 Algorithmic Differentiation

AD is a method of determining gradients from an existing source code. It determines the derivatives of all operations conducted within the software and obtains total derivatives using the chain rule. Similar to the complex step method, it requires access to the source code and may require substantial implementation changes.

2.4.1 Principle

AD decomposes the software into a series of operations (e.g., lines of code) V_i, to which the derivatives are known. Additionally, local variables are stored in an array v_i. In that sense, every line of code is differentiated and the total derivatives of the functions with respect to the design variables of interest are determined using the chain rule. This can be achieved by several different techniques, which are discussed in Section 2.4.2.

As in the semi-analytical methods, there are two methods for determining the total derivatives using AD. The forward mode is given by [46]:

$$(I - D_V)\, D_v = I \tag{2.20}$$

or the reverse mode is given by [46]:

$$(I - D_V)^T\, D_v^T = I \tag{2.21}$$

The matrices in Equations 2.20 and 2.21 are:

$$D_V = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ \frac{\partial V_2}{\partial v_1} & 0 & 0 & \cdots & 0 \\ \frac{\partial V_3}{\partial v_1} & \frac{\partial V_3}{\partial v_2} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 & 0 \\ \frac{\partial V_n}{\partial v_1} & \frac{\partial V_n}{\partial v_2} & \cdots & \frac{\partial V_n}{\partial v_{n-1}} & 0 \end{bmatrix} \tag{2.22}$$

$$D_v = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ \frac{dv_2}{dv_1} & 0 & 0 & \cdots & 0 \\ \frac{dv_3}{dv_1} & \frac{dv_3}{dv_2} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 & 0 \\ \frac{dv_n}{dv_1} & \frac{dv_n}{dv_2} & \cdots & \frac{dv_n}{dv_{n-1}} & 0 \end{bmatrix} \tag{2.23}$$

The forward mode solution is akin to solving for one column of D_V from Equation 2.20 using forward substitution. These operations are executed together with the original code. The reverse mode solves the derivatives using back substitution. This requires storing the individual operations and variables in what is typically called a tape. As with the analytical methods, the forward and reverse modes are applicable for different problems. The forward mode is typically more efficient when the number of functions is greater than the number of variables of interest, while the reverse mode is faster when the number of design variables is larger than the number of functions.

To illustrate the application of AD, consider the following program, adapted to C++ from Martins [46]:

```
vector<double> x(2);
double det;
vector<double> y(2);
vector<double> f(2);
```

```
det = 2 + x[0] * pow(x[1], 2);
y(1) = pow(x[1],2) * sin(x[0]) / det;
y(2) = sin(x[0]) / det;
f(1) = y[0];
f(2) = y[1] * sin(x[0]);
```

Here the array of variables is:

$$v = \begin{bmatrix} x[0] \\ x[1] \\ det \\ y[0] \\ y[1] \\ f[0] \\ f[1] \end{bmatrix} \tag{2.24}$$

The corresponding reverse mode representation can be solved over two back-substitutions given by [46]:

$$\begin{bmatrix} 1 & 0 & -v_2^2 & -\frac{v_2^2 \cos v_1}{v_3} & -\frac{\cos v_1}{v_3} & 0 & -v_5 \cos v_1 \\ 0 & 1 & -2v_1 v_2 & -\frac{2v_2 \sin v_1}{v_3} & 0 & 0 & 0 \\ 0 & 0 & 1 & -\frac{v_2^2 \sin v_1}{v_3^2} & -\frac{\sin v_1}{v_3} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & -\sin v_1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{dv_6}{dv_1} & \frac{dv_7}{\partial v_1} \\ \frac{dv_6}{dv_2} & \frac{dv_7}{\partial v_2} \\ \frac{dv_6}{dv_3} & \frac{dv_7}{\partial v_3} \\ \frac{dv_6}{dv_4} & \frac{dv_7}{\partial v_4} \\ \frac{dv_6}{dv_5} & \frac{dv_7}{\partial v_5} \\ 1 & \frac{dv_7}{\partial v_6} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (2.25)$$

2.4.2 Implementation and Accuracy

Beside the semi-analytical methods, AD may require the largest effort during implementation. However, it should return gradients at the algorithm's relative working precision, and importantly requires less function evaluations than the finite difference and complex step methods. Therefore, it is possible to obtain derivatives computationally more efficiently than the perturbation-based methods.

While term AD is used as an overarching term, it describes multiple methods: manual transformation, automatic source code transformation, and operator overloading [46]. Typically, manual source code transformation is impractical and error prone and should generally be avoided. Source code transformation tools exist for a variety of languages. However, tools for several higher-level languages may not exist or be freely available. In general terms, the more complex the programming language (e.g., the higher-level), the more difficult the creation of a source code transformation tool will be. As such, tools such as Tapenade [52] exist for Fortran and C, but not for C++. Additionally, the workflow for applying a source code transformation tool may not be trivial. While the tool returns transformed code, manual intervention may be required to obtain a compilable program. Finally, operator overloading offers a relatively simple tool to apply AD to higher level languages. In this method, a separate AD datatype is defined and used within the software. The AD type consists of a tuple. For every source code operation the AD type conducts a function evaluation and the corresponding gradient evaluation (of that operation only). The total derivatives are determined using the chain rule either via forward or back substitution. While operator overloading may result in performance penalties compared to source transformation, it is comparatively simple to apply. This is particularly true if the source code has been templated, as the AD type can be applied directly in this scenario with minimal source code modification.

CHAPTER 3

UM/NAST Theoretical Formulation

The University of Michigan's Nonlinear Aeroelastic Simulation Toolbox (UM/NAST) is a software package used at the Active Aeroelasticity and Structures Research Laboratory (A^2SRL) to model very flexible aircraft and constitutes the primary numerical tool used in this thesis. This chapter describes the theoretical formulation of UM/NAST and the new linearization schemes added to the framework within the scope of this work.

UM/NAST couples a geometrically nonlinear, strain-based beam formulation with fully coupled nonlinear flight dynamics to a variety of aerodynamic model ranging from strip theory with Peters' finite state aerodynamics [53] accounting for unsteady effects to an Unsteady Vortex Lattice Method (UVLM) implementation [54, 55]. The formulation of the equations of motion presented here has been developed over several generations of graduate students at A^2SRL advised by Prof. Cesnik (Brown [56], Shearer [57], Su [58], Dillsaver [59], Jones [60], Pang [61], Kitson [62], and Teixeira [63]). These developments have been previously summarized by Pang [61] and are recounted here to provide background information and context to the new developments in this thesis. As such, Section 3.1 presents the past work of Brown, Shearer, Su, and Cesnik, while Section 3.2 describes the finite difference linearization presented by Pang [61] as well as new, high-accuracy linearization methods I developed for this thesis. While the linearization processes are demonstrated using the A matrix, I

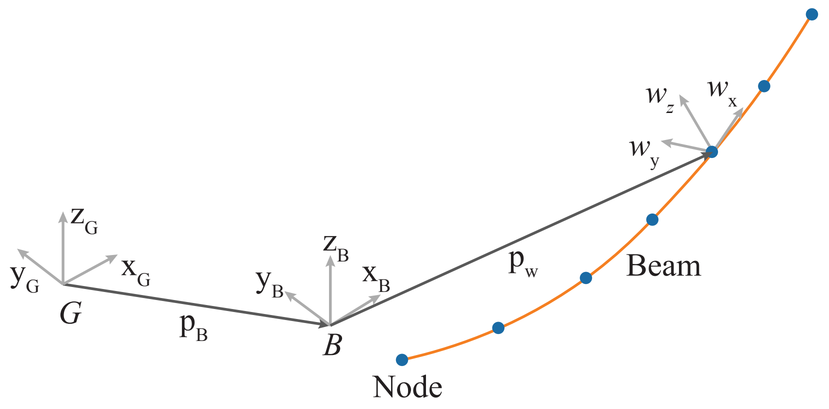


Figure 3.1: Coordinate system definitions within UM/NAST.

developed the linearization methods for both the A and B matrices[1].

3.1 Nonlinear Coupled Equations of Motion

3.1.1 Strain-Based Beam Formulation

UM/NAST uses four main coordinate systems types to obtain a geometrically nonlinear solution (Figure 3.1)[56]: the global frame (G), the body frame (B), the frame local to each beam node (w), and a local aerodynamic frame (a). The global frame serves as an inertial reference system, while the body frame serves as the vehicle reference frame and moves through space at the vehicle's velocity.

The body frame is offset from the global frame by the vector P_B and its orientation

[1]The implementations for the B matrix were tested and debugged by Mateus Pereira. The original linearizations of the B and B_w matrices were originally developed by Dillsaver [59]

with respect to the global frame is defined by the quaternion vector ζ:

$$P_B = \begin{bmatrix} x_B \\ y_B \\ z_B \end{bmatrix}, \tag{3.1}$$

$$\zeta = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}. \tag{3.2}$$

The body frame rigid body motion is captured by three linear and three angular velocities, resulting in a total of 13 rigid body states:

$$b = \begin{bmatrix} p_B \\ \theta_B \end{bmatrix}, \qquad \dot{b} = \beta = \begin{bmatrix} v_B \\ \omega_B \end{bmatrix}. \tag{3.3}$$

The vehicle structure is subdivided into subassemblies, called members. Each member contains beam elements. The geometrically nonlinear beam element used in UM/NAST is a three-noded, constant strain element. The strain states for every beam element are: extensional, twist, and two bending curvatures (in-plane and out-of-plane):

$$\varepsilon^{el} = \begin{bmatrix} \varepsilon_x \\ \kappa_x \\ \kappa_y \\ \kappa_z \end{bmatrix}. \tag{3.4}$$

The w frame, located at every node, describes the beam nodes' location and orientation in relation to origin of the body frame (Figure 3.1). Similar to P_B for the body frame, the vector P_w describes the offset of the w frame from the body frame. A column vector h can be defined that contains a point's spatial position and orientation information. The vector P_w and the coordinate system unit vectors (described in the body frame) w_x, w_y, and w_z, which are stacked to obtain the vector h_w:

$$h_w\left(s\right) = \begin{bmatrix} P_w(s) \\ w_x(s) \\ w_y(s) \\ w_z(s) \end{bmatrix}. \tag{3.5}$$

For the spatial information of the same point in the global frame, the h vector is defined:

$$h\,(s) = \begin{bmatrix} P_b + P_w(s) \\ w_x(s) \\ w_y(s) \\ w_z(s) \end{bmatrix}. \tag{3.6}$$

The direction cosine matrix that transforms from the local beam frame w to the body frame can be expressed in terms of the unit vectors of the local beam frame:

$$C^{Bw} = \begin{bmatrix} | & | & | \\ w_x & w_y & w_x \\ | & | & | \end{bmatrix}. \tag{3.7}$$

The transformation from the body to the global frame can be expressed via the quaternion vector:

$$C^{BG} = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}. \tag{3.8}$$

Finally, the kinematic relation between the strain and the boundary node $h_{BC,w}$, and the position of a point on the beam is given by:

$$h_w(s) = e^{K(s-s_0)} h_{BC,w} \tag{3.9}$$

$$= e^{G(s)} h_{BC,w}, \tag{3.10}$$

with

$$K = \begin{bmatrix} 0 & 1+\varepsilon_x & 0 & 0 \\ 0 & 0 & \kappa_z & -\kappa_y \\ 0 & -\kappa_z & 0 & \kappa_x \\ 0 & \kappa_y & -\kappa_x & 0 \end{bmatrix}_{12\times 12} . \tag{3.11}$$

3.1.2 Aerodynamics

While UM/NAST has a software interface to enable coupling with external aerodynamic solvers (see Section 10.2.3 for a description of the interface and its development), by default aerodynamic forces are modeled using strip theory aerodynamics. The vehicles lifting surfaces are subdivided into strips, coinciding with the beam nodes (Figure 3.2). Unsteady wake effects are accounted for using Peters' finite state aerodynamics [53] on every lifting element:

$$\dot{\lambda} = F_1 \ddot{y} + F_2 \dot{y} + F_3 \lambda \tag{3.12}$$

$$= F_1 \begin{bmatrix} \ddot{\varepsilon} \\ \dot{\beta} \end{bmatrix} + F_2 \begin{bmatrix} \dot{\varepsilon} \\ \beta \end{bmatrix} + F_3 \lambda \tag{3.13}$$

with:

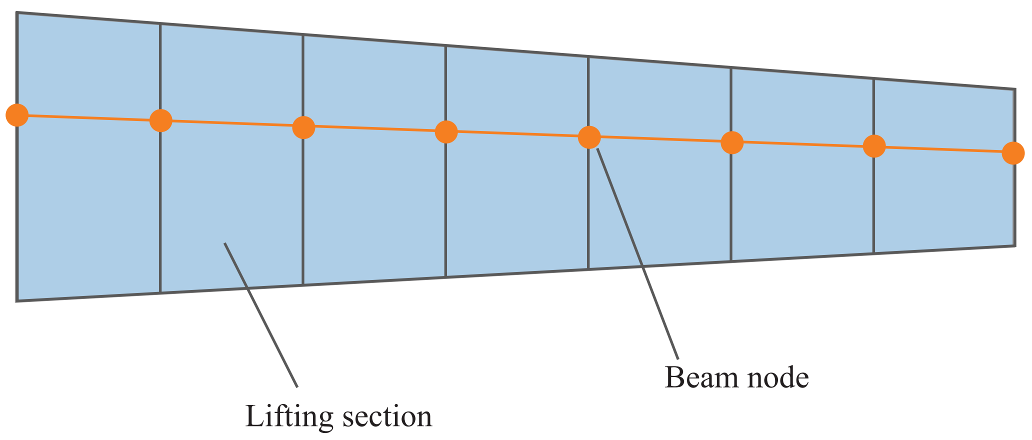


Figure 3.2: Aerodynamics within UM/NAST is accounted for using strip theory with a lifting section to every beam node.

$$\lambda = \begin{bmatrix} \lambda_1 \\ \\ \lambda_2 \\ \\ \vdots \\ \\ \lambda_n \end{bmatrix}, \tag{3.14}$$

where $4 \leq n \leq 8$ usually results in a convergence of the unsteady loads values [4].

The effective angle attack of every aerodynamic section used for the calculation of aerodynamic loads includes contributions from pitching and plunging motion:

$$\alpha_{eff} = \frac{\dot{z}}{\dot{y}} + \left(\frac{1}{2}b_c - d\right)\frac{\dot{\alpha}}{\dot{y}} - \frac{\lambda_0}{\dot{y}}, \tag{3.15}$$

with

$$\lambda_0 = \frac{1}{2} \sum_{i=1}^{N_{inflow}} b_i \lambda_i. \tag{3.16}$$

Finally, the unsteady lift, drag, and pitching moment about the aerodynamic center are, respectively:

$$l_{ac} = \pi \rho b_c^2 \left(-\ddot{z} + \dot{y}\dot{\alpha} - d\ddot{\alpha} \right) + \rho b_c \dot{y}^2 \left(c_l \left(\alpha_{eff} \right) + c_{l\delta} \delta \right) \tag{3.17}$$

$$d_{ac} = -\rho b_c \dot{y}^2 \left(c_d \left(\alpha_{eff} \right) + c_{d\delta} \delta \right) \tag{3.18}$$

$$m_{ac} = \pi \rho b_c^3 \left[\frac{1}{2}\ddot{z} - \dot{y}\dot{\alpha} - \left(\frac{1}{8} b_c - \frac{1}{2} d \right) \ddot{\alpha} \right] + 2\rho b_c^2 \dot{y}^2 \left(c_m \left(\alpha_{eff} \right) + c_{m\delta} \delta \right). \tag{3.19}$$

These unsteady aerodynamic loads include apparent mass effects as well as local lift, drag, and moment due to the effective angle of attack and control surface deflections. The effective angle of attack includes effects of pitching and plunging motions as well as unsteady wake effects.

3.1.3 Full Equations of Motion

The entire coupled aeroelastic system used in UM/NAST is [57, 58]:

$$\begin{bmatrix} M_{FF} & M_{FB} \\ M_{BF} & M_{BB} \end{bmatrix} \begin{bmatrix} \ddot{\varepsilon} \\ \dot{\beta} \end{bmatrix} + \begin{bmatrix} C_{FF} & C_{FB} \\ C_{BF} & C_{BB} \end{bmatrix} \begin{bmatrix} \dot{\varepsilon} \\ \beta \end{bmatrix} + \begin{bmatrix} K_{FF} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \varepsilon \\ b \end{bmatrix} = \begin{bmatrix} R_F \\ R_B \end{bmatrix} \tag{3.20}$$

$$-\frac{1}{2}\Omega_\zeta \zeta = \dot{\zeta} \tag{3.21}$$

$$\begin{bmatrix} C^{GB} \\ 0 \end{bmatrix} \beta = \dot{P}_B \tag{3.22}$$

$$F_1 \begin{bmatrix} \ddot{\varepsilon} \\ \dot{\beta} \end{bmatrix} + F_2 \begin{bmatrix} \dot{\varepsilon} \\ \beta \end{bmatrix} \beta + F_3 \lambda = \dot{\lambda} \tag{3.23}$$

Brown [56] reformulated the governing equations (Equations 3.20–3.23) to obtain a set of first order Ordinary Differential Equations (ODE):

$$Q_1 \dot{y} = Q_2 y + R\left(y, \dot{y}, u, v_g\right) \tag{3.24}$$

In this system of equations, the system states y, control states u, and the nodal gust velocities v_g are:

$$y = \begin{bmatrix} \varepsilon \\ \dot{\varepsilon} \\ \beta \\ \zeta \\ P_B \\ \lambda \end{bmatrix} \tag{3.25}$$

$$u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} \tag{3.26}$$

$$v_g = \begin{bmatrix} v_{g1} \\ v_{g2} \\ \vdots \\ v_{g,m} \end{bmatrix} \tag{3.27}$$

For time-domain solutions, a simple trapezoidal time integration scheme was formulated by Brown [56]. Shearer and Cesnik [57] developed a generalized-α scheme to increase numerical accuracy, while trading off computational expense.

3.2 Linearized Equations of Motion

The linearized ODE in Equation 3.24 can be rearrange to obtain:

$$\dot{y} = Q_1^{-1} Q_2 y + Q_1^{-1} \frac{\partial R}{\partial u} u + Q_1^{-1} \frac{\partial R}{\partial v_g} v_g \tag{3.28}$$

This may be rewritten in simplified matrix-form as:

$$\dot{y} = Ay + Bu + B_w v_g \tag{3.29}$$

These state-space equations can be used for a wide ranging set of tasks, varying from controller design to flutter analyses. For the numerical studies in this work, the matrices B and B_w are neglected, while the A matrix is used for flutter stability analyses.

There are a variety of methods to obtain the linearized matrices. The linearization method used to obtain the state space equations about the geometrical nonlinear deflected condition directly influences the accuracy and efficiency of a flutter analysis or constraint. In past work, Su [4] presented an analytical approach. Later, Pang [61] formulated a finite difference approach to obtain the A matrix, while Dillsaver [59] obtained the B and B_w matrices using finite differences.

The following sections present the existing linearization methods in UM/NAST. In addition to Pang's and Dillsaver's finite difference approach, I developed a complex step and an AD approach for determining the A and B matrices with high accuracy.

3.2.1 Finite Difference Method

In past work, Pang [61] proposed a forward difference approach to obtain Equation 3.29. A perturbation is applied to the state vector y:

$$\tilde{y}^i = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_i + h \\ \vdots \\ y_n \end{bmatrix}. \tag{3.30}$$

Using the perturbed state vector, the perturbed state derivative $\tilde{\dot{y}}^i$ is evaluated. The state space equations can then be determined one column at a time, i.e.,

$$a_i = \frac{1}{\tilde{y}_i} \begin{bmatrix} \tilde{\dot{y}}_1{}^i \\ \tilde{\dot{y}}_2{}^i \\ \vdots \\ \tilde{\dot{y}}_n{}^i \end{bmatrix} \tag{3.31}$$

$$= \frac{1}{\tilde{y}_i}\tilde{y}^i \tag{3.32}$$

with:

$$A = \begin{bmatrix} | & | & & | \\ a_1 & a_2 & \cdots & a_n \\ | & | & & | \end{bmatrix}. \tag{3.33}$$

The forward difference approach, however, is subject to truncation and cancellation errors depending on the perturbation step size chosen. In practice this would require a study of the linearized equations with respect to the step size to determine the accuracy of the linearization. Furthermore, individual columns of the matrix A may be more accurate than others, as different columns may require different step sizes to yield accurate results. Both choosing varying step sizes for different columns of A as well as convergence studies are impractical for optimization problems due to the required user intervention.

3.2.2 Complex Step Method

For this work, I developed a high-accuracy, perturbation-based linearization method using the complex step method (see Section 2.2). Previously, Kitson and Cesnik [64] had applied the complex step method to obtain the linearized Q_1, Q_2, and Q_3 matrices for individual studies. Similar to the finited-difference-based linearization method developed by Pang [61], the state vector is perturbed before evaluating the state derivatives. In this case, the perturbation is imaginary:

$$\tilde{y}^i = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_i + ih \\ \vdots \\ y_n \end{bmatrix}. \tag{3.34}$$

Using the perturbed state vector, the perturbed state derivative $\tilde{\dot{y}}^i$ is evaluated. The state space equations can then be determined one column at a time, i.e.,

$$\begin{aligned} a_i &= \frac{1}{h}\mathrm{Im}\begin{bmatrix} \tilde{\dot{y}}_1^{\ i} \\ \tilde{\dot{y}}_2^{\ i} \\ \vdots \\ \tilde{\dot{y}}_n^{\ i} \end{bmatrix} && (3.35) \\ &= \frac{\mathrm{Im}(\tilde{\dot{y}}^i)}{h}. && (3.36) \end{aligned}$$

Similarly, the B matrix can be determined by first perturbing the control states,

$$\tilde{u}^i = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_i + ih \\ \vdots \\ u_n \end{bmatrix}, \tag{3.37}$$

then determining the state gradients. From this the B matrix is obtained similar to the A matrix:

$$b_i = \frac{1}{h}\mathrm{Im}\begin{bmatrix} \tilde{\dot{y}}_1{}^i \\ \tilde{\dot{y}}_2{}^i \\ \vdots \\ \tilde{\dot{y}}_n{}^i \end{bmatrix} \tag{3.38}$$

$$= \frac{\mathrm{Im}(\tilde{\dot{y}}^i)}{h}. \tag{3.39}$$

As this is a direct application of the complex step method, it shares its properties outlined in Chapter 2, including its high level of accuracy for small perturbations.

3.2.3 Algorithmic Differentiation

Leveraging the AD implementation within UM/ NAST, I developed an AD-based linearization method. The general solution to the state space matrices A and B can be written as:

$$A = \frac{\partial \dot{y}}{\partial y} \tag{3.40}$$

$$B = \frac{\partial \dot{y}}{\partial u} \tag{3.41}$$

This reduces the problem to determining the Jacobian matrix of the state rate function with respect to the states. Given the AD application within UM/NAST, this is easily obtained to machine precision, thereby eliminating the truncation and rounding errors. And because the state rate function only needs to be called once, compared to once for every column for the forward difference and complex step approaches, improvements in computational efficiency are achieved. Finally, for cases in which both the A and B matrices are needed, the computation experience further performance gains, as the state velocities function is called once for both matrices instead of for every matrix column when using the perturbation-based methods.

3.2.4 Semi-Analytical

Finally, the AD approach to the linearization problem can be combined with an analytical approach to obtain a semi-analytical solution. An analytical solution to the linearized matrices was proposed by Su [58]. As shown previously in Equation 3.28, the linearized equations of motion are:

$$\dot{q} = Q_1^{-1} Q_2 q + Q_1^{-1} \frac{\partial R}{\partial u} u + Q_1^{-1} \frac{\partial R}{\partial v_g} v_g, \tag{3.42}$$

with:

$$A = Q_1^{-1} Q_2. \tag{3.43}$$

The matrices Q_1 and Q_2 are:

$$Q_1 = \begin{bmatrix} I & 0 & 0 & 0 & 0 & 0 \\ 0 & \bar{M}_{FF} & \bar{M}_{FB} & 0 & 0 & 0 \\ 0 & \bar{M}_{BF} & \bar{M}_{BB} & 0 & 0 & 0 \\ 0 & 0 & 0 & I & 0 & 0 \\ 0 & 0 & 0 & 0 & I & 0 \\ 0 & -F_{1F} & -F_{1B} & 0 & 0 & I \end{bmatrix} \tag{3.44}$$

$$Q_2 = \begin{bmatrix} 0 & I & 0 & 0 & 0 & 0 \\ -\bar{K}_{FF} & -\bar{C}_{FF} & -\bar{C}_{FB} & R^{grav}_{F/\zeta_0} & 0 & R^{aero}_{F/\zeta_0} \\ 0 & -\bar{C}_{BF} & -\bar{C}_{BB} & R^{grav}_{B/\zeta_0} & 0 & R^{aero}_{B/\zeta_0} \\ 0 & 0 & -\frac{1}{2}\Omega_{\zeta/\beta_0}\zeta_0 & -\frac{1}{2}\Omega_{\zeta} & 0 & 0 \\ 0 & 0 & [\ C^{GB} \quad 0\ ] & [\ C^{GB}_{/\zeta_0} \quad 0\ ]\beta_0 & 0 & 0 \\ 0 & F_{2F} & F_{2B} & 0 & 0 & F_3 \end{bmatrix} \tag{3.45}$$

The sub-matrices used within Q_1 and Q_2 are:

$$\bar{K}_{FF} = K_{FF} - J^T_{p\varepsilon} B_F \frac{\partial F^{aero}}{\partial \varepsilon} - J^T_{\theta\varepsilon} B_M \frac{\partial M^{aero}}{\partial \varepsilon} \tag{3.46}$$

$$\bar{M}_{FF} = M_{FF} - J_{p\varepsilon}^T B_F \frac{\partial F^{aero}}{\partial \ddot{\varepsilon}} - J_{\theta\varepsilon}^T B_M \frac{\partial M^{aero}}{\partial \ddot{\varepsilon}} \tag{3.47}$$

$$\bar{M}_{FB} = M_{FB} - J_{p\varepsilon}^T B_F \frac{\partial F^{aero}}{\partial \dot{\beta}} - J_{\theta\varepsilon}^T B_M \frac{\partial M^{aero}}{\partial \dot{\beta}} \tag{3.48}$$

$$\bar{M}_{BF} = M_{BF} - J_{pb}^T B_F \frac{\partial F^{aero}}{\partial \ddot{\varepsilon}} - J_{\theta b}^T B_M \frac{\partial M^{aero}}{\partial \ddot{\varepsilon}} \tag{3.49}$$

$$\bar{M}_{BB} = M_{BB} - J_{pb}^T B_F \frac{\partial F^{aero}}{\partial \dot{\beta}} - J_{\theta b}^T B_M \frac{\partial M^{aero}}{\partial \dot{\beta}} \tag{3.50}$$

and

$$\bar{C}_{FF} = C_{FF} - \frac{\partial C_{FF}}{\partial \dot{\varepsilon}} \dot{\varepsilon}_0 - \frac{\partial C_{FB}}{\partial \dot{\varepsilon}} \beta_0 - J_{p\varepsilon}^T B_F \frac{\partial F^{aero}}{\partial \dot{\varepsilon}} - J_{\theta\varepsilon} B_M \frac{\partial M_{aero}}{\partial \dot{\varepsilon}} \tag{3.51}$$

$$\bar{C}_{FB} = C_{FB} - \frac{\partial C_{FF}}{\partial \beta} \dot{\varepsilon}_0 - \frac{\partial C_{FB}}{\partial \beta} \beta_0 - J_{p\varepsilon}^T B_F \frac{\partial F^{aero}}{\partial \beta} - J_{\theta\beta} B_M \frac{\partial M_{aero}}{\partial \dot{\varepsilon}} \tag{3.52}$$

$$\bar{C}_{BF} = C_{BF} - \frac{\partial C_{BF}}{\partial \dot{\varepsilon}} \dot{\varepsilon}_0 - \frac{\partial C_{BB}}{\partial \dot{\varepsilon}} \beta_0 - J_{pb}^T B_F \frac{\partial F^{aero}}{\partial \dot{\varepsilon}} - J_{\theta b} B_M \frac{\partial M_{aero}}{\partial \dot{\varepsilon}} \tag{3.53}$$

$$\bar{C}_{BB} = C_{BB} - \frac{\partial C_{BF}}{\partial \beta} \dot{\varepsilon}_0 - \frac{\partial C_{BB}}{\partial \beta} \beta_0 - J_{pb}^T B_F \frac{\partial F^{aero}}{\partial \beta} - J_{\theta b} B_M \frac{\partial M_{aero}}{\partial \beta} \tag{3.54}$$

Of these sub-matrices, most quantities are analytical, with the exception of the aerodynamic derivatives. In past work, Su [58] derived analytical representations for the aerodynamic derivatives, provided that derivatives of the aerodynamic coefficients were known. This required the user to provide such derivatives, which eliminated some aerodynamic methods (such as the method of segments or UVLM) from being linearized with the analytical formulation. The semi-analytical formulation I developed here replaces the aerodynamic derivatives provided by the user with ones determined at code run time using AD. The derivatives determined in this manner are listed in Table 3.1.

Once the aerodynamic derivatives have been obtained using AD, the linearized A matrix can be determined. The advantage of this approach over the purely analytical

Table 3.1: Partials required for the (semi-)analytical linearization. Values for which analytical representations exist are marked by ○, while values determined using AD are marked by ●.

	$\partial\varepsilon$	$\partial\dot{\varepsilon}$	$\partial\ddot{\varepsilon}$	$\partial\beta$	$\partial\dot{\beta}$
∂F^{aero}	●	●	●	●	●
∂M^{aero}	●	●	●	●	●
∂C_{FF}		○		○	
∂C_{FB}		○		○	
∂C_{BF}		○		○	
∂C_{BB}		○		○	

method lies in the ability to linearize systems with aerodynamic models lacking derivatives. Compared to the perturbation-based methods, the semi-analytical approach may produce significantly better computational performance at high accuracy. The pure-AD linearization already outperforms these methods. The semi-analytical approach requires less code to be differentiated and evaluations of the chain rule, thereby providing further performance improvements compared to the pure-AD method. At the time of this writing, the semi-analytical approach has been partially implemented and remains to be completed. Code for the analytical terms exists from previous UM/NAST versions and was ported to the reorganized framework. The methods for determining the partials of the aerodynamic loads has been started, but not completed.

CHAPTER 4

Flutter Analysis and Interpretation

Flutter analyses constitute a vital part of any aircraft design process and are required for certification and operational safety. Traditionally, geometrically linear flutter algorithms have been used during vehicle design and certification. For advanced aircraft configurations, geometrically nonlinear flutter analyses are increasingly necessary as vehicles become more flexible. Geometrically nonlinear flutter problems, however, have been applied and interpreted akin to linear flutter analyses, trading one “black box” for another, while adding additional problem variables. As such, important consequences of nonlinear flutter analysis on the instability search process, interpretation, and visualization remain unanswered.

This chapter presents a generalized interpretation of flutter problems, unifying linear and nonlinear flutter problems (Section 4.1). In this context, the section describes visualization techniques for both linear and nonlinear analyses, discusses the limitations of those methods, and adds additional tools to analyze nonlinear flutter. Finally, Sections 4.3 and 4.4 present methods for the efficient determination of the flutter boundary for geometrically nonlinear structures.

4.1 Generalized Interpretation of Flutter Problems

Before discussing the interpretation of linear and nonlinear flutter problems, I should note that the logical progression within this section may seem counterintuitive. Linear flutter analyses are specialized forms of nonlinear flutter problems. However, historically, linear flutter methods were developed before their nonlinear counterparts. As a result, the interpretation of linear problems was well established when nonlinear flutter problems became necessary. Moreover, these nonlinear problems were then interpreted similarly to linear problems using root loci, etc. This led to the neglect of some ramifications of the nonlinear problems. As such, a progression from the general problem to its specialization is not practical here. Rather, I will present prevalent methods of interpreting linear problems before discussing the changes and caveats required by nonlinear problems.

For both types of problems, the dynamics of the entire system can be written as:

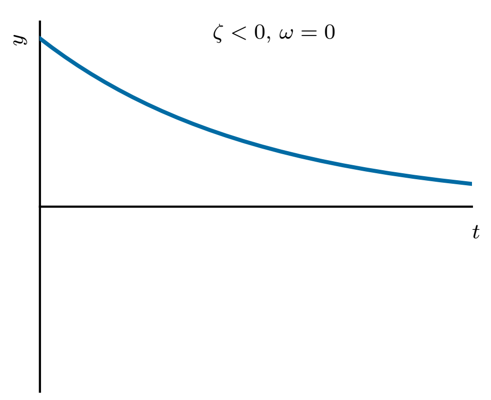
$$y\left(x,t\right)=\sum_{i=1}^{N}\phi_i\left(x\right)f_i\left(t\right), \tag{4.1}$$

where $\phi_i\left(x\right)$ is a linearized mode shape about the equilibrium that only depends on the spatial coordinate x, and $f_i\left(t\right)$ is the time-dependent component of the i-th mode. It is worth noting that Equation 4.1 holds true for continuous systems (in which case $N=\infty$), as well as computational models, which tend to be finite by truncating higher modes. Moreover, for stability solutions a general solution of the form exists:

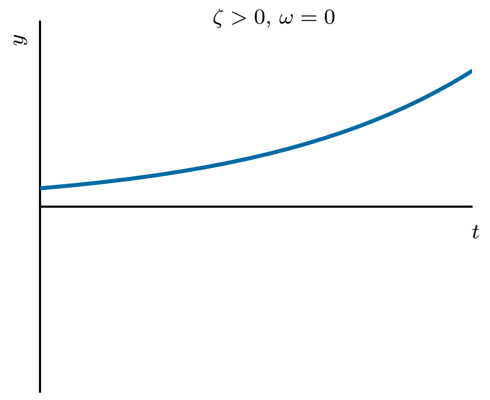
$$f\left(t\right)=e^{(\zeta+i\omega)t} \tag{4.2}$$

$$=e^{at} \tag{4.3}$$

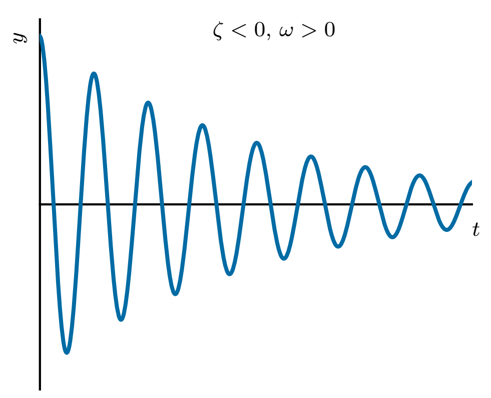
The stability of the system is determined by the real part of a (Figure 4.1). If ζ is negative, the system is stable. A positive value indicates an instability, while $\zeta=0$ constitutes the stability boundary, or the onset of the instability. Whether the instability is oscillatory or not is determined by the imaginary part of a. A non-zero



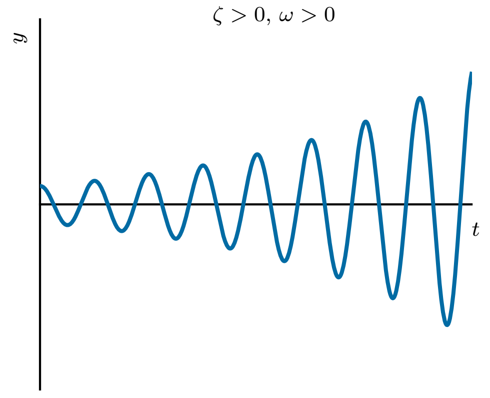
(a) Non-oscillatory, stable



(b) Non-oscillatory, unstable (divergence)



(c) Oscillatory, stable



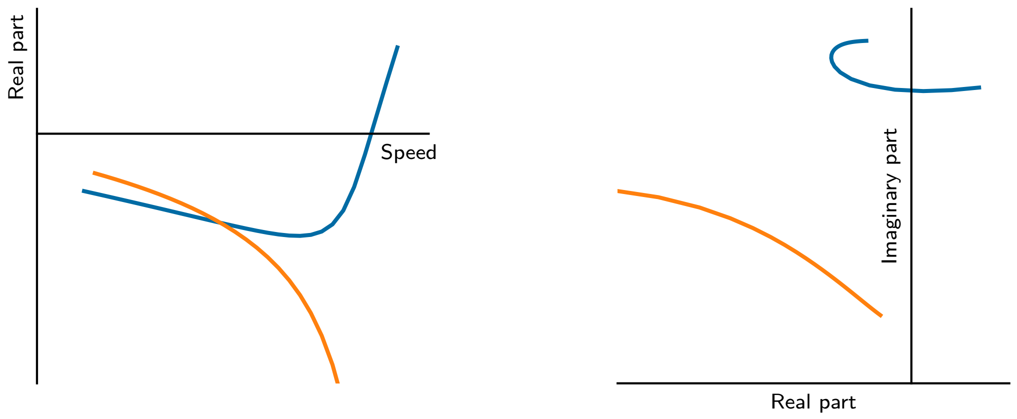
(d) Oscillatory, unstable (flutter)

Figure 4.1: Time component of the stability solution depending on the sign of the respective eigenvalue parts.

value for ω will result in an oscillatory instability such as flutter, while a zero value will result in non-oscillatory instabilities such as divergence.

4.1.1 Interpretation of Linear Flutter Analyses

Flutter analyses generally seek to find the flutter boundary. This can be achieved by evaluating the stability problem along a line which is defined by the flutter search variable. This variable may be the flight speed or the dynamic pressure of the vehicle (a more detailed explanation of the search variable is presented in Section 4.3 and in



(a) V-g diagrams. (b) Root locus diagram

Figure 4.2: Examples of V-g and root locus diagrams (showing two modes) typically used during linear flutter analyses.

Figure 4.4). While different solution strategies exist, linear flutter solutions fundamentally constitute a mapping of $\mathbb{R} \rightarrow \mathbb{R}^N$; from the search variable to the resulting set of modes (where N is the number of modes retained for the analysis).

Traditional V-g (Figure 4.2a) diagrams are a two-dimensional representation of this univariate search and lend themselves naturally to the interpretation of the linear flutter problem. The V-f diagram presents the frequency (or the imaginary part of the flutter analysis) progression of the N modes as a function of the flutter search variable. Similarly, the V-g diagram provides the progression of the modes' damping progression. From it one can determine the flutter point (classically called the flutter boundary) as well as deduce the severity of the flutter onset. The flutter point is the crossing of a mode with the V-axis. For linear flutter problems, the severity of the flutter onset is often quantified by the slope of the mode curve at the flutter point. A steeper slope indicates a more violent onset while a more moderate slope translates to a more gradual one.

Root-locus diagrams (Figure 4.2b) also find wide-spread use when interpreting linear flutter problems. They combine the real (x-axis) and imaginary (y-axis) parts of the N retained modes into one diagram. Instabilities are determined by crossings of the y-axis (real part is zero). Each point in a root locus diagram corresponds to an eigenvalue at an individual linearization evaluation point. The mode progressions still correspond to a sequence along the search variable, although the values of the search variable may be more difficult to discern than in the V-g diagram. Similar to the V-g representation, the severity of the flutter onset may be determined by the angle at the flutter point (or the rate of damping change due to a change in the search parameter). In some representations, the search variable value is indicated using a color map on the mode lines. Because it contains a more explicit representation of information, the V-g/V-f diagrams will be used for interpretative purposes within this chapter. However, the statements made regarding these diagrams remains applicable to the root locus representation as well as the flutter problem as a whole.

4.1.2 Interpretation and Visualization of Nonlinear Flutter Problems

V-g and root-locus diagrams constitute a useful tool during linear flutter analyses to determine the flutter point and interpret results. While the use of flutter analyses including geometrical nonlinearities has increased over time, the questions of how to visualize and interpret the results has remained unanswered. Consequently, the solutions to these nonlinear problems have been interpreted using traditional root-locus or V-g diagrams. However, flutter problems including geometrical nonlinearities, differ significantly from their linear counterparts and require a different interpretive approach.

When including geometrical nonlinearities, the flutter search is no longer univariate as the angle of attack, control surface deflections, thrust level, etc. may change the static equilibrium and result in different structural modes and linearized flutter modes. The number of search variables now depends on the vehicle conditions encountered as well as the aircraft configuration itself, expressed as the mapping of $\mathbb{R}^M \to \mathbb{R}^N$ (where M is the number of variables affecting the flutter stability).

The dimension of the flutter search space may be very large and prohibitive for visualization. While the problem illustrated in Equation 4.4 constitutes a multidimensional root-finding problem, solving this with a multidimensional Newton's method would yield different results for different starting conditions, as illustrated in Figure 4.3. Fortunately, the flutter search can still be performed in a pseudo-univariate manner, akin to a line search along a one-dimensional curve through a higher dimensional space (Figure 4.3). That is, for example, the dynamic pressure is chosen as the search variable, while the other variables (angle of attack, control surface deflections, etc.) are determined by the dynamic pressure, e.g., because of trim requirements:

$$\mathrm{Re}\left(\zeta_i\left(q, \alpha, \eta_{CS}, \ldots\right)\right) = 0 \tag{4.4}$$

with:

$$\begin{aligned} \alpha &= f\left(q\right) && (4.5) \\ \eta_{CS} &= g\left(q\right) && (4.6) \\ &\vdots && (4.7) \end{aligned}$$

The results of this pseudo-univariate search, similar to a linear flutter problem, can be visualized using a root-locus or V-g diagram. Differing from the linear analysis, the resulting V-g diagram is the result of a series of cuts through a much higher dimensional flutter search space, thereby only representing information along the search line.

Data from flutter search locations in two dimensions can also be combined, yielding an extended V-g diagram (Figure 4.3). The axes constitute the input variables/flight

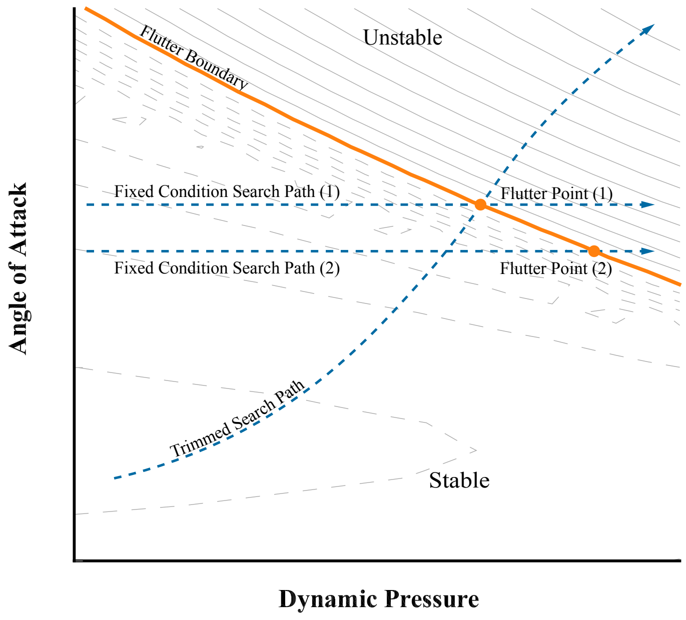


Figure 4.3: A qualitative extended V-g diagram (angle of attack vs. dynamic pressure) for the first mode damping values with different flutter search paths. The contours denote the damping values of the mode closest to zero. Dashed contour lines indicate negative damping values.

conditions for the flutter analysis. For legibility the damping values from only one mode are displayed as contour lines; multiple modes may be visualized using a multitude of diagrams. The extended V-g diagram contains all information of a traditional V-g diagram but extended by another axis (here: angle of attack). While the classical V-g or root-locus diagram contains information regarding the flutter point and some indication of hardness of the crossing (along the search direction), the insights gleaned from the diagram may remain incomplete. The extended diagram, by contrast, permits additional correlations regarding the sensitivity to changes in flight conditions. For two flutter variables (e.g., dynamic pressure and angle of attack), the diagram represents the entire search space. However, if there are more than two search variables, the extended V-g diagram also only captures a snapshot of the search space and additional extended diagrams (with other variables) may be needed to decipher trends.

Finally, the extended V-g diagram illustrates the effects of the path choice of the flutter search process. As previously discussed and depicted in Figure 4.3, two fundamental search processes exist: trimmed conditions and pre-determined conditions for a given dynamic pressure. In Figure 4.3, the trimmed search path shares the same flutter point with the first fixed condition search path. Despite sharing the flutter point along their respective search paths, the traditional V-g diagrams for each search path differ substantially. Strikingly, the gradient along the search direction, often used to indicate severity of the flutter onset, is different for the two searches. As a result, the information obtained from a traditional V-g diagram along the search direction (except for the flutter point itself) must be weighed judiciously and if appropriate, be supplemented with extended V-g diagrams and gradient information. Finally, in the case of pre-determined vehicle conditions, the choice of search path may have a significant effect on the predicted flutter point (Figure 4.3). As such, trim points, if available, are preferable as a flutter search path and vehicle conditions must be chosen carefully for aircraft configurations without trim data and may result in accelerated conditions.

As previously stated, linear analyses are a special case of nonlinear flutter search problems. Inspecting the extended V-g diagram in Figure 4.3, a line coincident with the dynamic pressure axis constitutes a linear flutter search. In this case, the angle of attack does not influence the flutter damping values or the location of the flutter point. The choice of a linear vs. a nonlinear flutter analysis, therefore must be decided by the expected dimensionality of the flutter problem.

4.2 Flutter Search Including Geometrical Nonlinearities

The flutter algorithm developed in this work consists of two parts: the search process and the postprocessing of the search data (Figure 4.6). The search process follows the methodology outlined in Su and Cesnik [4] and was implemented in UM/NAST (see Chapters 3 and 10). A search point evaluation begins with the evaluations of the steady state solution for given boundary conditions (Mach number, dynamic pressure, angle of attack, angle of sideslip, etc.). A linearization is then performed about this nonlinear state of equilibrium to obtain the system matrix A, such that:

$$\dot{y} = Ay \tag{4.8}$$

The stability check is performed by an eigenvalue analysis of the system matrix, resulting in eigenvalues of the form:

$$a_i = \zeta_i + i\omega_i, \tag{4.9}$$

from which the stability of a mode can be determined from the real part of the eigenvalue. The spatially dependent mode shape ϕ_i (Equation 4.1 can be determined from the eigenvectors of the A matrix.

The advantage of Su and Cesnik's approach is that several different modes of stability analysis can be conducted such as flutter of a vehicle with constrained rigid body motion, flutter of free flying vehicles, and the vehicle flight dynamic stability. While the proposed algorithm uses the same fundamental formulation, it differs from Su and Cesnik's methodology by not iterating the search variable according to the result of the eigenvalue analysis; Su and Cesnik's algorithm performs a check after every eigenvalue analysis and only increases the velocity until an instability occurs. The proposed algorithm requires a set of search points to be determined a priori. This results in the complete set of iterations being completed irrespective of the other search iterations. This data independence makes the algorithm highly parallelizable and large improvements in runtime can be achieved by using parallel programming paradigms such as MPIMessage Passing Interface (MPI). The independence of the individual search points can also be used to sample the flutter space and create an extended V-g diagram or determine the flutter boundary (see Section 4.4).

4.3 Determining a Flutter Point

This section details the flutter point algorithm developed for this work. Using the search data described in Section 4.2, the flutter point is determined via postprocessing. To this end, a mode tracking algorithm is applied and an interpolation surrogate created. Finally, a hybrid root-finding algorithm is applied to the surrogate to find the location of the stability boundary.

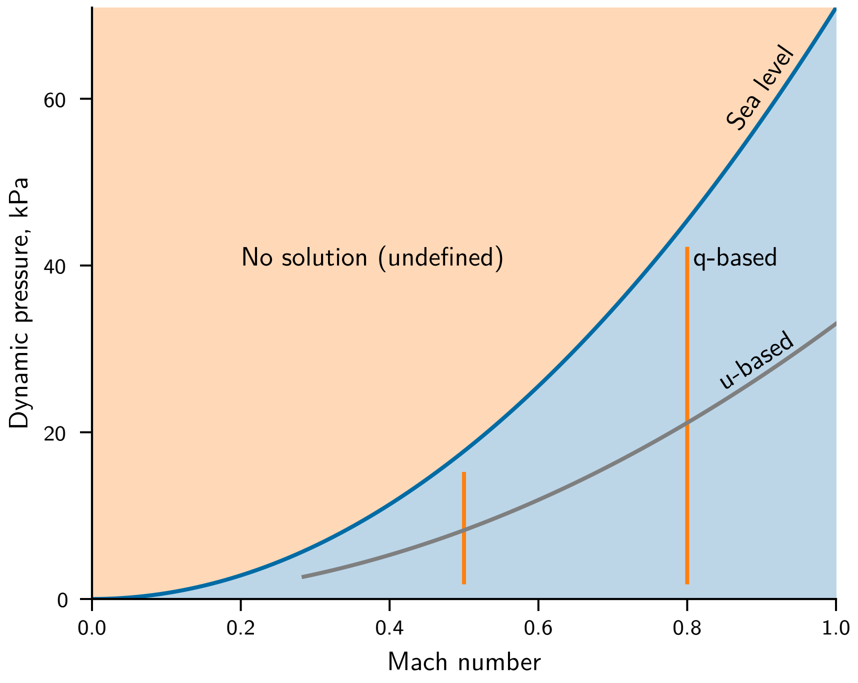


Figure 4.4: Different flutter search types in a modeled atmosphere.

4.3.1 Choosing the Search Variable

Two fundamental search types exist to determine the flutter point: a speed-based (u-based, with fixed altitude) or dynamic pressure-based (q-based, with fixed Mach number) search. While, at first glance, these methods may appear equivalent, the choice in search method entails practical consequences depending on the application.

Consider the search lines shown in the Mach-dynamic pressure plane in Figure 4.4. The atmospheric model, with the International Standard Atmosphere (ISA) shown here, naturally imposes bounds on the choice of search variable: the lower altitude is limited by sea level, and the edge of space imposes an upper limit. In numerical models, this upper bound may be lower than the edge of space due the altitude restrictions on standard atmospheres. As a result, the blue area in Figure 4.4 represents the space in which all physically feasible searches can be conducted. The two different search types are displayed: a u-based search for a fixed altitude, as well as two different q-based searches for fixed Mach numbers.

The Mach-dynamic pressure plane is a significant representation for flutter analyses for a variety of reasons. Often, flutter boundaries are displayed as the flutter dynamic pressure as a function of the Mach number. Additionally, the aerodynamic data used for flutter analyses is often provided as a function of the Mach number. As such, the relationship of the flutter search variable to the M-q space is relevant to obtain accurate results.

The u-based search, while fixing an altitude, varies in Mach number. As a result, if the aerodynamic data is given only until a particular Mach number (e.g., 0.8), extrapolation of the aerodynamic coefficients may occur without the knowledge of the user. The errors incurred by the extrapolation process may be significant and render the resulting flutter analysis useless. The q-based search, by constrast features a fixed Mach number, so that extrapolation cannot be encountered unless the user explicitly requests it. Moreover, the dynamic pressure-based search permits the evaluation at the seed points of the aerodynamic data, thus reducing the error of the aerodynamic coefficient interpolation (i.e., via kriging surrogates in the method of segments [65]). Finally, the speed-based search is not bounded beyond the requirement that $u \geq 0$. The speed can be increased infinitely. The dynamic pressure search is bounded by the limits of the atmospheric model used. As a result, the search space at low Mach numbers is very small and may yield very high accuracy results with few search points, while higher Mach numbers entail a much larger search range.

4.3.2 Mode Tracking

The mode tracking algorithm developed for this work functions based on a modal assurance criterion (MAC)-based approach [66, 67]. The MAC is typically used in ground vibration testing (ground vibration testing (GVT)) to compare the similarity of mode shapes from two different sources (i.e., experiment vs. numerical simulation).

In its general form theMAC compares two vectors ϕ_i and ϕ_j which may be real and complex, and is given by:

$$MAC = \frac{\left|\phi_i^H \phi_j\right|^2}{\phi_i^H \phi_i \phi_j^H \phi_j} \tag{4.10}$$

The superscript H indicates the Hermitian or the conjugate transpose vector, which simplifies to the transpose for real vectors. The implemented mode tracking algorithm is described in Figure 4.5. The algorithm conducts a discrete optimization to find the modes which match best according to the MAC. As the algorithm is inherently serial, matched modes are eliminated from the pool of comparison modes for computational efficiency.

4.3.3 Root-Finding Algorithm

Several root-finding algorithms exist varying from bracketing techniques (bisection, false position, etc.) to fixed-point iterations (such as Newton's method). Each method offers benefits and drawbacks regarding solution speed, accuracy, and robustness. As such, a hybrid root-finding method was used for this work combining the advantages of the bisection method and Newton's method.

Newton's method is defined by [68]:

$$x_{i+1} = x_i - \frac{f\left(x\right)}{f'\left(x\right)} \tag{4.11}$$

or written as the fixed-point iteration:

$$g\left(x\right) = x - \frac{f\left(x\right)}{f'\left(x\right)} \tag{4.12}$$

As a fixed-point iteration, it fulfills:

$$g\left(p\right) = p \tag{4.13}$$

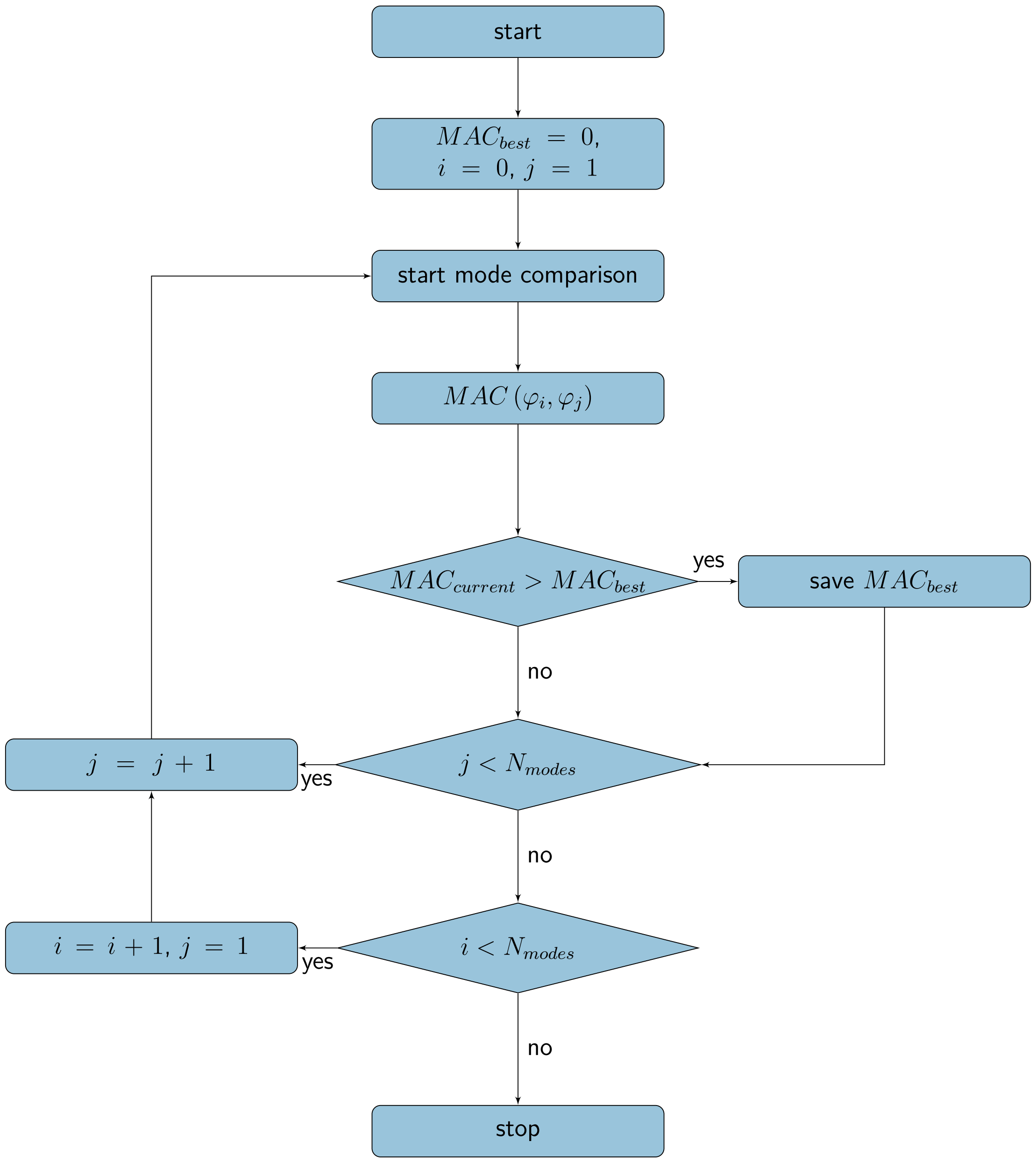


Figure 4.5: Flow chart of the implemented mode tracking algorithm.

where p is the root of the function f. Fixed-point iterations must converge to the fixed point/root if the following criteria are met (for a detailed proof, see Bradie [69]):

1. g is continuous on the closed interval $[a, b]$.
2. g is differentiable over the open interval (a, b).
3. g' is continuous on the open interval (a, b).
4. $|g'(x)| \leqslant k < 1$ for a positive constant k.

If these conditions are not met, convergence of Newton's method is not guaranteed. If, on the other hand, these conditions are fulfilled, Newton's method converges with $\mathcal{O}(n^2)$.

On the other hand, following from the Intermediate Value Theorem (IVT) [69], the bisection method converges to a root within a closed interval (at approx. $\mathcal{O}(n)$) if:

1. The function f is continuous over the closed interval $[a, b]$.
2. $f(a)$ and $f(b)$ have opposite signs.

Clearly, the uncertainty regarding Newton's method convergence and the risk of divergence from the root disqualify it from being used without modification to find the root (flutter point), while the convergence characteristics of the bisection method entails a performance penalty. Therefore, the root-finding algorithm used in this work combines the bisection method and Newton's method to ensure converge to a root, should it exist (follows from the IVT), while enabling the same convergence speed (or close) of Newton's method. The hybrid algorithm conducts the following steps:

1. Conduct an initial bisection iteration on the interval to obtain x_1 (x_i for $i = 1$).
2. Conduct a Newton iteration using x_i to obtain x_{i+1}

3. If x_{i+1} lies within the interval
    (a) keep x_{i+1}.
4. Else
    (a) Conduct a bisection step to obtain x_{i+1} and adjust the interval bounds appropriately.
5. Increase i and repeat from 3. until convergence tolerance is achieved.

4.3.4 Flutter Point Algorithm

The second part of the newly developed algorithm is the postprocessing of the data obtained during the search process to find potential instabilities. This part contains the largest deviation from existing methodologies and is agnostic to the type of aeroelastic model (beam-based or full FEM).

Mode tracking is applied to solutions of the EVP to obtain the eigenvalues and eigenvectors as a function of dynamic pressure. Next, the first unstable mode is determined. A surrogate of this unstable mode is created from the resulting eigenvalues (separating real and complex parts) and the surrogate roots determined.

The solution to this root finding problem is solved using the hybrid fixed-point/ bisection method described previously, yielding the flutter point (flutter dynamic pressure). A final flutter search iteration is conducted using the flight conditions at the flutter point (dynamic pressure, control surface deflections, angle of attack, etc.) to obtain the flutter frequency and flutter mode shape.

To improve computational efficiency the number of retained modes is reduced for the first search iteration using filters (to remove inflow dominated modes, rigid-body modes, etc.) prior to applying the mode tracking algorithm. By removing these modes from the initial set, the filtering propagates across all search iterations while reducing the number of mode comparisons and thereby improving computational efficiency.

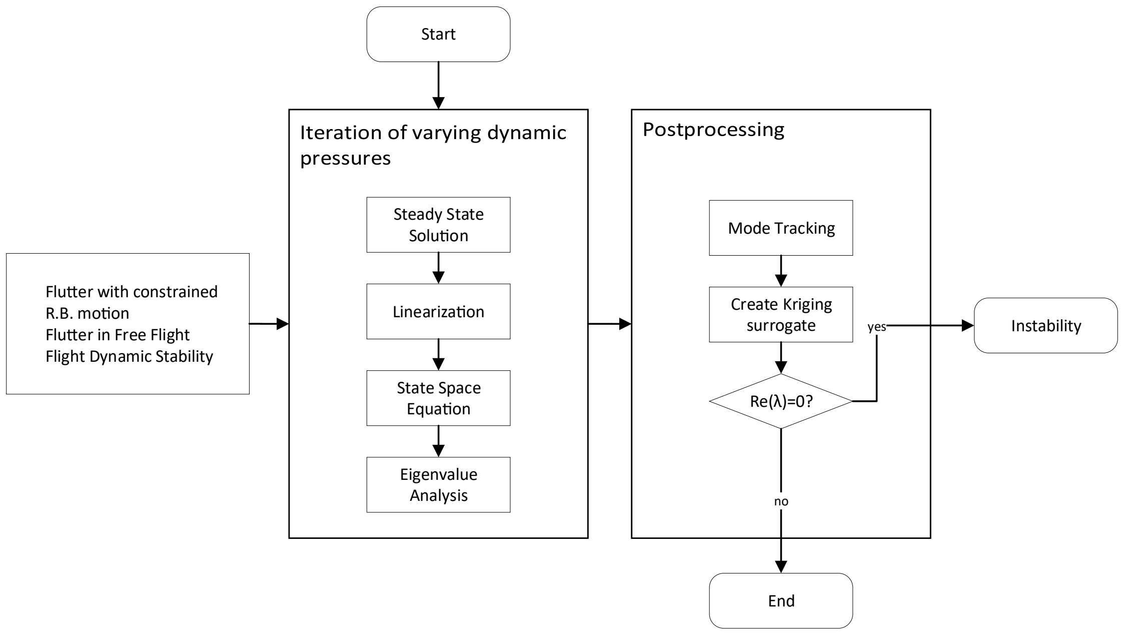


Figure 4.6: Algorithm utilizing mode-tracking and kriging surrogates for accurate prediction of the flutter point including geometrical nonlinearities.

4.4 Determining the Flutter Boundary

The typical method for determining a flutter boundary involves running a predefined number of flutter analyses to determine their respective flutter points. While this is simple and computationally feasible for linear analyses, this approach may be computationally less than optimal for nonlinear flutter analyses. As such, a more efficient approach to determining flutter boundaries was developed for this work. Although nonlinear flutter problems constitute the focus of this work, the underlying methodology remains applicable to linear problems as well.

First, the concept of a flutter boundary, as it is discussed here, is clarified. The stability problem can be expressed as a linear map between the variables affecting the flutter problem and the critical damping value:

$$\mathbb{R}^N \to \mathbb{R} \tag{4.14}$$

or

$$\mathbb{R}^N \to \max\left(\zeta_i\right) \tag{4.15}$$

Therefore, in general terms, the flutter boundary is the map of the flutter variables to the neutrally stable point:

$$\mathbb{R}^N \to 0 \tag{4.16}$$

However, flutter boundaries are usually visualized for interpretation and visualizations beyond three dimensions are inpractical. Therefore, this section will limit the determination of the flutter boundary to within the extended V-g diagram or or a projection from two flutter variables to the critical damping value:

$$\mathbb{R}^2 \to \max\left(\zeta_i\right) \tag{4.17}$$

4.4.1 Non-Adaptive Search

As previously discussed, evaluating several flutter point analyses in series to determine the flutter boundary is computationally inefficient. Such analyses are analogous to sampling a domain using a rectilinear grid. While this may work, it typically is an ineffective method of sampling. Instead of determining individual flutter points, this method uses non-structured sampling (see Chapter 6) combined with a surrogate to determine the flutter boundary.

To create an extended V-g diagram, one must first sample the flutter problem. If a flutter boundary over one variable is sought, the sampling may be conducted in two dimensions instead of sampling all flutter variables. The flutter boundary constitutes the roots of the diagram. Differing from the flutter point analysis, however, the flutter boundary is a line in the diagram instead of a point. Therefore, a root finding process must be run several times, using multiple starting points x_k to (hopefully) obtain unique roots. Once the samples have been created, a interpolation surrogate is created which is used instead of the search point evaluation during the root finding process. The flutter boundary algorithm uses the extended V-g sample points as starting points for a two-dimensional Newton method. Because the flutter boundary search is conducted solely using the surrogate, substantial performance gains are obtained compared to a search using the flutter search point analyses directly.

Predicting the flutter boundary in this manner also yields more information than the typical evaluation. As with the flutter point search, due to the use of a surrogate, the accuracy of the flutter boundary prediction can be quantified using the prediction variances. This permits the designer or researcher to quantify the confidence of their prediction and refine the search if necessary. Additionally, the extended V-g can be visualized on top of the flutter boundary, providing a designer more information as to the nature of the instability.

4.4.2 Additional Remarks

As previously discussed, the methods presented here function on the three-dimensional projection from two flutter variables to the critical damping value (Equation 4.17). Practical nonlinear flutter problems, however, may depend on many additional variables and evaluating the flutter boundary with respect to several variables may be required. It should be noted that this scenario does not require a re-sampling of the flutter problem, which would be computationally expensive. Instead, the initial sampling needs to be conducted such that it sufficiently samples the multi-dimensional space of the flutter problem. This will likely be significantly more expensive than sampling in two dimensions, due to the so-called “curse of dimensionality.” However, once the sampling has been conducted, the multi-dimensional data set must be reduced to two dimensions, so that the extended V-g diagram can be created:

$$\mathbb{R}^N \to \mathbb{R}^2 \to \max\left(\zeta_i\right) \tag{4.18}$$

Such a reduction must be conducted for every flutter boundary that will be created. This reduction in dimensions is cheap, as it only entails excluding dimensions before transferring data to the flutter boundary search algorithm. Therefore, the determination of the flutter boundary remains the same, while the flutter search point sampling is expanded.

CHAPTER 5

Geometrically Nonlinear Flutter Constraint

As vehicles become more flexible and optimization finds wide-spread use, flutter constraints including geometrically nonlinear effects are required. These constraints ensure that flutter can be considered early in the design process, reducing or eliminating costly modifications during the final detailed design or certification.

While linear flutter analyses and constraints account for much of the existing literature, Henshaw et al. [70] noted that the use of linear flutter constraints may yield a conservative design. In contrast to linear flutter analyses, nonlinear flutter problems depend on initial conditions (see Section 4.1 for a detailed explanation). Formulating a flutter constraint including geometric nonlinearities therefore requires a much larger analysis space than a linear flutter constraint. Therefore, an excessive number of constraints may be obtained unless constraint aggregation is used.

This chapter discusses the geometrically nonlinear flutter constraint I formulated for this work. Section 5.1 discusses the considerations of constraining either the flutter dynamic pressure or the flutter search damping values. As constraint aggregation is required to ensure an efficient optimization problem, Section 5.2 reviews constraint aggregation methods and details the properties of the individual methods as well as their applicability to the current problem. Finally, Section 5.3 presents the formulation of the geometrically nonlinear flutter constraint used in this work, based on a similar methodology developed by Jonsson et al. [71, 34].

5.1 Constraining Damping vs. Dynamic Pressure

Multiple approaches exist to creating a flutter constraint for optimization. One approach is to constrain the flutter dynamic pressure such that it remains above a defined threshold:

$$q_{limit} - q_F < 0, \tag{5.1}$$

for all i.

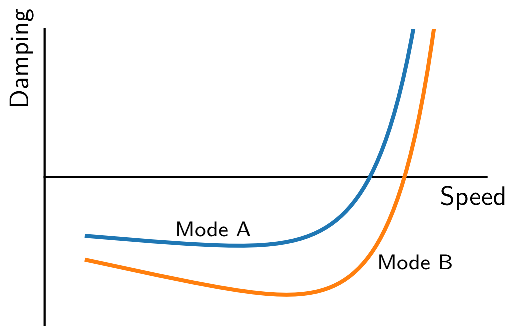
This formulation, however is not practical, as hump modes or mode switching may yield discontinuities in the constraint and constraint gradient (Figure 5.1). As such, the flutter constraint is formulated as a constraint of the real part (damping) of the eigenvalues from the flutter search such that:

$$\operatorname{Re}(\zeta_i) < 0 \tag{5.2}$$

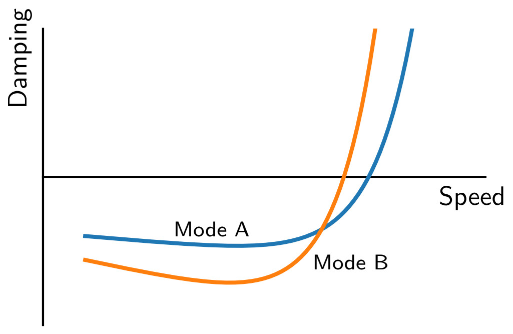
for all i.

5.2 Aggregation Methods

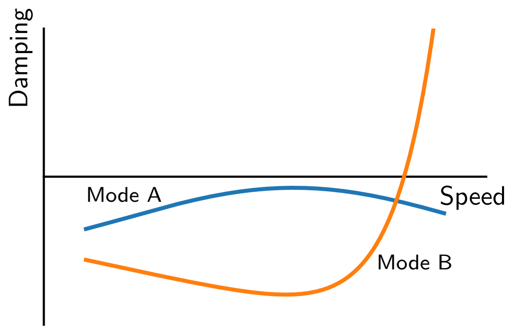
Large number of constraints can increase the cost of an optimization problem and cause its solution to become untenable. Constraint aggregation combines multiple constraints into one scalar value, attempting to find a conservative approximation for the largest constraint value $g_{max} = max\left(g_j\right)$. Ideally, the aggregation method used avoids excessively conservative approximations, as this can yield an entirely infeasible



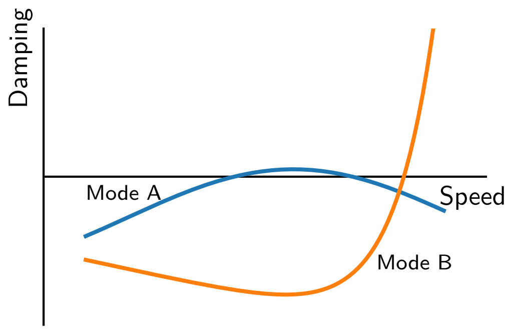
(a) Mode-switching, configuration A



(b) Mode-switching, configuration B



(c) Hump mode, configuration A



(d) Hump mode, configuration B

Figure 5.1: Design changes from a configuration A to a configuration B may lead to discontinuities in a flutter constraint due to mode-switching (5.1a, 5.1b) or a hump mode (5.1c, 5.1d).

optimization problem.

This section compares two frequently used constraint aggregation functions, the p-norm [72] and Kreisselmeier-Steinhauser [35] functions, and discusses their properties relevant for application in a flutter constraint. Specifically, the applicability of these properties to the constraint formulation at hand are discussed.

5.2.1 p-Norm Aggregation

The p-norm function is a popular method of constraint aggregation. The aggregation function is given by [72]:

$$\|g\|_p = \left(\sum_{i=1}^{N} |g_i|^p \right)^{\frac{1}{p}} \tag{5.3}$$

It has been used as an aggregation function for stress constraints. However, as evident from Equation 5.3, the p-norm function is only applicable to positive constraint values. Constraining the damping value of the flutter eigenvalue problem would result in constraint violations. As the negative damping values indicate a stable solution, the absolute value imposed by the p-norm would result in all aggregated values being positive, and thus, infeasible. As such, the p-norm was not considered for the flutter constraint.

5.2.2 Kreisselmeier-Steinhauser Functions

KS functions [35] are a constraint aggregation methodology for m constraints $g_j(x)$ originally given by:

$$KS(g(x)) = \frac{1}{\bar{\rho}} \ln \left(\sum_{j=1}^{m} e^{\bar{\rho} g_j(x)} \right) \tag{5.4}$$

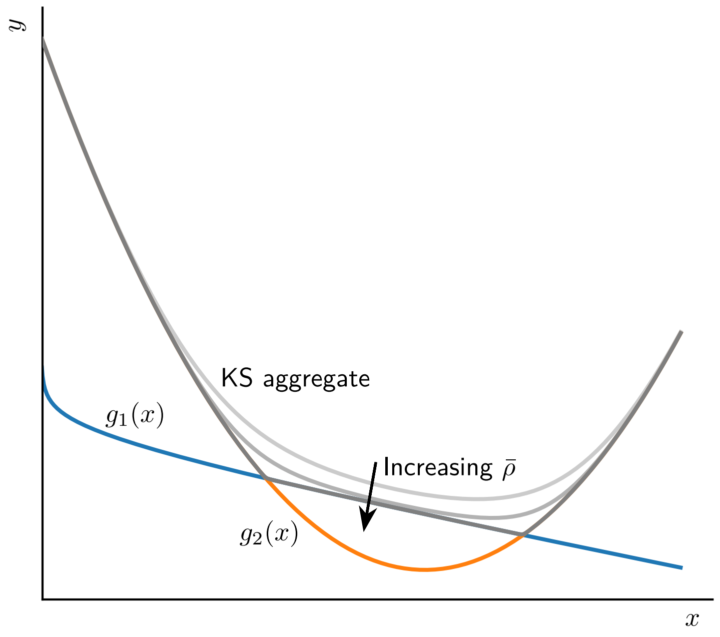


Figure 5.2: Example of KS aggregation over two constraints with varying parameter $\bar{\rho}$.

An alternate formulation was proposed [35], which avoids numerical instabilities due to overflow [71]:

$$KS\left(g\left(x\right)\right) = g_{\max}\left(x\right) + \frac{1}{\bar{\rho}}\ln\left(\sum_{j=1}^{m} e^{\bar{\rho}\left(g_j(x) - g_{\max}(x)\right)}\right) \tag{5.5}$$

The KS function derivatives are [71]:

$$\frac{\partial KS\left(g\left(x\right)\right)}{\partial x} = \frac{\sum\limits_{j=1}^{m} e^{\bar{\rho}\left(g_j(x) - g_{\max}(x)\right)}\frac{\partial g_j(x)}{\partial x}}{\sum\limits_{j=1}^{m} e^{\bar{\rho}\left(g_j(x) - g_{\max}(x)\right)}} \tag{5.6}$$

As Jonsson and coworkers summarized [34] et al.[73] noted that KS functions:

- are convex if (and only if) the constraint functions are convex;
- are larger than the maximum constraint for all $\bar{\rho} > 0$;
- converge to the maximum constraint for $\bar{\rho} \to \infty$;
- approach the maximum constraint monotonically in $\bar{\rho}$.

An example of a KS aggregation for multiple values of the factor $\bar{\rho}$ is shown in Figure 5.2. Clearly, a larger $\bar{\rho}$ results in a less conservative aggregate. However, the example also illustrates, that if the factor is not chosen to be sufficiently large, the aggregation may artificially limit the optimization as feasible parts of the design space become infeasible. Finally, it should be noted that too many constraints aggregated, as well as too large factor $\bar{\rho}$ will lead to inaccurate aggregations that may not yield any feasible designs [34]. Because the KS functions yields a smooth approximation of the critical constraint value, this type of constraint aggregation was chosen for this work.

5.3 Flutter Constraint Formulation

To reduce the number of constraints required to include flutter in the optimization problem, Jonsson and coworkers [71] proposed using KS functions to aggregate the flutter constraints for a linear flutter solution. This results in a continuous constraint and reduces the number of constraints in the optimization problem, i.e.,

$$KS\left(\mathrm{Re}\left(\zeta_i\right)\right) < 0 \tag{5.7}$$

Jonsson et al. [34], proposed a flutter constraint using a sequential KS functions. This work utilizes the concept of two KS aggregations in series and applies it to geometrically nonlinear flutter problems. The first KS aggregation is applied to every search iteration to obtain the most critical damping value for that iteration, yielding a set of critical damping values. The second KS aggregation yields the most critical value of the previous set. Thus, the nonlinear flutter constraint can be formulated in terms of the real part of the flutter solution eigenvalues and KS functions:

$$KS\left(KS\left(\mathrm{Re}\left(\zeta_i\right)\right)\right) < 0 \tag{5.8}$$

The sequential application of KS aggregations are, in fact, equivalent to a single KS aggregation over all constraint values (see Appendix B for the derivation). Nonetheless, using this two-tiered approach has organizational advantages. It permits an aggregation within the flutter solver to obtain the most critical damping value at a given search point, while the final aggregation is conducted within the MDO framework. Chapter 11 discusses how this distribution of tasks can lead to a reduction in the number of user-provided gradients.

The formulation in Equation 5.8 yields a scalar constraint over the entire flight envelope (Figure 5.3). As no connectivity information is required, the method permits a variety of approaches to sampling (a more detailed discussion of the flight envelope sampling process, along with pitfalls, is presented in Chapter 6). However, it should be noted that the quality of the sample set, and thereby the sampling process, play a crucial role in obtaining an accurate flutter constraint. Finally, to limit the number of constraints that need to be aggregated, and to avoid numerical instability in the KS aggregation, the number of modes used for the constraint is limited to a user defined quantity (substantially less than full set of eigenvalues obtained from the eigenvalue problem (EVP)).

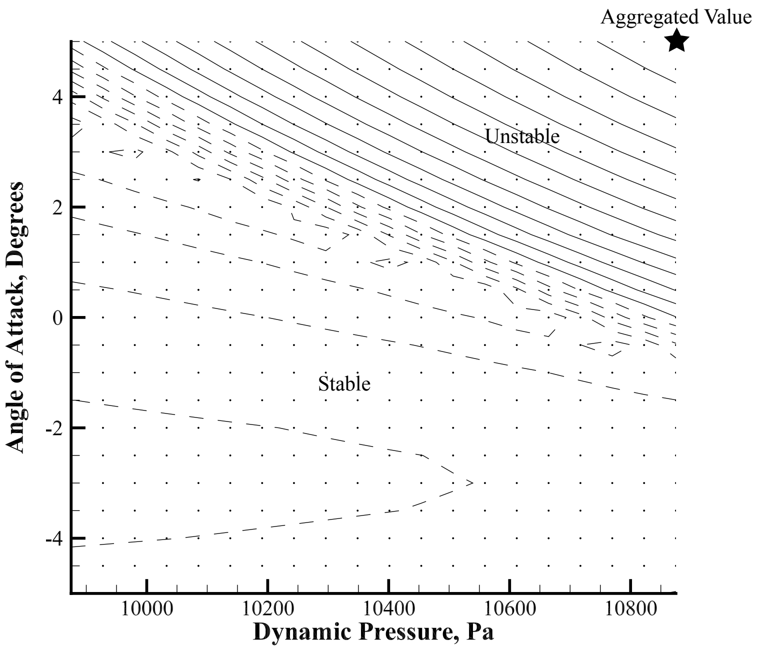


Figure 5.3: Qualitative example of the sequential application of KS aggregation to obtain a scalar, geometrically nonlinear flutter constraint.

CHAPTER 6

Flight Envelope Sampling

Many aircraft design problems require sampling the vehicle flight envelope for design exploration or constraint applications. For example, vehicle performance may differ significantly at different operating points, so the designer needs to be able to sample the operational space to determine the effects on a predetermined mission. Similarly, the flutter constraint formulated in Chapter 5 requires adequate flight envelope sampling to ensure feasibility across all operating conditions. As such, the sampling method becomes indispensable for design and optimization problems and may have a significant impact on the resulting configurations.

However, sampling a flight envelope is fraught with difficulties. Few flight envelopes or design spaces are hypercubes, and therefore, using conventional sampling methods may miss critical points. This chapter discusses typical sampling methods (Section 6.1), presents issues encountered when using these conventional methods (Section 6.2), and proposes sampling methods intended for aircraft design and optimization problems. Section 6.3 presents a constrained sampling algorithm adapted from Golchi and Loepky [74].

6.1 Hypercube Sampling

Traditional sampling methods used during Design of Experiments (DOE) function within an N-dimensional hypercube. Popular methods for sampling a hypercube are random (Figure 6.1a), Latin Hypercube Sampling (LHS) (Figure 6.1b), Halton (Figure 6.1c) and Hammersley sampling (Figure 6.1d).

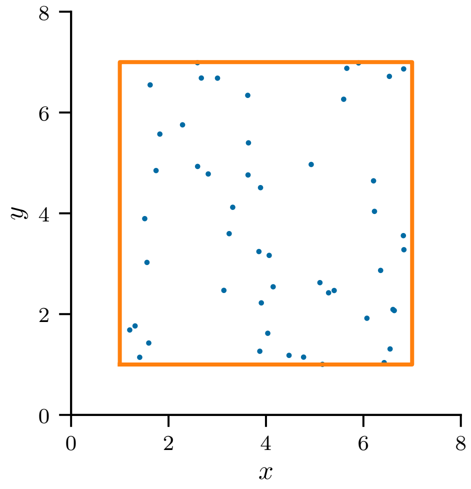
As seen in Figure 6.1, the different hypercube sampling methods result in vastly different sampling points. To be able to choose an appropriate method, a metric must be chosen that defines a “good” method. For this thesis, space filling DOE are sought. Therefore, the discrepancy of the resulting samples should be as low as possible. The discrepancy of the samples is a measure of how evenly spaced a distribution of samples is. Inspecting Figure 6.1, one can observe that the random sampling and LHS yield samples that are more clustered than the other methods. Halton and Hammersley samples, by contrast, both show good space filling designs. Because of this, this work uses Hammersley sampling any time hypercube samples are required.

6.2 Problems with Hypercube Sampling

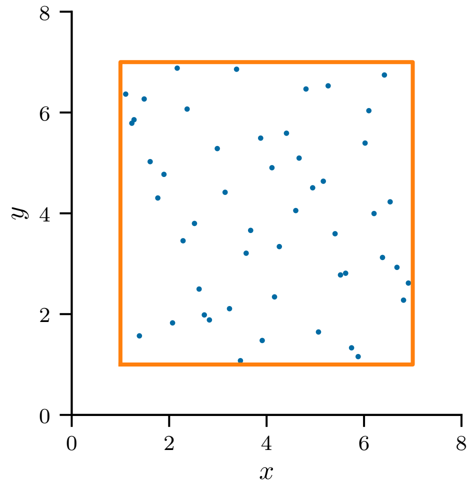
The sampling process is complicated by the fact that the flight envelope is not (generally) a hypercube. If the flight envelope is sampled using a hypercube (Figure 6.2), either the sample will only represent a subspace of the entire flight envelope or extend past its bounds. In the case of under-sampling (Figure 6.2a) critical parts of the flight envelope may not be sampled, potentially missing constraint violations within the envelope. On the other hand, points sampled outside the flight envelope are of no interest and may even fail to yield an aeroelastic solution, making the oversampling approach (Figure 6.2b) wasteful for an optimization constraint.

6.3 Constrained Sampling

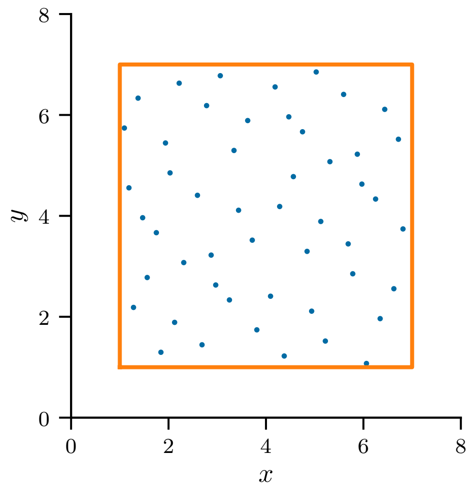
As such, an algorithm capable of sampling a non-hypercube, non-convex flight envelope is needed. This is achieved by a two-step process: an initial fine sampling of a hypercube surrounding the entire flight envelope (Figure 6.3a) followed by an ensuing



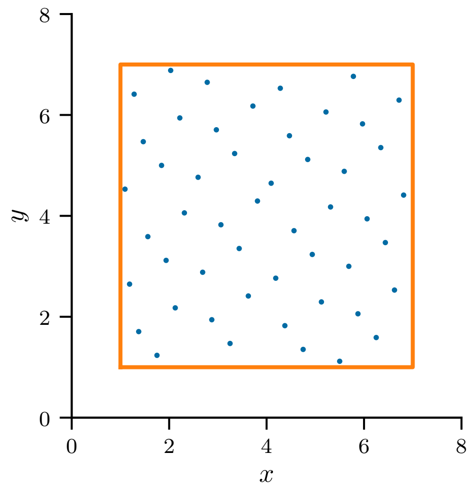
(a) Random sampling



(b) Latin Hypercube sampling

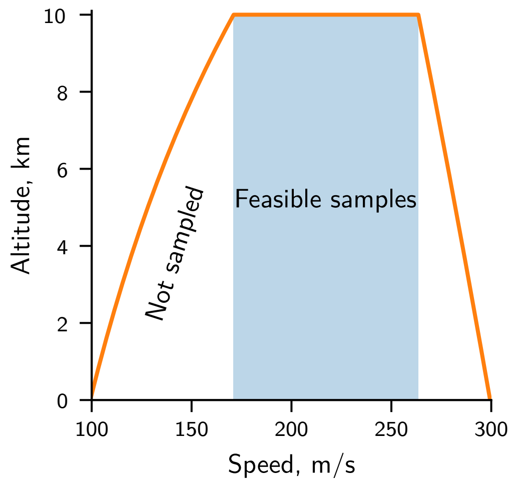


(c) Halton sampling

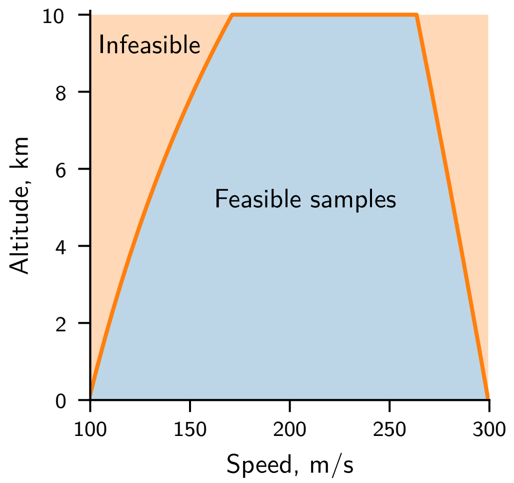


(d) Hammersley sampling

Figure 6.1: Comparison of different hypercube sampling methods.



(a) Underfitting



(b) Overfitting

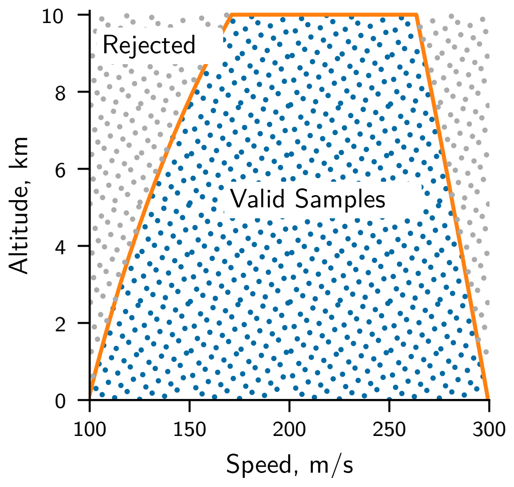
Figure 6.2: Underfitting (left) and overfitting (right) of the flight envelope using a hypercube. If the evelope is underfit critical damping values may not be considered for the flutter constraint, while overfitting may not yield a solution for points outside the flight envelope.

constrained Maximin (cMm) approach [74] (Figure 6.3b).

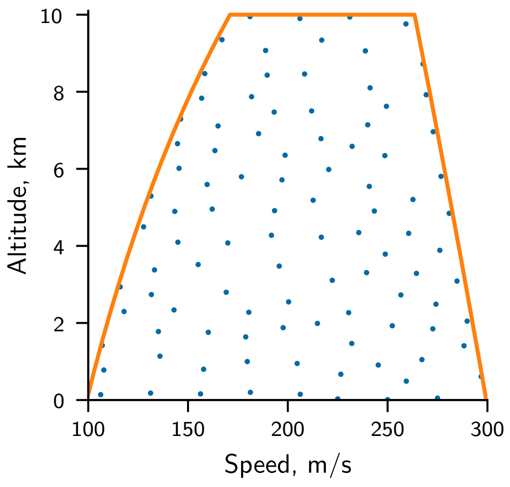
The initial sampling of the flight envelope is conducted using Hammersley pseudo random hypercube sampling [75] encompassing the entire flight envelope. As this initial sampling serves as the seed for the second sampling process, a very fine sampling is required. During this initial step a filter is applied such that only points within the constrained region are kept. The final cMm sampling step uses the samples obtained in this manner and applies a Maximin algorithm to obtain a low discrepancy constrained sample of a size specified by the user.

The cMm sampling algorithm maximizes the minimum distance between sample points using a defined distance metric. The cMm method in this work uses a weighted Euclidean distance metric between two points a and b:

$$\delta\left(a,b\right)=\sqrt{\sum_{i=1}^{N}w_i^{\delta}\left(a_i-b_i\right)^2} \tag{6.1}$$



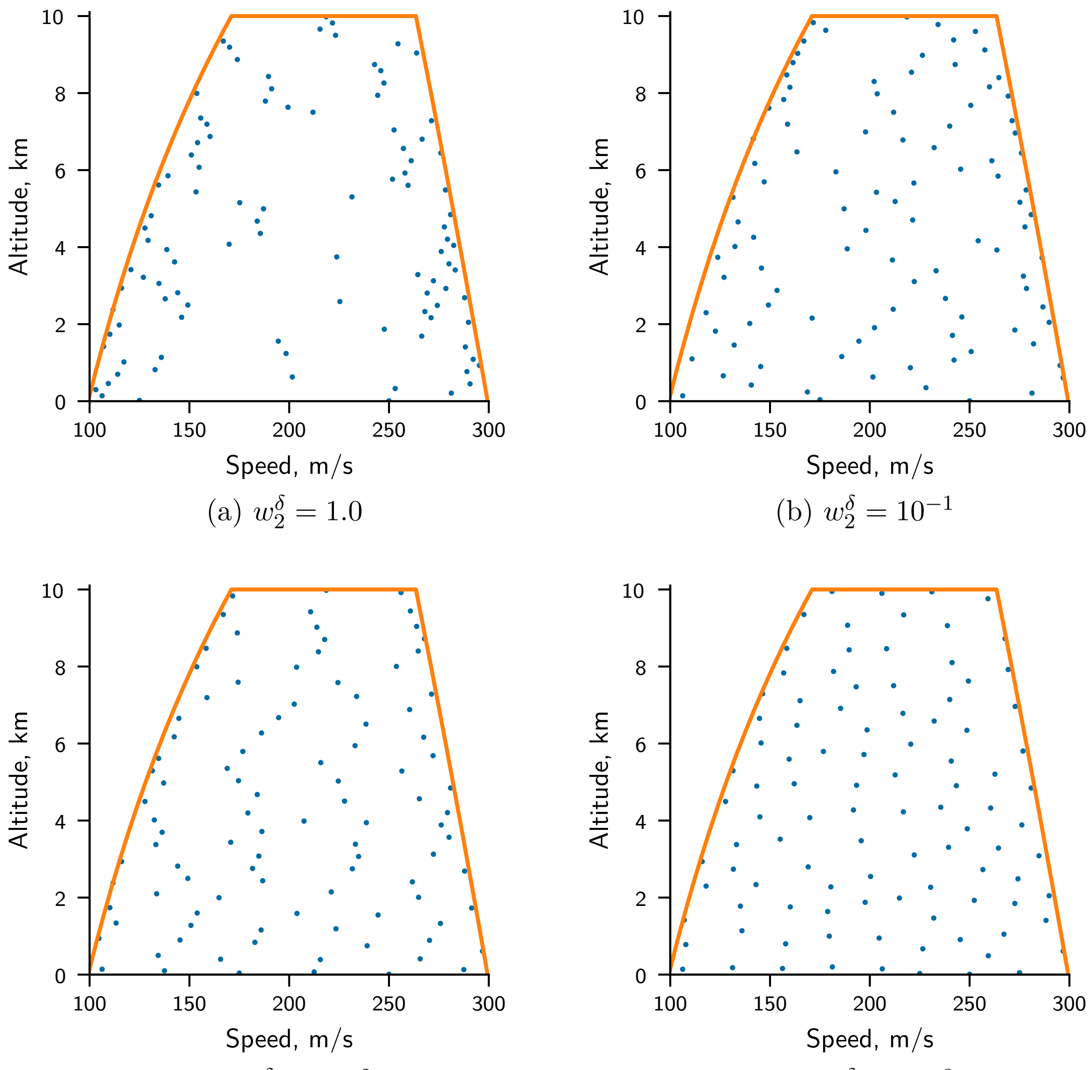
(a) Initial sampling



(b) Constrained Maximin sampling

Figure 6.3: Initial pseudo random sampling of the domain using hypercube and final sampling process using a cMm method. Sample points outside of the constrained area are discarded during the initial sampling. The final sampling process using a cMm approach yields a low discrepancy sample.

It should be noted that using a simple Euclidean distance ($w_i^\delta = 1$) will yield poor results when sampling the flight envelope (Figure 6.4a). Typically, the aircraft altitude range and the speed range will differ by an order of magnitude. As a result, an unweighted cMm sampling will yield a biased sampling, as in Figure 6.4a, where the boundaries are sampled more extensively than the rest of the flight envelope. If applied in a flutter constraint, this would mean that a large portion of the flight envelope in which the aircraft regularly operates will remain unsampled. As a result, critical flutter damping values may not be accounted for during the optimization, yielding an inaccurate flutter constraint and an infeasible design. While modifying the Euclidean weights (Figure 6.4b–6.4d) results in a better sampling of the envelope, the lowest discrepancy sample is achieved only for the correct weighting factor (Figure 6.4d). Generally, this weight is determined such that the weighted components of the sample vector are of the same order of magnitude. While flutter is more likely to occur towards the high-speed boundary of the flight envelope, this sampling approach accounts for hump modes in the flight envelope. Finally, the flutter constraint approach using this sampling methodology can be easily modified to account for other aeroelastic and rigid-body instabilities (such as phugoid modes, etc.).



(a) $w_2^\delta = 1.0$

(b) $w_2^\delta = 10^{-1}$

(c) $w_2^\delta = 10^{-2}$

(d) $w_2^\delta = 10^{-3}$

Figure 6.4: Comparison of the constrained Maximin sampling using different Euclidean distance weights.

CHAPTER 7

Determining Equivalent Beam Properties

An equivalent beam condensation is required to integrate the nonlinear beam-based simulations with higher-fidelity analyses. Additionally, gradients of the beam properties are determined with respect to higher-fidelity design variables. This chapter describes the methods used to determine both the equivalent beam properties and their gradients. These properties are verified in Appendix **??** and applied to a multi-fidelity optimization problem in Chapter 16.

The beam condensation consists of two separate processes: a mass condensation and a stiffness condensation. The mass condensation simplifies every element as a point mass and determines the equivalent beam mass properties from these. The equivalent beam stiffness properties are determined from FEM runs for linearly independent load cases. Gradients for the mass condensation are determined analytically, while the stiffness sensitivities are determined using AD.

7.1 Mass Condensation

7.1.1 Function Values

The mass condensation uses the high-fidelity FEM model and reduces it to point masses—one for every element (Figure 7.1). For an entire model, the high fidelity model must be subdivided such that every beam node is associated with its neigh-

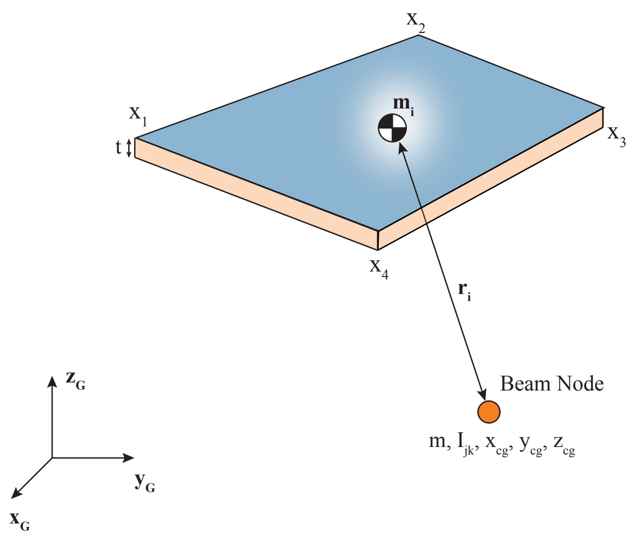


Figure 7.1: Relationship between a mass element and the beam reference node.

boring high fidelity elements. This can be achieved, for example, using a nearest neighbor approach. The mass of every element is determined from the element area and density, assuming a constant element thickness:

$$m_j = \rho A t \tag{7.1}$$

The element area is determined from the element corner points using Heron's formula [76] (using two triangles for quadrilateral elements).The equivalent beam mass is then obtained from the sum of the element masses associated with the beam section.

$$m_e = \sum_{j=1}^{N_{elem}} m_j \tag{7.2}$$

The center of gravity of the beam section (determined from the individual element masses) is:

$$x_{cg} = \frac{\sum_{j=1}^{N_{elem}} m_j r_{xj}}{\sum_{j=1}^{N} m_j} \tag{7.3}$$

$$y_{cg} = \frac{\sum_{j=1}^{N_{elem}} m_j r_{yj}}{\sum_{j=1}^{N} m_j} \tag{7.4}$$

$$z_{cg} = \frac{\sum_{j=1}^{N_{elem}} m_j r_{zj}}{\sum_{j=1}^{N} m_j} \tag{7.5}$$

The location of the quadrilateral centroids is determined by obtaining the weighted average of the vertex locations, (x_i^1, y_i^1, z_i^1), (x_i^2, y_i^2, z_i^2), (x_i^3, y_i^3, z_i^3), and (x_i^4, y_i^4, z_i^4):

$$r_{xi} = \frac{x_i^1 + x_i^2 + x_i^3 + x_i^4}{4} \tag{7.6}$$

$$r_{yi} = \frac{y_i^1 + y_i^2 + y_i^3 + y_i^4}{4} \tag{7.7}$$

$$r_{zi} = \frac{z_i^1 + z_i^2 + z_i^3 + z_i^4}{4} \tag{7.8}$$

Finally, the inertia of the equivalent beam section (determined from the individual element masses) is:

$$I_{xx} = \sum_{j=1}^{N_{elem}} m_j \left(r_{yj}^2 + r_{zj}^2\right) \tag{7.9}$$

$$I_{xy} = \sum_{j=1}^{N_{elem}} m_j r_{xj} r_{yj} \tag{7.10}$$

$$I_{xz} = \sum_{j=1}^{N_{elem}} m_j r_{xj} r_{zj} \tag{7.11}$$

$$I_{yy} = \sum_{j=1}^{N_{elem}} m_j \left(r_{xj}^2 + r_{zj}^2 \right) \tag{7.12}$$

$$I_{yz} = \sum_{j=1}^{N_{elem}} m_j r_{yj} r_{zj} \tag{7.13}$$

$$I_{zz} = \sum_{j=1}^{N_{elem}} m_j \left(r_{xj}^2 + r_{yj}^2 \right) \tag{7.14}$$

7.1.2 Gradients

Because the beam condensation is applied to a gradient-based optimization problem, efficiently and accurately determining the gradients is paramount. As mentioned previously, the determination of the gradients is subdivided on the component level of the optimization problem. As such, the gradients of the mass properties with respect to the component design variables (element thicknesses and densities) are required. The formulae for the mass condensation process are comparatively simple, so the gradients were obtained analytically. The derivatives of the beam section mass with respect to element thickness and density are:

$$\frac{\partial m}{\partial t_i} = \sum_{j=1}^{N} \frac{\partial m_j}{\partial t_i} = \frac{\partial m_i}{\partial t_i} \tag{7.15}$$

$$\frac{\partial m}{\partial \rho_i} = \sum_{j=1}^{N} \frac{\partial m_j}{\partial \rho_i} = \frac{\partial m_i}{\partial \rho_i} \tag{7.16}$$

for $i \in [1, N]$, otherwise $\frac{\partial m}{\partial t_i} = 0$ and $\frac{\partial m}{\partial \rho_i} = 0$.

Note, that the gradients simplify to the derivative of the mass element with respect to its inputs, with all other entries of the gradient vector equaling zero. This simplifies the derivatives of the other mass properties and also results in a sparse Jacobian, which improves computational efficiency.

The derivatives of the center of gravity are determined using the quotient rule:

$$\frac{\partial x_{cg}}{\partial t_i} = \frac{\frac{\partial m_i}{\partial t_i} r_{xi} \sum_{j=1}^{N} m_j - \frac{\partial m_i}{\partial t_i} \sum_{j=1}^{N} m_j r_{xj}}{\left(\sum_{j=1}^{N} m_j \right)^2} \tag{7.17}$$

For the element mass the equation for the center of gravity is rearranged to obtain:

$$\sum_{j=1}^{N} m_j r_{xj} = x_{cg} \sum_{j=1}^{N} m_j \tag{7.18}$$

Using Equation 7.18, the center of gravity gradients with respect to element thickness and density are:

$$\frac{\partial x_{cg}}{\partial t_i} = \frac{\frac{\partial m_i}{\partial t_i} \left(r_{xi} - x_{cg} \right)}{m_e} \tag{7.19}$$

$$\frac{\partial y_{cg}}{\partial t_i} = \frac{\frac{\partial m_i}{\partial t_i} \left(r_{yi} - y_{cg} \right)}{m_e} \tag{7.20}$$

$$\frac{\partial z_{cg}}{\partial t_i} = \frac{\frac{\partial m_i}{\partial t_i} \left(r_{zi} - z_{cg} \right)}{m_e} \tag{7.21}$$

$$\frac{\partial x_{cg}}{\partial \rho_i} = \frac{\frac{\partial m_i}{\partial \rho_i} \left(r_{xi} - x_{cg} \right)}{m_e} \tag{7.22}$$

$$\frac{\partial y_{cg}}{\partial \rho_i} = \frac{\frac{\partial m_i}{\partial \rho_i} \left(r_{yi} - y_{cg} \right)}{m_e} \tag{7.23}$$

$$\frac{\partial z_{cg}}{\partial \rho_i} = \frac{\frac{\partial m_i}{\partial \rho_i} \left(r_{zi} - z_{cg} \right)}{m_e} \tag{7.24}$$

Finally, the derivatives of the inertia of the beam section with respect to the element thickness and density are:

$$\frac{\partial I_{xx}}{\partial t_i} = \frac{\partial m_i}{\partial t_i} \left(r_{yj}^2 + r_{zj}^2 \right) \tag{7.25}$$

$$\frac{\partial I_{xy}}{\partial t_i} = \frac{\partial m_i}{\partial t_i} r_{xj} r_{yj} \tag{7.26}$$

$$\frac{\partial I_{xz}}{\partial t_i} = \frac{\partial m_i}{\partial t_i} r_{xj} r_{zj} \tag{7.27}$$

$$\frac{\partial I_{yy}}{\partial t_i} = \frac{\partial m_i}{\partial t_i} \left(r_{xj}^2 + r_{zj}^2\right) \tag{7.28}$$

$$\frac{\partial I_{yz}}{\partial t_i} = \frac{\partial m_i}{\partial t_i} r_{yj} r_{zj} \tag{7.29}$$

$$\frac{\partial I_{zz}}{\partial t_i} = \frac{\partial m_i}{\partial t_i} \left(r_{xj}^2 + r_{yj}^2\right) \tag{7.30}$$

$$\frac{\partial I_{xx}}{\partial \rho_i} = \frac{\partial m_i}{\partial \rho_i} \left(r_{yj}^2 + r_{zj}^2\right) \tag{7.31}$$

$$\frac{\partial I_{xy}}{\partial \rho_i} = \frac{\partial m_i}{\partial \rho_i} r_{xj} r_{zj} \tag{7.32}$$

$$\frac{\partial I_{xz}}{\partial \rho_i} = \frac{\partial m_i}{\partial \rho_i} r_{xj} r_{zj} \tag{7.33}$$

$$\frac{\partial I_{yy}}{\partial \rho_i} = \frac{\partial m_i}{\partial \rho_i} \left(r_{xj}^2 + r_{zj}^2\right) \tag{7.34}$$

$$\frac{\partial I_{yz}}{\partial \rho_i} = \frac{\partial m_i}{\partial \rho_i} r_{yj} + r_{zj} \tag{7.35}$$

$$\frac{\partial I_{zz}}{\partial \rho_i} = \frac{\partial m_i}{\partial \rho_i} \left(r_{xj}^2 + r_{yj}^2\right) \tag{7.36}$$

7.2 Stiffness Condensation

7.2.1 Function Values

The equivalent beam stiffness condensation used in this work was first proposed by Malcolm and Laird [45] to accurately deduce beam properties of wind turbine blades for subsequent aeroelastic analyses. The process has since been applied to aircraft structures[77, 44]. Furthermore, Stodieck et al. [44] extended Malcolm's process to obtain gradients of the stiffness properties for equivalent beam condensations in optimization problems.

The stiffness condensation component within this work consists of two distinct processes (Figure 7.2): high-fidelity FEM runs to obtain equivalent beam displacements and rotiations and the determination of the stiffness properties (from the equivalent beam displacements and rotations previously determined).

The high-fidelity FEM simulations are conducted for six linearly independent load cases to obtains six sets of beam displacements. In this work, these load cases are evaluated and the equivalent beam displacements recovered using Rigid Body Elements (RBEs). A set of six linearly independent load cases are:

$$\left(F^t\right)_1 = \begin{bmatrix} F_x & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T \tag{7.37}$$

$$\left(F^t\right)_2 = \begin{bmatrix} 0 & F_y & 0 & 0 & 0 & 0 \end{bmatrix}^T \tag{7.38}$$

$$\left(F^t\right)_3 = \begin{bmatrix} 0 & 0 & F_z & 0 & 0 & 0 \end{bmatrix}^T \tag{7.39}$$

$$\left(F^t\right)_4 = \begin{bmatrix} 0 & 0 & 0 & M_x & 0 & 0 \end{bmatrix}^T \tag{7.40}$$

$$\left(F^t\right)_5 = \begin{bmatrix} 0 & 0 & 0 & 0 & M_y & 0 \end{bmatrix}^T \tag{7.41}$$

$$\left(F^t\right)_6 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & M_z \end{bmatrix}^T \tag{7.42}$$

The element stiffness matrix is evaluated from the internal forces f^i, which results

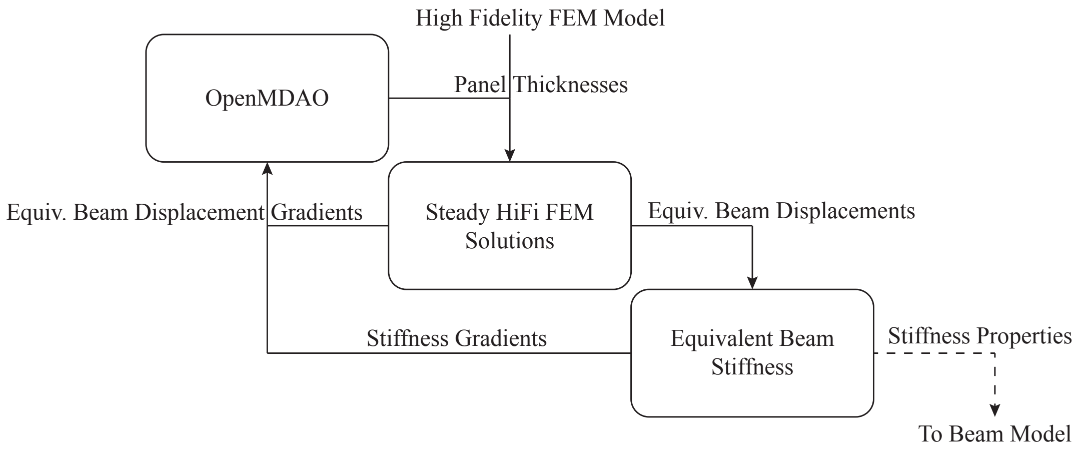

High Fidelity FEM Model
OpenMDAO
Panel Thicknesses
Equiv. Beam Displacement Gradients
Steady HiFi FEM Solutions
Equiv. Beam Displacements
Stiffness Gradients
Equivalent Beam Stiffness
Stiffness Properties
To Beam Model

Figure 7.2: Block diagram of the complete stiffness condensation process including high-fidelity FEM solutions and the ensuing determination of equivalent beam stiffnesses.

from the applied tip loads (Figure 7.3), and the element strains:

$$f^i = \begin{bmatrix} f_x^i \\ f_y^i \\ f_z^i \\ m_x^i \\ m_y^i \\ m_z^i \end{bmatrix} = k\varepsilon \tag{7.43}$$

The element strains are defined in Equation 7.44 and can be rewritten in terms

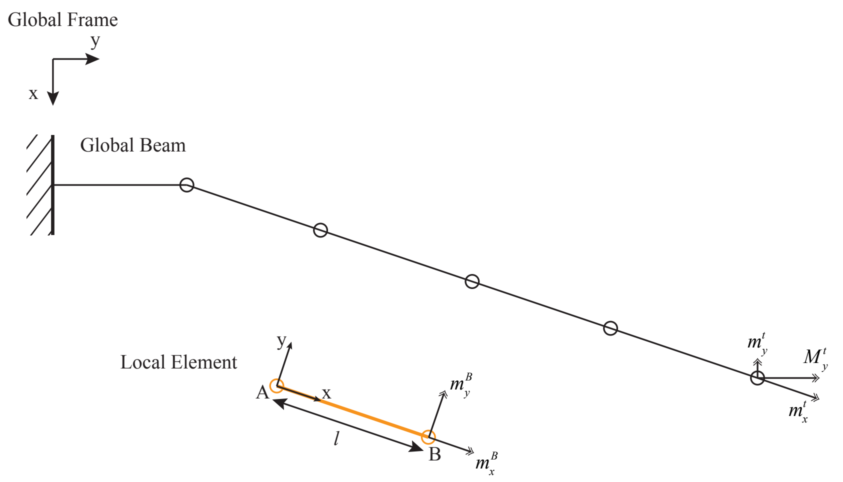


Figure 7.3: Diagram of the coordinate frames and global and local load conventions (shown here for an applied tip moment) of the stiffness condensation process.

of displacements:

$$\varepsilon = \begin{bmatrix} \varepsilon_x \\ \gamma_y \\ \gamma_z \\ \kappa_x \\ \kappa_y \\ \kappa_z \end{bmatrix} \tag{7.44}$$

(7.45)

Assuming a geometrically linear relation, this can be expanded to:

$$\varepsilon = \begin{bmatrix} \frac{\partial u_x}{\partial x} \\ \frac{\partial u_y}{\partial x} \\ \frac{\partial u_z}{\partial x} \\ \frac{\partial \theta_x}{\partial x} \\ \frac{\partial \theta_y}{\partial x} \\ \frac{\partial \theta_z}{\partial x} \end{bmatrix} + \begin{bmatrix} 0 \\ -\theta_z \\ \theta_y \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{7.46}$$

$$= \begin{bmatrix} \frac{\partial u_x}{\partial x} \\ \frac{\partial u_y}{\partial x} \\ \frac{\partial u_z}{\partial x} \\ \frac{\partial \theta_x}{\partial x} \\ \frac{\partial \theta_y}{\partial x} \\ \frac{\partial \theta_z}{\partial x} \end{bmatrix} + \int_0^x \begin{bmatrix} 0 \\ -\kappa_z \\ \kappa_y \\ 0 \\ 0 \\ 0 \end{bmatrix} \mathrm{dx} \tag{7.47}$$

The element local displacement and local internal force vectors u and f are obtained by transforming the global displacements U and F in to the local coordinate system:

$$u = \begin{bmatrix} u_x \\ u_y \\ u_z \\ \theta_x \\ \theta_y \\ \theta_z \end{bmatrix} = TU \tag{7.48}$$

$$f^t = TF^t \tag{7.49}$$

The element stiffness properties are determined by initially solving for the stiffness matrix in the local frame. The local element stiffness matrix is obtained from:

$$f^i = K\Delta u \tag{7.50}$$

with

$$\Delta u = \begin{bmatrix} u_x^B - u_x^A \\ u_y^B - u_y^A - l\theta_z^A \\ u_z^B - u_z^A + l\theta_y^A \\ \theta_x^B - \theta_x^A \\ \theta_y^B - \theta_y^A \\ \theta_z^B - \theta_z^A \end{bmatrix} = k\varepsilon \tag{7.51}$$

Malcolm[45] derived the relationship between the local stiffness matrix K and the stiffness matrix k in Lyapunov form, which can be solved for k^{-1} using Lyapunov's method:

$$K^{-1}Q^{-1} = k^{-1}HQ^{-1} + Ek^{-1} \tag{7.52}$$

with

$$E = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{7.53}$$

$$H = \begin{bmatrix} l & 0 & 0 & 0 & 0 & 0 \\ 0 & l & 0 & 0 & 0 & 0 \\ 0 & 0 & l & 0 & 0 & 0 \\ 0 & 0 & 0 & l & 0 & 0 \\ 0 & 0 & -\frac{l^2}{2} & 0 & l & 0 \\ 0 & \frac{l^2}{2} & 0 & 0 & 0 & l \end{bmatrix} \tag{7.54}$$

$$Q = \begin{bmatrix} \frac{l^2}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{l^2}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{l^2}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{l^2}{2} & 0 & 0 \\ 0 & 0 & -\frac{l^3}{3} & 0 & \frac{l^2}{2} & 0 \\ 0 & \frac{l^3}{3} & 0 & 0 & 0 & \frac{l^2}{2} \end{bmatrix} \tag{7.55}$$

7.2.2 Gradients

As with the equivalent beam mass condensation, the gradients of the stiffness condensation with respect to its input variables are required for the optimization problem (Figure 7.2). While the mass condensation gradients were determined analytically, the gradients of the stiffness condensation component were determined using AD, while the gradients of the FEM solutions needed for the condensation are obtained from the FEM solution itself. Based on previous studies of AD libraries (Chapter 8), the C++ library CoDiPack [78] was chosen. The application of AD to this problem mirrors the implementation for UM/NAST detailed in Chapter 10.

The stiffness condensation in this work was implemented using templates, permitting the function evaluation using standard C++ floating precision types without the overhead of operator overloading AD. The evaluation of the gradient with respect to the equivalent beam displacements is then evaluated using the CoDiPack types similar to the work conducted in UM/NAST (see Chapter 10). The templated function is evaluated using AD only when the gradients are desired. Otherwise, the function evaluation of the equivalent beam stiffnesses is determined using the standard *double* implementation. It should be noted, that the stiffness matrix of a given element only depends on the displacements of that element's corner nodes and is independent of any other nodes' displacements. As a result, the Jacobian of the stiffness properties is very sparse, yielding a computationally efficient solution.

Part II

Tools

CHAPTER 8

Algorithmic Differentiation

The conceptual background for determining gradients numerically were presented in Chapter 2. As a summary, AD is a method of determining derivatives by applying the chain rule to source code. AD methods include source code transformation and operator overloading. Because of the lack of available source transformation tools for the C++ language, operator overloading is the only practical solution. This chapter discusses AD tools available for this work and explains the tool selection criteria and process. Two AD libraries, Adept [79] and CoDiPack [78], were evaluated with respect to feature set and performance. Finally, the practical ramifications of applying AD to existing code, and how I addressed these for the software in this work, are detailed.

8.1 Adept

Adept is operator-overloading C++ library that implements forward and reverse mode AD and is freely available under the Apache 2 license. It was developed at the University of Reading by Robin Hogan [79]. While two major versions of the library exist (versions 1.1 and 2.0), they are differentiated solely by the addition of array manipulation in version 2.0. Specifically, versions 1.1 and 2.0 are compatible with each other regarding their AD operations. For this work, Adept version 2.0 was used, as it is the newest version, updated in 2018.

The authors describe Adept as an efficient library for determining gradients and claim significant speed-ups compared to other C++ AD tools (cppAD and ADOLC), while being only approximately 10% slower than a hand-coded adjoint implementation [79]. To apply the library to an existing code, the floating point variables of the algorithm must be swapped with the *adouble* type. The library also contains a number of options that are intended to simplify the creation of non-AD objects, to avoid the performance penalties associated with AD.

8.2 CoDiPack

The Code Differentiation Package (CoDiPack) is an operator-overloading AD library developed at the TU Kaiserslautern and licensed under the GPL3 license. As it is newer than cppAD, ADOLC, and Adept, it was not compared against Adept by Hogan [79]. It is, however, used for the discrete adjoint approach implemented in SU2 [80]. CoDiPack, like Adept, uses expression templates for the fast evaluation of derivatives in forward or reverse mode. It was developed in parallel to the Message Differentiation Package (Medipack) [81], a library that enables AD applications with MPI. As such, CoDiPack was designed specifically with high performance computing (HPC) applications (such as SU2) in mind. Additionally, CoDiPack, due to its use in SU2, enjoys support within the aerospace community, while other tools (in C++) remain relatively unknown. Finally, Sagebaum et al. [78] detailed the methods they applied to efficiently determine gradients within SU2, providing a blueprint for future applications.

8.3 Selecting the Algorithmic Differentiation Tool

While choosing an AD tool may be driven primarily by performance, there are a plethora of criteria that need to be considered. Compatibility with the target software is key and often not a given, thereby requiring code modifications. Additionally, developers must anticipate the features they will need from the tool, as functionality varies greatly between AD libraries. This section provides a performance benchmark that investigates the scalability of the Adept and CoDiPack libraries. This together with package functionality provides the basis on which I chose the AD library for this work.

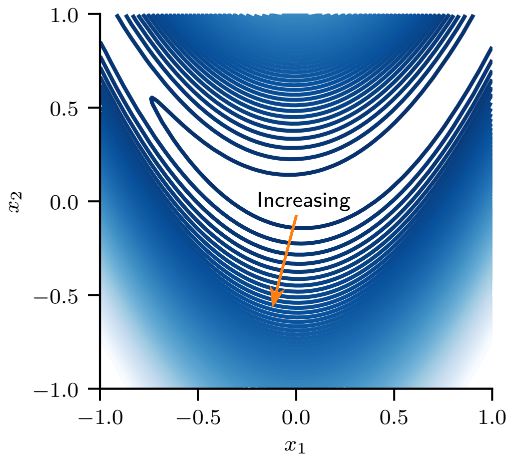
8.3.1 Performance Benchmarks

Benchmarking software performance can be difficult, as different packages may have varying strengths and weaknesses. For the choice of AD library in this work, however, it is key that the tool performs its evaluation quickly and scales well with the number of design variables. To this end, the Rosenbrock function [82] (Figure 8.1) was chosen to benchmark Adept and CoDiPack and compare them to the analytical gradient evaluation. The Rosenbrock function is a typical benchmark function to test the functionality and robustness of optimizers due to its steep walls and shallow valley. Moreover, it was chosen for this benchmark due to its multi-dimensionality. This permits an evaluation of the scalability of both the Adept and CoDiPack libraries with respect to the problem dimension. The n-dimensional function is defined as:

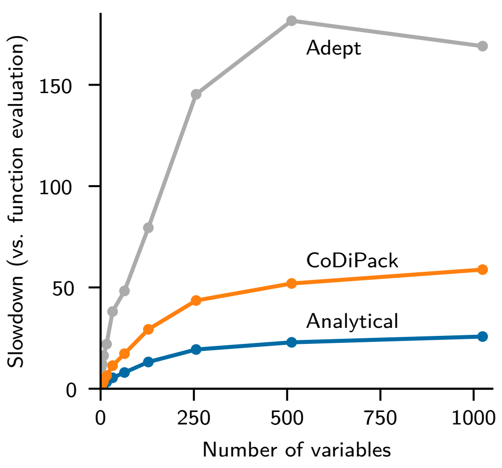
$$f(x) = \sum_{i=1}^{n-1} \left[100\left(x_{i+1} - x_i^2\right)^2 + \left(1 - x_i\right)^2 \right] \tag{8.1}$$

The dimensionality of the function was varied in powers of two, while the gradient evaluation time was averaged from 1000 sequential runs. The benchmarks were run on computer configuration A (see Appendix C). Figure 8.1b shows the slowdown of the respective AD libraries as well as the analytical gradient compared to a Rosenbrock function evaluation.

The analytical gradient shows a slowdown compared to the function evaluation



(a) Rosenbrock function



(b) AD tool comparison

Figure 8.1: Contour plot of the two-dimensional ($n = 2$) Rosenbrock function (left). Comparison of the slowdown of the gradient for the n-dimensional Rosenbrock function determined analytically, using Adept, and using CoDiPack in reverse mode compared to the time of a function evaluation (right).

as the function is scalar, while the gradient is a vector and therefore requires more operations to compute, even in the analytical case. CoDiPack performs well compared to the analytical gradient with a slowdown of approximately 2-3 relative to the analytical gradient and a factor of approximately 50 relative to the function evaluation. Furthermore, CoDiPack outperforms Adept which shows a slowdown factor of approximately 150 compared to the function evaluation.

8.3.2 Tool Selection

The performance benchmarks in Section 8.3.1 show that CoDiPack evaluates the gradients more efficiently and scales better with the number of design variables than Adept. Additionally, CoDiPack features a larger set of functionality, including the evaluation of n-th-order derivatives, which Adept does not. This is particularly important, as when applying the AD linearization of the equations of motion in UM/NAST (see Chapter 3 for the theoretical formulation and Chapter 10 for an implementation description) to an optimization problem, the second-order derivative of the state space matrix A is required:

$$\frac{\partial A}{\partial(\cdot)} = \frac{\partial \dot{y}}{\partial y \partial(\cdot)} \tag{8.2}$$

As such, CoDiPack was chosen for this work when numerical differentiation (instead of analytical derivatives) are required.

8.4 Implementational Aspects

Many pieces of software in research work solely use simple floating point types, usually in double precision. This is possible if the software must only determine the function value or the gradient is determined using either finite differences or an analytical method. The application of complex step and AD in this work, however, required a different approach. Without special considerations, each type would require an independent class definition, doubling or tripling the number of structures contained within the framework. Such redefinitions would result in substantial difficulties during code maintenance and would undermine the reuse of code. Instead all classes and methods can be templated using ISO C++17 to allow implementation flexibility.

Templated classes and functions enable the programmer to create one definition which can be reused for a large number of types. Consider the following example:

```cpp
template <typename Type>
Type Add(Type input1, Type input2)
{
    Type output = input1 + input2;

    return output;
};
```

The function *Add* takes two input arguments returns their sum. The function shown here was templated to allow additional flexibility when using the function.

This is illustrated applying the same function to two different datatypes.

```cpp
// evaluate the function using integers
int a1 = 1;
int b1 = 2;
int c1 = Add(a1, b1);

// evaluate the function using double precision floating point numbers
double a2 = 1.5;
double b2 = 2.5;
double c2 = Add(a2, b2);
```

Therefore, templating permits the programmer to write code once and apply it flexibly according to the types they require. Applying the same principle to UM/-NAST resulted in one set of class and function definitions, which could then be applied to several different numerical types. In this way, a complex step type and AD types were added to the UM/NAST framework for gradient recovery in addition to the existing function evaluation using the *double* type.

CHAPTER 9

MDO Framework

The studies presented in Part III encompass a multitude of disciplines and levels of fidelity. Because coupling these disciplines manually with an optimizer would be tedious and error prone, a MDO framework was required. The purpose of the MDO framework is to couple disciplines, preferably at a high level, thus, alleviating work for the researcher. This chapter presents the MDO framework and the high-fidelity tools used in this thesis. While a plethora of frameworks exist, OpenMDAO was chosen because of the ease of problem implementation. The individual disciplines (nonlinear aeroelastic, finite element, and aerodynamic analyses, etc.) have been integrated using OpenMDAO.

9.1 OpenMDAO

OpenMDAO [83] is an open source software framework designed to enable Multidisciplinary Design Analysis and Optimization (MDAO), using gradient-based optimization. The framework was written in Python and allows the integration of external software, e.g., C++ programs using Cython. Within this work, I employ OpenMDAO as the coupling agent between the individual disciplines, taking advantage of the behind-the-scenes assembly of the global sensitivities.

An OpenMDAO problem consists of *Components* that are assembled together to define the global problem. This modular approach simplifies evaluating the problem to defining the gradient at the Component level, providing derivatives for the Component outputs with respect to its inputs. OpenMDAO uses the partials defined in this manner to assemble the global Jacobians while utilizing problem sparsity for computational efficiency.

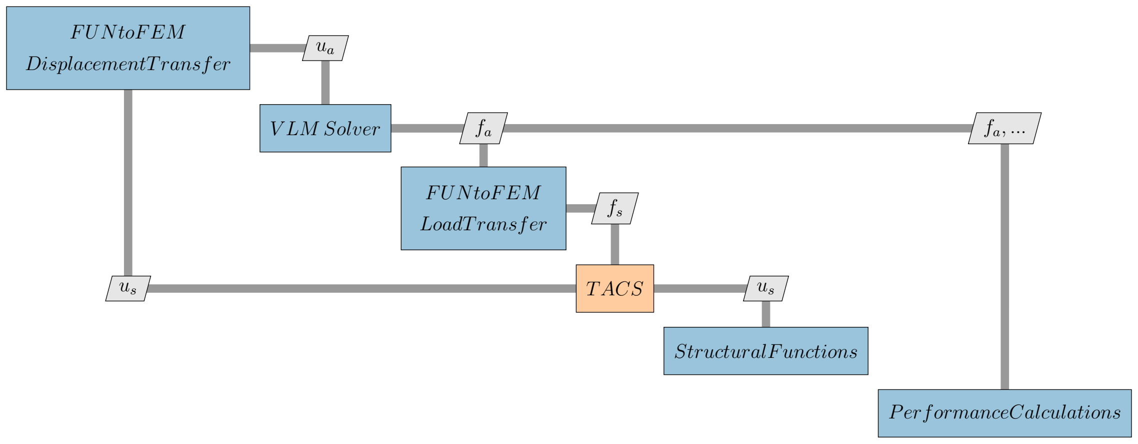
9.2 mphys

mphys is a python package developed at NASA and the University of Michigan's MDO Lab [84] that offers a toolset for creating a high-fidelity FSI optimization problem. For this work I used an older version of mphys, before it was reorganized. In the version utilized for this work, mphys offers the user so-called assemblers, which are responsible for creating and connecting structural and aerodynamic components in an OpenMDAO problem to form an FSI problem. Components provided within the mphys framework are include the TACS [85] structural solver as well as ADflow [86] load and displacement transfer components defined using FUNtoFEM [87].

The high-fidelity objective function of the multi-fidelity problem is defined and solved using the TACS FEM code [85]. TACS uses MPI for parallel solution evaluations and has the ability to provide an adjoint-based gradient for gradient-based optimization. It has been used for shell-based optimization problems and topology optimization using solid elements. As such, TACS is well suited for the proposed multi-fidelity problem.

For the "high" fidelity portion of the multi-fidelity problem presented in this work, I coupled the VLM solver from OpenAeroStruct [88] with the TACS shell-based FEM solver. The force and displacement transfer is achieved using FUNtoFEM, while the general problem setup was facilitated using the mphys toolbox.

The FSI solution cycle (Figure 9.1) begins with the evaluation of the VLM solver. FUNtoFEM then transfers the aerodynamic loads to the structural mesh used for the



$FUNtoFEM$ $DisplacementTransfer$
u_a
$VLM\ Solver$
f_a
$f_a, \ldots$
$FUNtoFEM$ $LoadTransfer$
f_s
u_s
$TACS$
u_s
$StructuralFunctions$
$PerformanceCalculations$

Figure 9.1: XDSM diagram of the coupled FSI solution between OpenAeroStruct's VLM solver and TACS.

TACS analysis. The resulting displacements are transfered back to the aerodynamic solver using FUNtoFEM. This iterative process is solved using a Nonlinear Block Gauss-Seidel solver.

9.3 FEMtoBeam

FEMtoBeam is a Python library that implements the mass and stiffness condensation process presented in Chapter 7. The mass condensation was written in pure Python and the gradients are programmed and obtained analytically. The stiffness condensation required a different approach, as it consists of a two-step process: obtaining the equivalent beam displacements from a higher-fidelity FEM solver (e.g., TACS or Nastran) and the process of determining the equivalent beam stiffnesses from the equivalent beam displacements. FEMtoBeam is responsible for the second part of the process, while an external solver must provide the equivalent beam properties. The derivatives of the equivalent beam stiffness properties are not provided analytically, but rather determined using AD via CoDiPack. Therefore, the stiffness condensation code was templated in a similar manner to UM/NAST, permitting both the evaluation of gradient using AD as well as their verification using complex step (Chapter **??**).

CHAPTER 10

UM/NAST Version 4.2

The University of Michigan's Nonlinear Aeroelastic Simulation Toolbox (UM/NAST) is a beam-based geometrically nonlinear aeroelastic framework that has been developed over several generations of graduate students at the Active Aeroelasticity and Structures Laboratory (A2SRL) at the University of Michigan. This thesis both utilizes and presents enhancements to the UM/NAST framework.

10.1 Historical Background and Motivation

The foundation of the UM/NAST framework was developed by Brown and Cesnik [56]. Brown wrote the initial Matlab strain-based beam code used in UM/NAST. The Matlab-based code was further developed by Shearer [57], Su [58], Dillsaver [59] among other Ph.D. students to contribute. In 2014, Pang [61] out to rewrite the Matlab-based UM/NAST into C++. During this time the file input/output mechanisms were changed, as were the fundamental code structure. Though written in C++, the code retained a function-based paradigm, as had been the case with the Matlab-based code. The initial version of the C++ became version 2.0 of the framework and improvements to various solvers and functionality were added in version 2.1 through 2.3. Despite additions, the fundamental structure of the code remained largely the same.

While an increase in the major version number implies fundamental structural changes, version 3.0 added few features, but rather focused on the modification of the source code to adhere to the Google C++ Style Guide [89]. The style cleanup was conducted by Ziyang Pang, this author, and Jessica Jones, with contributions from Patricia Teixeira. Similarly, version 3.1 was an effort by this author to reorganize the code directory structure and build system to ease the learning process for the large incoming class of graduate students in mid 2017. Versions 3.2 and 3.3 followed with the final thesis contributions by Jessica Jones [60] and Ziyang Pang [61], as well as an enhanced flutter module written by this author.

During the development of UM/NAST 3.1 it became apparent that the existing code structure was not conducive to design and optimization problems. At that time, such problems required the user to write an interface in a scripting language (e.g., Python or Matlab) which would write or modify an xml input file. Next, the user executed a system call of the UM/NAST executable to run the input file and obtain an output file. Finally, the user parsed the output either manually or via another interface for further post-processing. Security concerns regarding system calls notwithstanding, this workflow, and more generally the code structure, presented a number of obstacles for practical design and optimization applications.

A file-based workflow presents challenges to the efficient execution of batch or optimization problems. An obvious limitation imposed by this system is the performance bottleneck created by input/output (I/O) operations. I/O operations, in general terms, are serial, so writing large amounts of data could yield a significant performance penalty. A less obvious consequence of reading and writing input and output files is that these files prevent parallelism. Many design problems either require or benefit from parallel execution to make the problem computationally feasible or to enable more studies in the same amount of time. Reading and writing files makes such a parallel problem more difficult and in some cases impossible to realize.

Existing files with the same filename either block parallel execution or may result in files being overwritten unintentionally, yielding wrong simulation data. Even if separate filenames are used for the individual threads, housekeeping would be significantly more difficult because of the use of file I/O. Finally, writing and parsing files presents an additional step that the simulation must conduct before utilizing data. An Application Programming Interface (API), by contrast, can skip this step and grant direct access to data that has been stored in system memory.

Equally, the general code structure of UM/NAST 3 complicated the setup of design and optimization problems. The work presented in this thesis, in particular, foreshadowed the need for large numbers of concurrent analyses, as well as the recovery of gradients. UM/NAST versions prior to version 4.0 used a functional programming paradigm, meaning that the code was written as a series of functions to which input data was passed and from which output data was retrieved. While a functional paradigm does not necessarily exclude an optimization workflow, it posed a challenge when considering the changes to the code needed to recover aeroelastic sensitivities. Furthermore, an object-oriented programming paradigm appeared to be more suitable to conducting large numbers of simulations in parallel, due to more contained data management. Finally, many modern optimization problems require Python bindings. Python has become a common denominator in optimization and interoperability with it, more often than not, may be required, especially when collaborating with other research groups. However, Python is an object-oriented language and functional programming is not considered "pythonic[1]." Creating a Python wrapper for the previous functional UM/NAST implementation would therefore have been particularly difficult yielding complex workflows for setting up more than simple analysis problems.

A further hurdle to design and optimization problems was the absence of a com-

[1]The term pythonic is often used by Python programmers to denote code that adheres to Python design principles or philosophy.

plete API within UM/NAST prior to version 4.0. As UM/NAST was initially designed to read an input file, no API was conceived to allow users to assign or modify input data or solver options. The absence of an API thereby complicated the removal of the input/output file workflow and ultimately required a more extensive rework of the UM/NAST code.

UM/NAST 4.0 was designed and implemented as an attempt to address the previously discussed issues as well as improve code extensibility with external tools such as external aerodynamic solvers, feedback controllers, gust models, etc. The redesign was conducted and implemented by this author, with help towards the end of the development cycle by Cristina Riso to complete the high-level API, reorganize the trim solver, regression testing, and documentation. In contrast to UM/NAST 3, an object-oriented paradigm was adopted for version 4.0. An extensive API was written to replace the previous input file. While writing an input file interface is currently possible, it is not the primary means of inputting data. A Python API, which closely resembles the underlying C++ API, was written as the main user interface. It can be run passively via Python scripts and functions or interactively via IPython, etc. The Python interface also enables batch analyses (both serial and parallel) that were previously not possible. Finally, gradient capabilities, as well as enhanced linearization and stability/flutter solvers were added to the software in version 4.0.

UM/NAST 4.2, which was used to generate the results in this thesis, is a comparatively minor iteration of the software. The general code structure remains the same as version 4.0. New features were added to support other research projects, as well as to improve the API for design and optimization work[2].

[2]Versions 4.1 and 4.2 were jointly authored between this author, C. Riso, D. Sanghi, M. Pereira, and L. Lustosa.

10.2 Code Design

Version 4.0 introduced an object-oriented design to UM/NAST. This resulted in a utilitarian subdivision of the software into two main types of structure: A model and a solver or analysis. While UM/NAST contained a model class and a simulation class prior to version 4.0, they predominantly served as data storage that could be passed as an argument to functions. The new software design expands on both classes and transforms them into sovereign entities. Each class contains member data and a set of member functions that manipulate the member data. In that sense, every class can be viewed as an object with a set of properties and methods to modify its properties.

10.2.1 Model and Solvers

The model class contains the geometrical and property information required to conduct aeroelastic analyses. Or when viewed from another perspective, it defines the aircraft that will be analyzed by the solver. The model class contains a series of high-level API functions for user input, as well as functions that process the input into data the various solvers require.

The different solver classes contain member data and methods, which together with model data enable geometrical nonlinear aeroelastic solutions. Inheritance is used widely across all solvers (Figure 10.1) and not all solver classes defined within UM/NAST 4.0+ are designed for direct user interaction. Base classes define basic functions and data available to all solvers which inherit from them. While inherited solvers such as the static solver (StaticSolver) are intended for user interaction, the base classes StructuralSolver and CoupledSolver from which StaticSolver inherits are not. The inheritance from these solvers enables the extensive reuse of code throughout the UM/NAST framework, while making individual member data and functions available to a large set of solver classes. Moreover, multiple inheritance is used by

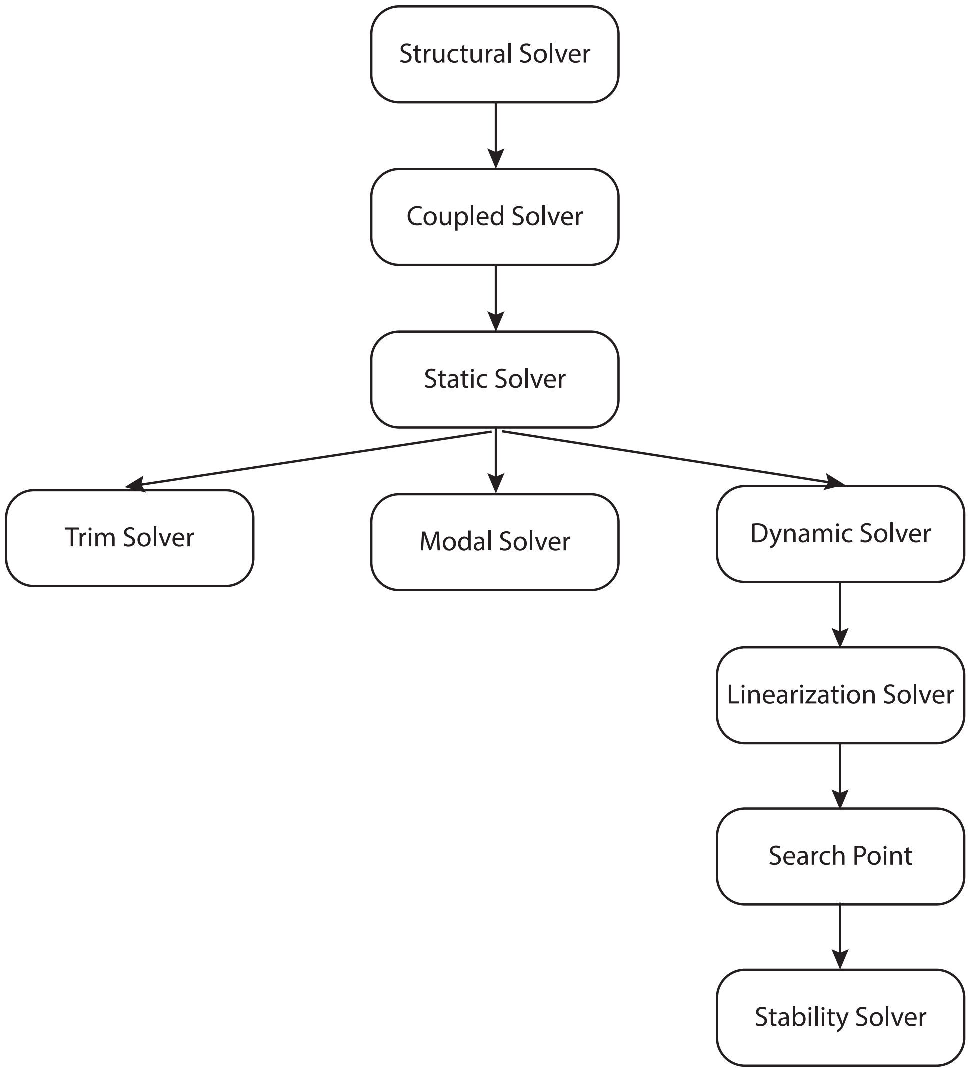


Figure 10.1: Inheritance structure of the solvers provided in UM/NAST.

several solvers to enable both their own analyses as well as those of their respective parents. The dynamic solver (DynamicSolver), for example, inherits from the static solver, therefore enabling both static (e.g., for initial conditions) and dynamics solutions.

The model and the solver, however, offer little utility on their own. While the model can be created and updated without the existence of a solver, the reverse is not true. While solver options can be assigned without a defined model, an attempt

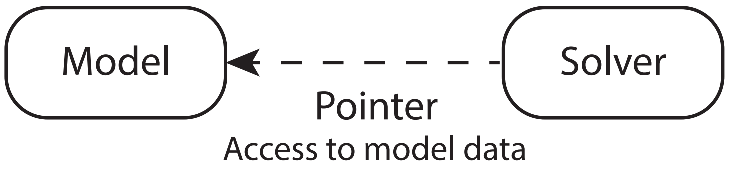


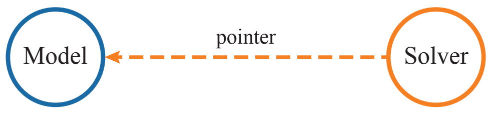
Figure 10.2: Linked relationship between the model and solver classes.

to run a solution without a model will fail due to unaccessible memory, as model information is required for all aeroelastic solutions. To ensure the accessibility of the necessary model data, each solver creates a link to an associated model (Figure 10.2) via a C++ pointer. The access via pointer ensures that copies are made, thereby reducing memory footprint and improving performance.

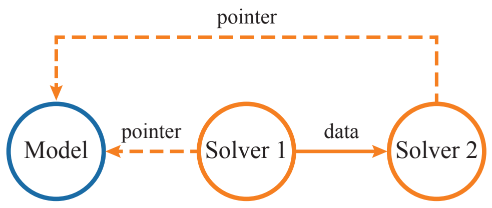
While the model and solver classes coexist to enable aeroelastic analyses, their relationship, by design, is unequal. A solver may only link to one model at a time, while a model can be linked to a practically infinite amount of solvers. This relationship enables a variety of solvers to be executed concurrently working off of a single model's data. Such a workflow may be used in a design problem, where the aircraft parameters are modified and updated centrally, yet the resulting data is available to all solvers (e.g., different load cases).

10.2.2 Design Usage Patterns

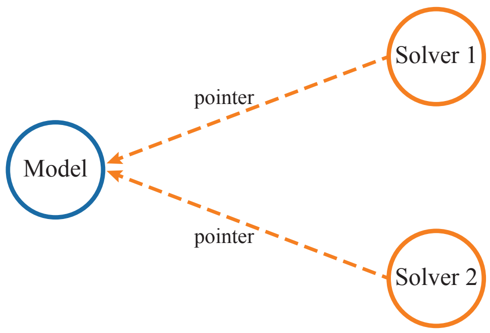
During the design of UM/NAST 4, a number of usage patterns were considered to be required for design and optimization problems utilizing nonlinear aeroelasticity (Figure 10.3). As is the case in any programming problem, there are many different approaches to solving problems. As such, many more viable usage patterns may exist than are listed here. The purpose of the listed usage patterns was not to be



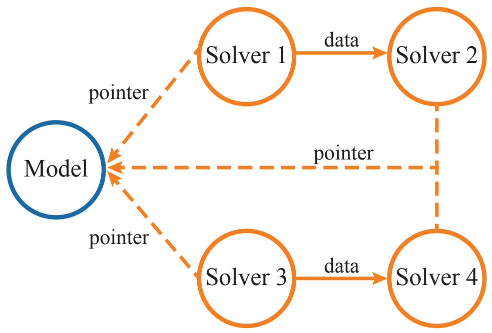
(a) Standard/legacy pattern



(b) Serial pattern



(c) Parallel pattern



(d) Parallel-serial pattern

Figure 10.3: Usage patterns considered during the design of UM/NAST 4.

comprehensive, but rather to address scenarios that users needed to, but could not resolve using previous iterations of the software.

The first usage pattern (Figure 10.3a) could be described as the standard or legacy pattern as it constitutes the legacy workflow of UM/NAST. A single solver links to a single model to produce a single set of output data. This pattern accounts for a large number of expected user-required scenarios.

The serial pattern (Figure 10.3b) extends the legacy pattern by further solvers. In its most basic form, two solvers are linked to the model. After the first solver completes, data is transferred to the second solver as an input to the second solution. The second analysis is run and the data from both solver can be used for post-processing. A common example of this pattern is a trim analysis followed by a flutter/stability analysis.

Another required workflow is the parallel pattern (Figure 10.3c). Parallel problems are particularly important in design, where several analyses (e.g., independent load cases) must be evaluated that have no dependencies on each other. The ability to run these cases concurrently can make the difference between a practical and inpractical design problem. Many parametric/batch studies are examples of this workflow, as each analysis is independent of the others.

Finally, the parallel-serial pattern (Figure 10.3d) is a hybrid of the last two systems and its feasibility follows from them. This workflow is of particular importance to this work, as the flutter constraint utilizes this principle. For example, a flutter constraint including rigid body degrees of freedom requires the evaluation of a trim condition before determining the system stability.

10.2.3 Extensions

UM/NAST is one of the main tools used at A^2SRL. As such, requirements on its use are both wide-ranging and specific to project foci. For example, while one project may require the inclusion of a feedback controller, which in turn requires state data from (simulated) sensors, another project may not require any of this functionality. In fact, the inclusion of said functionality into the core UM/NAST code would bloat the codebase and result in an unmanageable toolset. Thus, the descriptions of “UM/NAST” provided so far refer to the core functionality of the toolbox, referred to as the UM/NAST Kernel.

To avoid code bloat, the UM/NAST framework functionality was divided into functional components, which are maintained in separate repositories. I developed a plugin system within the UM/NAST Kernel which allows particular building blocks to be swapped at run time, whithout recompilation using polymorphism. This permits users/developers to customize:

- aerodynamic solvers
- feedback controllers
- simulated sensors
- atmospheric models

and tailor them to project needs while minimizing the impact on other projects. These plugins are defined using baseclasses that determine the basic functionality of aerodynamic solvers, etc. for the Kernel. In this manner, the Kernel has no dependencies on the various plugins, thus, reducing the maintenance overhead of unrelated plugins.

10.2.4 Templated Code

Templates, by default, are not compiled unless they are called. This creates practical concerns when developing code or the installing extensions, as every object must be compiled. This would result in unacceptable compilation times and their associated loss of productivity. As such, UM/NAST utilizes explicit instantiation, in which a handful of known types are provided for compilation of the objects. The compiler takes these types and compiles objects for the double, complex step, and reverse-mode AD types and combines them together to a binary library (Figure 10.4). Then, when a user builds a UM/NAST extension, the library can be linked, reducing the total compilation time.

10.2.5 Impact of Code Design on Performance

Execution performance constitutes an important metric that can determine the feasibility of design or optimization problems. Large performance gains were achieved by the rewrite of UM/NAST from Matlab into C++ (version 2.0) due to the inherent

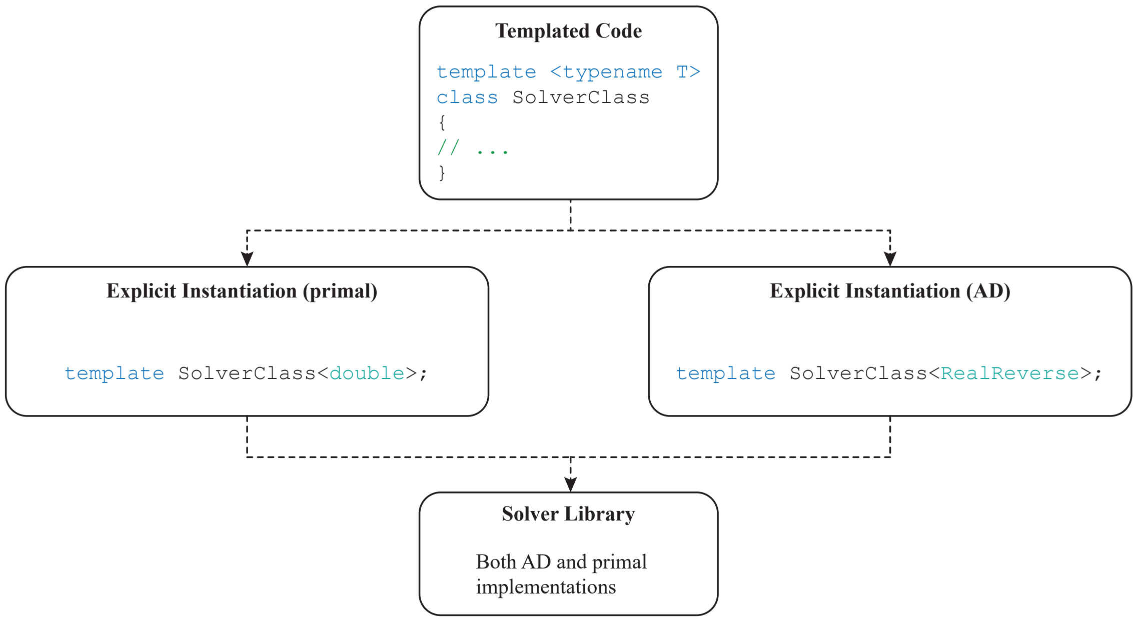

Templated Code
template <typename T>
class SolverClass
{
// ...
}
Explicit Instantiation (primal)
template SolverClass<double>;
Explicit Instantiation (AD)
template SolverClass<RealReverse>;
Solver Library
Both AD and primal implementations

Figure 10.4: Compilation of the templated code using explicit instantiation to obtain a binary library.

performance benefits of a compiled language compared to Matlab. While single-threaded performance was not the primary concern during the redesign in version 4.0, performance remains paramount to conducting practical and scientifically interesting design problems. A comparison of individual code sections to past versions is not feasible due to the significant change in code structure. However, it is possible to quantify overall performance.

Overall, version 4 does experience a performance increase compared to previous versions. Particularly the removal of the file-based I/O weighs heavy in the performance gains. For every static simulation run in UM/NAST version 3, approx. ten solutions can be run in version 4. These improvements do not necessarily originate from faster code execution—version 2 and 3 are very memory efficient and passed variables via reference with little computational overhead—but rather by eliminating unnecessary program sections. This performance gain becomes more pronounced when running batch analyses, especially in parallel. Any batch job in version 4 does not require model updates (processing of user inputs to solver information) unless the model is changed (i.e., model property modification). Version 3 requires a reload of the input data and a model update for every analysis resulting in substantially more operations than version 4. Further improvements are achieved by running version 4 batch studies in parallel. Previous versions did not offer this capability[3].

[3]Version 3.0 offered a parallel execution for flutter searches only.

CHAPTER 11

MDO–NAST

Chapter 10 describes the structure of the UM/NAST Kernel and how AD was implemented to obtain coupled nonlinear aeroelastic sensitivities. However, the process of setting up a gradient analysis in UM/NAST with CoDiPack is fairly involved and would require a custom C++ implementation and subsequent Python wrapping for every gradient analysis. Additionally, this would also entail a recompilation every time the problem is changed. As such, I developed a set of helpers into a separate library, MDO–NAST [**?**], for this thesis. MDO–NAST offers gradient helpers for the static, modal, dynamic, and stability/flutter solvers. These helpers substantially ease the process of setting up and executing a gradient analysis using UM/NAST.

11.1 Common Concepts

The common idea behind all tools within MDO–NAST is that the user has a function of interest (or multiple) f_i and has defined the design variables x_j of the problem. MDO–NAST then provides utilities to determine both the functions and the derivatives $\frac{\partial f_i}{\partial x_j}$ for all i and j.

Differing from other analysis tools, the functions and variables of interest can be set to any UM/NAST quantity, as long as proper read/write permissions have been granted to external objects. This is achieved by a dynamic module system which permits the user to define the design variable assignments and functions of interest as templated C++ modules (see Figure 11.1). These classes can be loaded into MDO–NAST dynamically and used by the gradient helpers independent of the datatype required for the respective gradient analysis. The user-defined variable assignment (gray) derives from a variable assignment base class. This permits the user to write a dynamic library containing a custom variable assignment and dynamically load it into MDO-NAST. At runtime, MDO-NAST then executes the user code for assigning the design variables to the model or solver object. The same process is applicable for the function of interest (function evaluation, gray), which also builds on a common base class and executes a user-defined function at runtime.

The purpose of the individual gradient helpers is to enable the easy evaluation of derivatives without the user requiring a deep understanding of finite difference, complex step, or AD processes (see Chapter 2). Despite the multitude of gradient helpers, the AD solution is intended to be the main workflow with the other helpers primarily serving as reference solutions for validations (see Chapter 14). Each solver type (static, modal, etc.) has three gradient helpers available, one for each derivative type. Common to each gradient helper is the function evaluation, which is conducted using the *double* implementation of the respective solver for reasons of computational efficiency.

To reduce overhead and simplify the setup of every analysis, MDO-NAST classes link to existing UM/NAST models and solvers in a similar fashion to the method described in Chapter 10. This results in compatibility between a UM/NAST and MDO-NAST workflows, constituting a natural extension of the established capabilities and practices of UM/NAST 4.0 and avoids a redefinition of options, loads, etc.

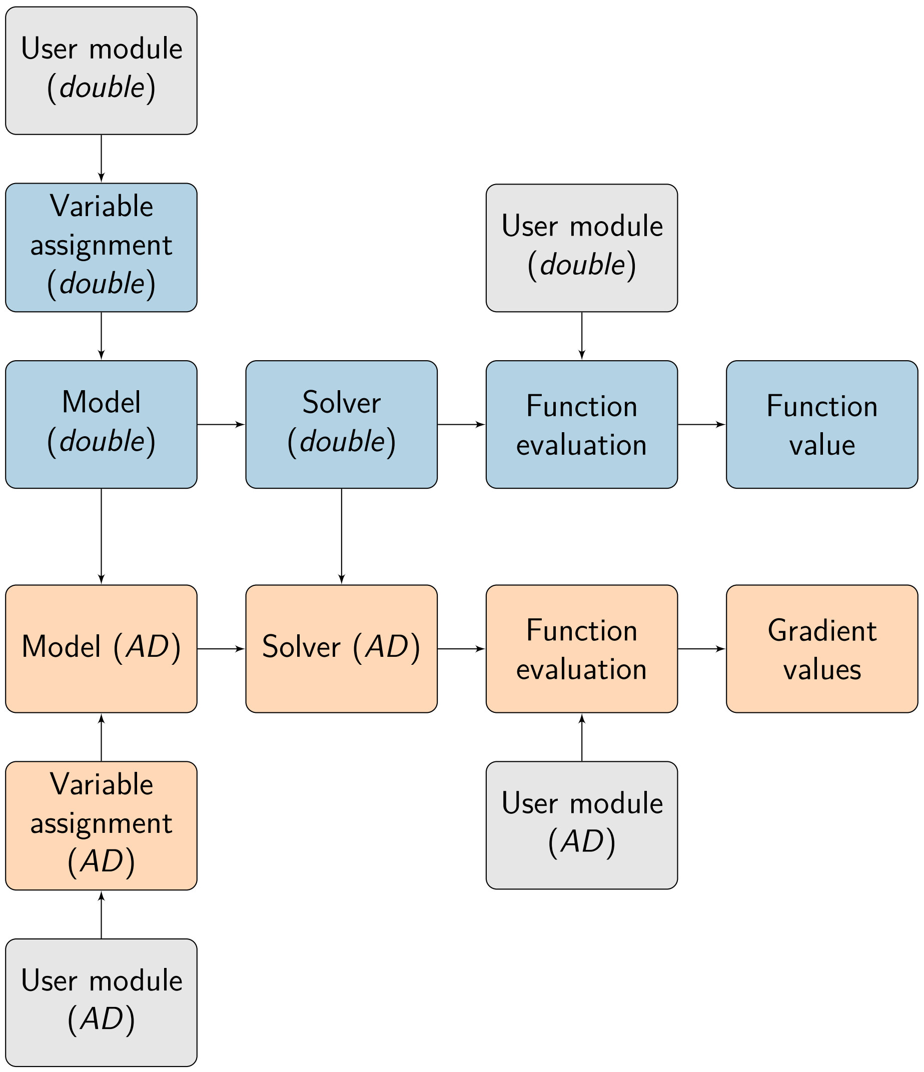


Figure 11.1: Common solution structure for the gradient helpers exemplified by an AD gradient helper.

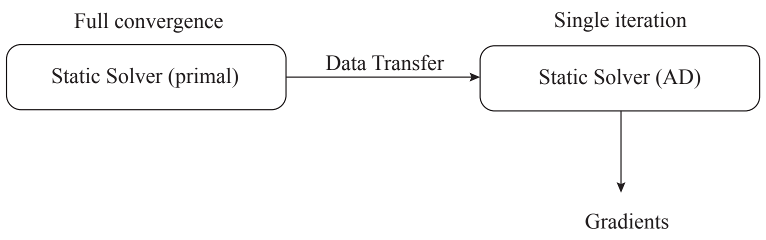


Figure 11.2: Efficient hybrid methodology for determining gradients of the geometrically nonlinear static solution.

11.2 Static Gradient Helpers

The static gradient helpers form one of the simpler MDO-NAST utilities, while serving as the basis of all subsequent ones. Differing from UM/NAST's implementation, the more complex gradient helpers are not derived from the static solver utilities, but rather from one central gradient helper class for each derivative method (*GradientHelper*, *GradientHelperComplex*, *GradientHelperAD*).

For the AD solution, the entire convergence cycle is run in AD by default. As exemplified by the studies in Chapter 14, this results in a substantial performance penalty compared to the simple function evaluation. To counteract this I developed a hybrid-AD solution, similar to other fixed-point AD approaches [90], in which convergence of the geometrically nonlinear static problem is first achieved using the function evaluation, before transfering data to an AD solver (Figure 11.2) and conducting an AD iteration to obtain gradients. A more detailed study of the effects on the computational time can be found in Chapter 14.

11.3 Modal Gradient Helpers

By default, the UM/NAST modal solver constitutes a simpler solution than the static solver with the option to use any nonlinearly deformed state as the reference condition for a modal analysis. MDO-NAST offers a helper to obtain gradients from modal analyses about either the undeformed condition or a nonlinearly deformed state. The modal helpers simply run a modal or deformed modal solution in AD (or complex step) without using the hybrid AD solution. This is due to a different solution structure for the modal solver, which would require a more involved solution to the hybrid approach. While it certainly is not impossible to implement, this may be considered an area for future improvements.

11.4 Dynamic Gradient Helpers

The dynamic and the static solvers follow a similar solution process, as the static solution is obtained by stepping in pseudo-time. At first glance, it may appear that the hybrid AD method should be applicable to the dynamic gradients. However, this has not been tested for the dynamic solver and it remains doubtful that the coupled gradients would be accurate without a convergence process. As such, the dynamic gradient helper was not written to include the hybrid AD solution, similar to the modal helpers. While I implemented the dynamic gradient helpers within MDO-NAST, no benchmarks or problems have been run with them as this lies outside the scope of the current work.

11.5 Search Point Gradient Helpers

The solution process for the search point gradients builds on the static gradient helpers. This is possible because the search point runs a static analysis followed by a linearization and a stability solution (Figure 11.3). As such, the most efficient way to determine the stability gradients is to run a primal static analysis, transfer the data to an AD solver, solve a single static iteration, followed by a linearization and the stability solution. The results of this approach are discussed in Chapter 14.

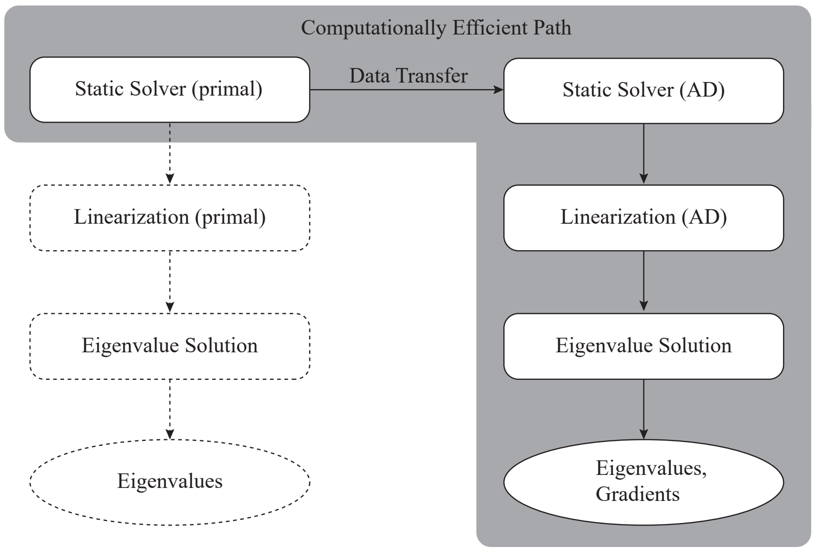


Figure 11.3: Efficient hybrid methodology for determining gradients of the geometrically nonlinear stability/flutter solution.

11.6 OpenMDAO Components

While MDO-NAST is a Python library, by default, it serves as a helping interface to determine gradients using the UM/NAST framework using serialized inputs and outputs. This means that although a variety of functions or variables of interest may be used, they are stacked to obtain input and output vectors of the form:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \tag{11.1}$$

$$f = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}. \tag{11.2}$$

In most practical scenarios, the individual entries x_i (or f_i) may actually contain subvectors instead of scalars. While the simplest solution to using MDO-NAST within OpenMDAO would have been a simple *Component* wrapper around the MDO-NAST gradient helpers, the actual OpenMDAO problem would become more complex. To assign the individual inputs from seperate OpenMDAO components or variables a so-called *mux* component would be necessary for every corresponding MDO-NAST component. A mux component takes inputs from several separate sources and combines them together to a single output. This would entail additional work for the user during the problem setup phase.

To avoid this, the MDO-NAST components use individual inputs instead of the serialized data required by the gradient helpers themselves. The individual inputs, which can vary from scalars to vectors, are then assembled into a serialized vector for further use by the gradient helper. By default, the MDO-NAST components use the AD gradient helpers, although the complex step helpers are available as well. The name and lengths of the individual component inputs and outputs are user-defined within the variable assignment and function modules. This permits a simpler assembly of problems using descriptive variable names, rather than requiring the manual stacking of variables and index arithmetic.

Part III

Numerical Studies

CHAPTER 12

Aeroelastic Models

The numerical studies in this work share a common set of numerical aeroelastic models. Simpler beam-based studies rely on a Blended Wing Body (BWB) configuration, while the multi-fidelity studies utilize the more complex uCRM transport aircraft configurations. This chapter describes the individual models used for the numerical studies in this thesis.

12.1 Blended Wing Body

A published BWB reference configuration [4] based on the High Lift over Drag Active (HiLDA) Wing [91] is widely used throughout this work. While it is a relatively simple configuration, Su and Cesnik [4] showed that it exhibits interesting flutter and post-flutter behavior, including body-freedom-flutter and limit cycle oscillations (LCO). As such, the vehicle provides a desired characteristics for several of the benchmark and test cases within this work. The vehicle consists of a trapezoidal body and swept uniform-chord wings (Figure 12.1). The wing contains three control surfaces that are used for trimming the aircraft and maneuvers. The wing stiffness, mass, and aerodynamic properties were taken from Su and Cesnik [4].

The BWB structure is modeled using the strain-based nonlinear beam elements within UM/NAST. Aerodynamics are modeled using strip theory with Prandtl-

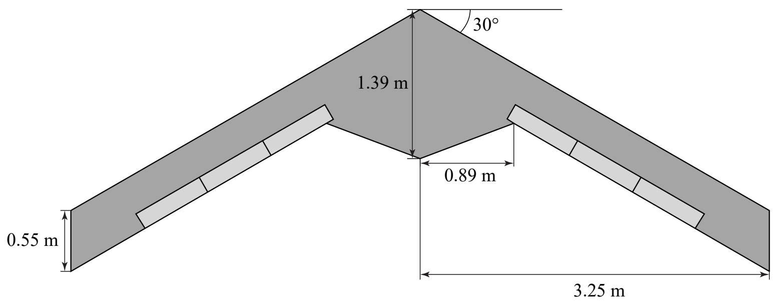


Figure 12.1: BWB configuration planform.

Glauert corrections and Peters' finite state inflow to account for unsteady aerodynamics [53].

12.2 Undeflected Common Research Model

The uCRM, designed by the MDO Lab at the University of Michigan [33], was used within this for the multi-fidelity problems. Brooks and coworkers designed the two configurations (aspect ratios 9 and 13.5) using high-fidelity aerostructural (TACS and ADflow) optimization without a flutter constraint. As the aircraft experiences large deformations (Figure 12.2), it offers a compelling case study for applying the geometrically nonlinear flutter constraint within a multi-fidelity optimization problem. Furthermore, the existence of two uCRM models with differing aspect ratios and wing spans permits a study of how the flutter constraint may drive the design for a high aspect ratio wing compared to a more conventional aspect ratio.

Because these models are used in a multi-fidelity context, several different numerical models were used and/or created for these studies. The aerodynamics of the higher-fidelity problem are modeled using the OpenAeroStruct vortex lattice method (VLM) solver. To create meshes for this solver, the aerodynamic CFD mesh provided by Brooks and coworkers was sliced and the chord, twist, and leading edge

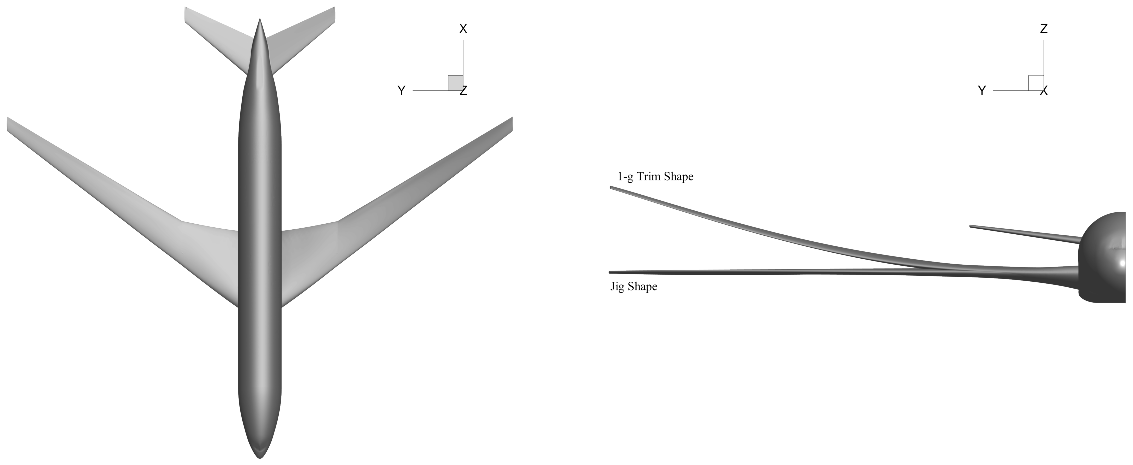
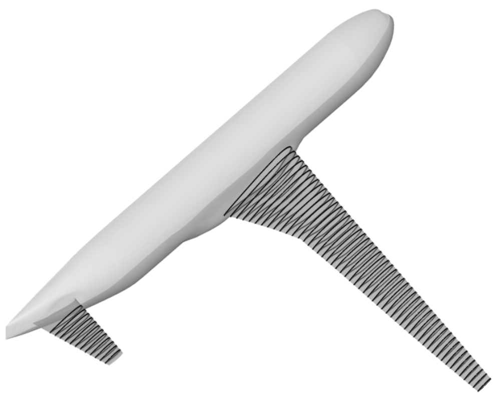


Figure 12.2: Top and front views of the uCRM 13.5 optimized configuration by Brooks and coworkers.

positions were obtained (Figure 12.3), and the VLM meshes were generated from this information.

The shell-based structural mesh was taken from the open-sourced data provided by Brooks and coworkers [33] (Figure 12.4). For the lower-fidelity beam model, reference beam locations were set at 50% chord location.[1] The aerodynamics of the beam-based model are provide, as before with the BWB, via strip theory and Peters' finite state aerodynamics [53].

[1]The RBE3 nodes shown here were created by Joshua Deaton (AFRL) based on reference beam locations I provided.



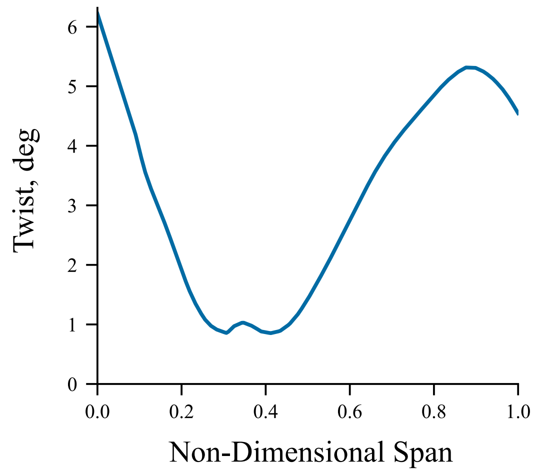


Figure 12.3: Slices and twist distribution obtained from the CFD meshes for the uCRM 13.5.

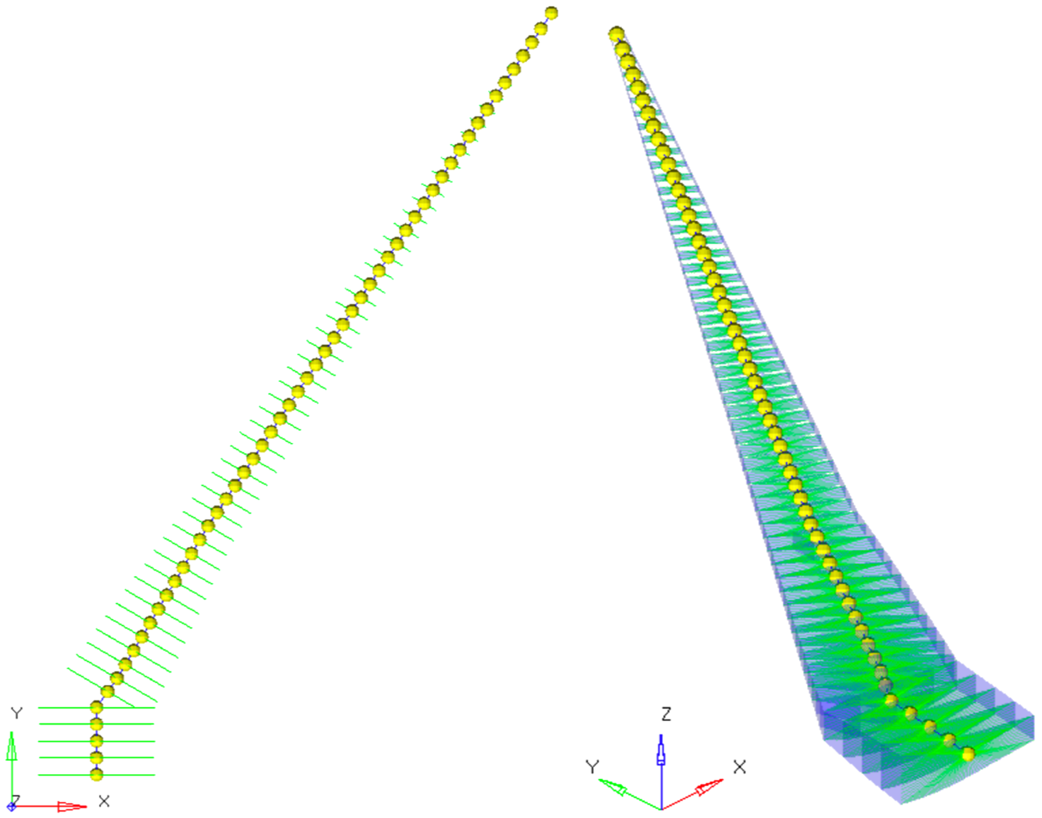


Figure 12.4: Top and isometric view of the wing structural mesh with the beam structural nodes.

CHAPTER 13

Flutter Prediction

A new algorithm for predicting flutter was presented in Chapter 4. In this chapter, I study the accuracy and efficiency of the proposed method. The studies utilize the BWB (Chapter 12). Despite its simplicity, the BWB exhibits body-freedom flutter as well as nonlinear behavior (LCO), making it a suitable test case.

13.1 Flutter Point Prediction

Flutter simulations for the BWB were conducted using the existing methodology [4], a mode-tracked solution without flutter, and the proposed flutter prediction algorithm. These simulations were conducted for a varying discretization of the pseudo-univariate search space without the inclusion of rigid body degrees of freedom (DOF), while maintaining consistency in search range and discretization between the respective algorithms.

Comparing the prediction error (Figure 13.1a) of the different algorithms shows significant advantages for the new flutter method. The surrogate-based approach converges for very few search iterations. By contrast, the existing (linear interpolation) method requires significantly more search iterations to achieve the same level of accuracy. For example, the proposed method achieves a relative error at 30 search iterations that the previous method requires approximately 200 search iterations to achieve. This increases computational efficiency by nearly an order of magnitude.

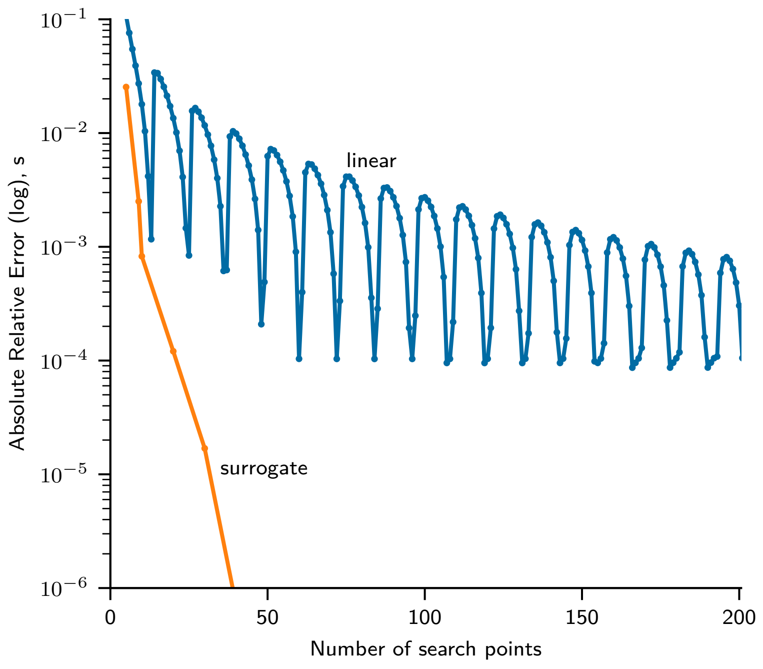
While the existing method converges to an accurate flutter prediction, it is unable to achieve the same accuracy as the proposed, surrogate-based algorithm.

Furthermore, the existing method is accompanied by significant amounts of uncertainty, while the proposed method converges near monotonically with the number of search iterations. Limitations of the new algorithm are illustrated by Figure 13.1b. At 5 search iterations the surrogate predicts multiple inflection points within the unstable mode, thus reducing the accuracy of the flutter prediction. Noticeably, for 10 search iterations the severity of the inflection points decreases, while the search using 20 iterations shows a better resolution of the damping values. Despite the inflection point in the analysis using only five points, the discrepancies between the solutions are small. This is particularly remarkable, given that a five-point discretization of the search space is exceedingly coarse. Furthermore, looking at the comparison of V-g diagrams for 20 and 30 search iterations (Figure 13.2) shows little difference in the surrogate used for flutter prediction. The exact number of search iterations needed to achieve an accurate flutter prediction depends on the size of the interval as well the discretization pattern (uniform, Chebychev, etc.) and therefore is problem specific.

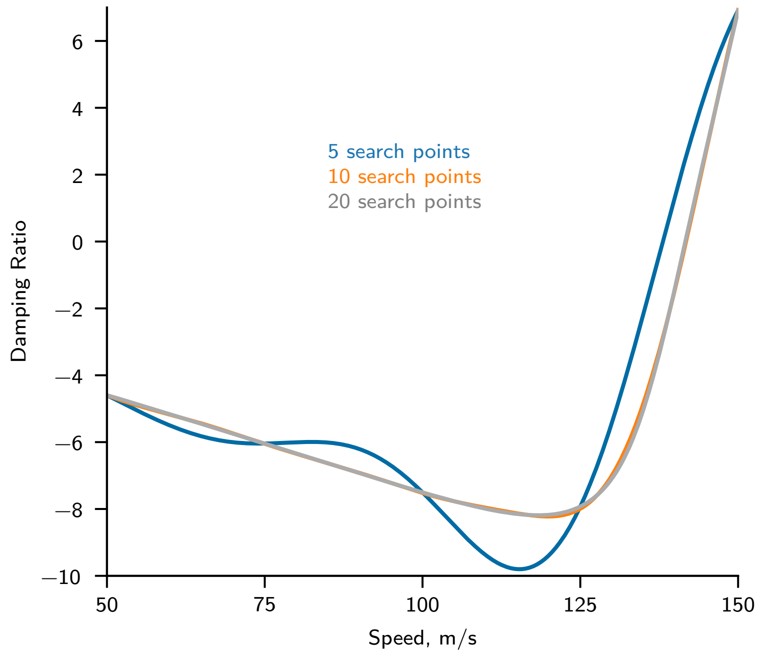
Comparing the accuracy of the mode-tracked, surrogate-based algorithm to the mode-tracked algorithm with linear interpolation shows a similar trend to previous comparison. The error used for this work is the absolute relative error, defined as:

$$\epsilon = \left| \frac{v_{exact} - v_{predicted}}{v_{exact}} \right| \tag{13.1}$$

The surrogate-based prediction method significantly outperforms the mode-tracked algorithm with linear interpolation. This is not surprising, as the mode tracking only prevents interpolation errors due to mode crossings. While the mode tracking may reduce the error in the case of a mode crossing, it does not inherently reduce the interpolation error if mode crossings do not exist on the interpolation interval. The



(a) Comparison of the prediction error of the proposed compared to the existing method.



(b) Comparison of the V-g diagram of the unstable mode surrogate prediction for different search space discretizations.

Figure 13.1: Study of the flutter prediction method accuracy.

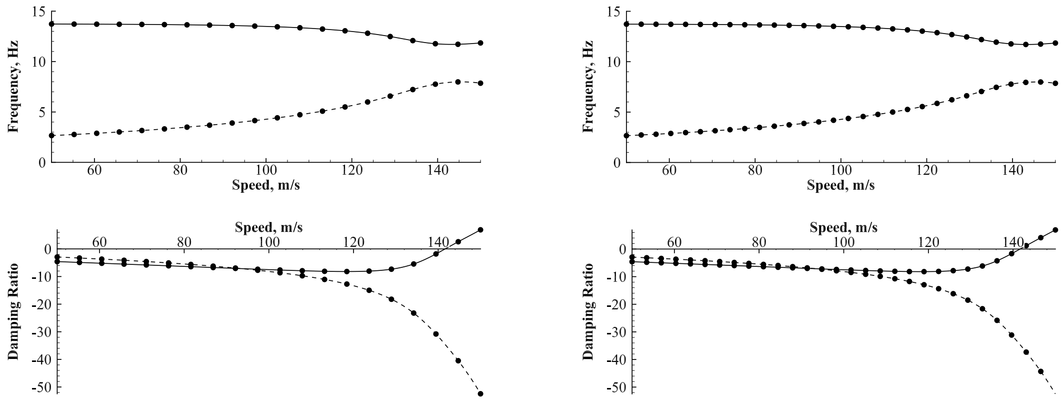


Figure 13.2: Comparison of the V-g diagram for the trimmed BWB for 20 search iterations (left) and 30 search iterations (right).

present example of the BWB does not exhibit mode crossings near the instability and the linear interpolation is thereby not affected. However, the linear mode-tracked solution regularly crashes for sparse search sampling due to failed LU decompositions, caused by insufficient interpolation of the flutter point conditions.

It is noteworthy, however, that the kriging surrogate does require a mode tracking algorithm to reliably predict the flutter point. While the linear interpolation only is affected by mode crossings near the instability, the surrogate is created from all damping values of a mode. If the "mode" is chosen as the damping values closest to zero, mode crossing may occur away from the flutter point, cause inflections (as seen before in Figure 13.1b) and affect the flutter prediction.

Finally, comparing the execution wall time over the number of search iterations of the proposed method (Figure 13.4) to the existing method illustrates the efficiency of the new flutter algorithm. Three seperate computers were used for this study: Computers B, and C (see Appendix C). Computer A features two Intel Xeon 2650 CPUs with 8 physical cores (16 threads) each and 64 GB of RAM. Computer C offers four Intel Xeon 2699v4 CPUs with 16 physical cores (32 threads) per processor and 256 GB RAM. While the RAM is listed here, UM/NAST requires on the order of Megabytes of memory to execute the presented solutions, meaning that the system

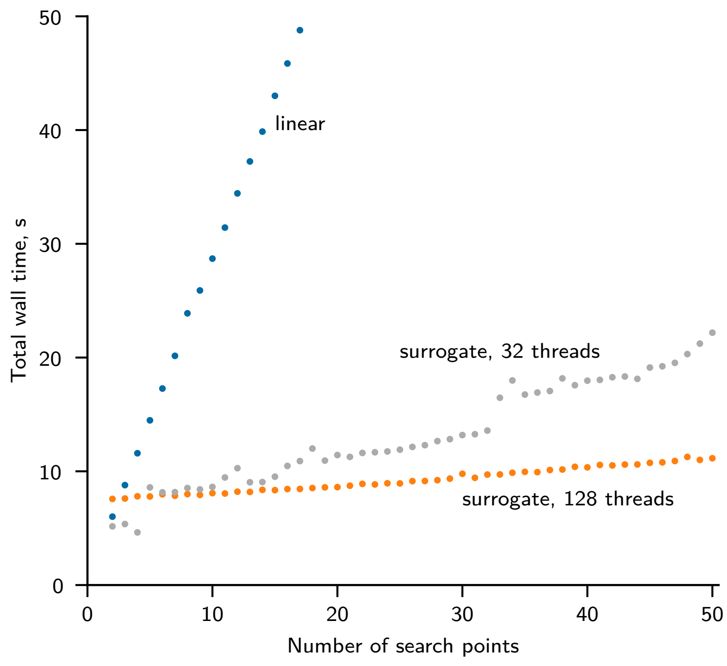


Figure 13.3: Comparison of the execution wall time for the new and existing flutter algorithms run on Computer B [32 threads] and Computer C [128 threads].

memory is not the bottle neck in this scenario. It should be noted, that while the original reference was written in Matlab, the benchmark here was written in C++ to maintain parity between the two methods. Due to the data-dependence of the previous algorithm, and increase in search iterations directly results in an increased execution wall time. The new algorithm, due to data-independence, scales well w.r.t. an increase in flutter search iterations. Comparing the results between the two computer configurations illustrates that the mode-tracked, surrogate-based algorithm is primarily limited by the number of available processor threads. The slight slope of the surrogate-based algorithms is caused by the scaling of the mode tracking algorithm, which requires more time as more search points, and thereby more mode comparisons, are added.

13.2 Flutter Boundary Prediction

In addition to the trimmed BWB, flutter studies were conducted for the BWB with the body subject to a clamped condition. This configuration constitutes the fixed search path strategy outlined in Chapter 4 and requires the definition of boundary conditions a priori, as they are not constrained by trimmed flight. Multiple flutter searches were run for varying angles of attack, the resulting data collated and an extended V-g diagram created (Figure 13.4).

The nonlinearity of the flutter problem is clearly visible in the extended V-g diagram, as a change in body angle of attack clearly changes the resulting flutter dynamic pressure. Furthermore, the high sensitivity of the flutter boundary w.r.t. the angle of attack ($\frac{\partial q_F}{\partial \alpha}$) should be noted. As such, if a fixed path flutter search is conducted with a constant angle of attack setting, the choice of that setting is vital as even small deviations result in large changes in the predicted flutter dynamic pressure. Finally, this behavior is exhibited by the BWB despite moderate wing deflections, indicating that geometrically nonlinear effects may contribute to the flutter analysis far earlier than the 10–15 percent relative tip deflection typically assumed from static analyses.

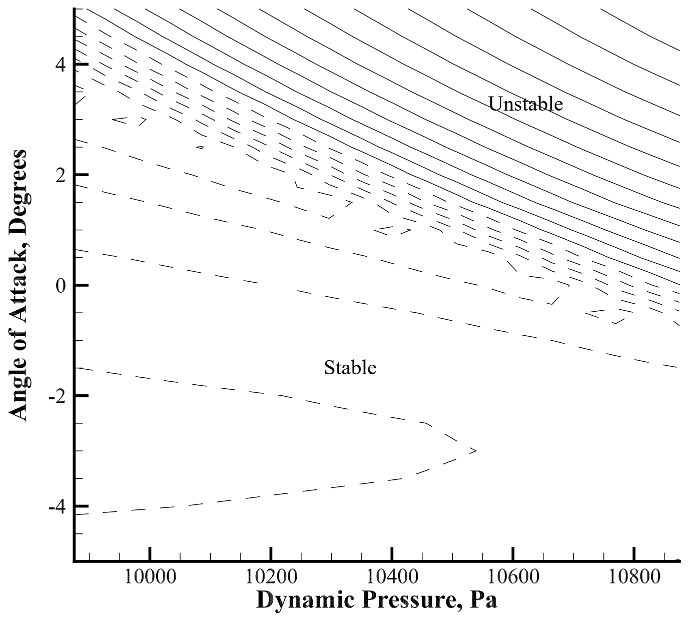


Figure 13.4: Extended V-g diagram, depicting the dynamic pressure vs. angle of attack, for the BWB.

CHAPTER 14

Verification of Nonlinear Aeroelastic Gradients

The efficient and accurate determination of gradients is imperative for the flutter constraint. Both the accuracy and computational efficiency of the static tip deflection and flutter damping gradients were determined and benchmarked against reference results. The gradient of the most critical flutter value was evaluated with respect to the out-of-plane bending stiffness of the BWB wing.

14.1 Static Aeroelastic Gradient Verification

To verify the accuracy of the static solver derivatives, I compared the gradients of the static deflection with respect to the out-of-plane bending stiffness determined by UM/NAST against finite difference and complex step results. The numerical inaccuracy of the finite difference results (see Chapter 2) required a convergence study with variable step sizes to be conducted. From the convergence study a reference result accurate to five digits was obtained. Next, the gradients were determined using the complex step method and AD in reverse mode and compared to the reference finite difference value. This was conducted for two seperate span locations, at the Yehudi break (the junction between the inner and outer wing, Table 14.1), as well at the wing tip (Table 14.2).

Table 14.1: Comparison of gradient values predicted by the forward difference method, the complex step method, algorithmic differentiation in reverse mode for the static deflection at the Yehudi break. Agreeing digits are marked in bold.

Method	Gradient Value)
Forward Difference	$\mathbf{7.}85916172635695 \times 10^{-9}$
Complex Step	$\mathbf{7.7124391821827}2 \times 10^{-9}$
AD, Reverse Mode	$\mathbf{7.7124391821827}6 \times 10^{-9}$

Table 14.2: Comparison of gradient values predicted by the forward difference method, the complex step method, algorithmic differentiation in reverse mode for the static deflection at the wing tip. Agreeing digits are marked in bold.

Method	Gradient Value)
Forward Difference	$-\mathbf{8.04894}81746326419 \times 10^{-6}$
Complex Step	$-\mathbf{8.0494494321228}5 \times 10^{-6}$
AD, Reverse Mode	$-\mathbf{8.0494494321228}3 \times 10^{-6}$

For all span locations, the AD results replicate the complex step results to fourteen significant digits. While this is a very close agreement and indicates that the gradients obtained from UM/NAST can be used with confidence, the results do no agree to machine precision. The reason for this may be due to the complex step results. As discussed in Chapter 2, the ability of the complex step method to achieve machine precision is dictated by two conditions for the step size h. As Martins [46] notes, it may not be possible to fulfill both conditions, in which case the complex step method would yield a result below machine precision.

The finite difference results, as expected based on the discussion in Chapter 2, are not able to correlate as many significant digits. However, the results in Tables 14.1 and 14.2 do offer an insight into another problem facing the finite difference method. Both tables were generated by a single MDO-NAST run, so they share the same step size, as would be the case in an optimization problem. Interestingly, the finite difference results for the Yehudi break are only able to reproduce a single significant digit, while the results for the wing tip can reproduce four digits. This discrepancy highlights that even if the finite difference method were to yield acceptable results for a single value at a given step size, other functions of interest may be poorly captured.

14.2 Modal Gradient Verification

To verify the modal gradient capabilities within the toolbox, I examined the first two structural mode frequencies and their gradients with respect to the out-of-plane bending stiffness of the BWB wing member. Following the same methodology as in Section 14.1, I first conducted a convergence study to obtain a representative value for the gradients obtained with finite differences. Next, I compared this value to the data obtained from the complex step and AD analyses (Tables 14.3 and 14.3).

Table 14.3: Comparison of gradient values predicted by the forward difference method and algorithmic differentiation in forward and reverse mode for the first modal frequency. Agreeing digits are marked in bold.

Method	Gradient Value
Forward Difference	$-\mathbf{9.1455}316479915894 \times 10^{-5}$
Complex Step	$-\mathbf{9.145554871}5790915 \times 10^{-5}$
AD, Reverse Mode	$-\mathbf{9.145554871}7515379 \times 10^{-5}$

As with in Section 14.1, the finite difference approach is able to correlate few (here, 5) significant digits. The comparison between the complex step and AD results are close enough to verify the AD gradient recovery, but the number of correlating digits has been reduced to eight. Again, this likely stems from an inability to fulfill both criteria for the complex step method to achieve machine precision.

Table 14.4: Comparison of gradient values predicted by the forward difference method and algorithmic differentiation in forward and reverse mode for the second modal frequency. Agreeing digits are marked in bold.

Method	Gradient Value)
Forward Difference	$-\mathbf{2.81116}129396963 \times 10^{-5}$
Complex Step	$-\mathbf{2.81116314}352830 \times 10^{-5}$
AD, Reverse Mode	$-\mathbf{2.81116314}532585 \times 10^{-5}$

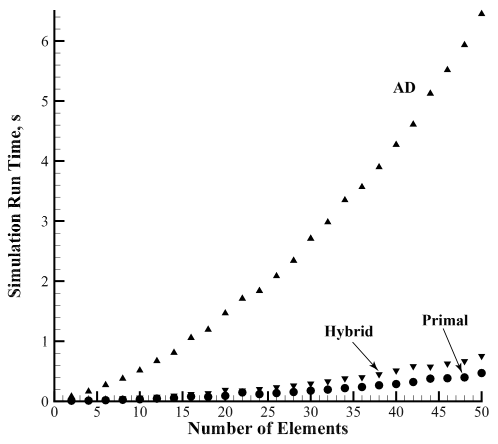
14.3 Flutter Gradient Verification

To verify the flutter gradient capabilities, I examined the KS aggregate damping value (the most critical damping value) and it's gradients with respect to the out-of-plane bending stiffnesses of the BWB wing member. Again, a convergence study of the finite difference results was required before a comparison to the AD data could be made (Table 14.5)

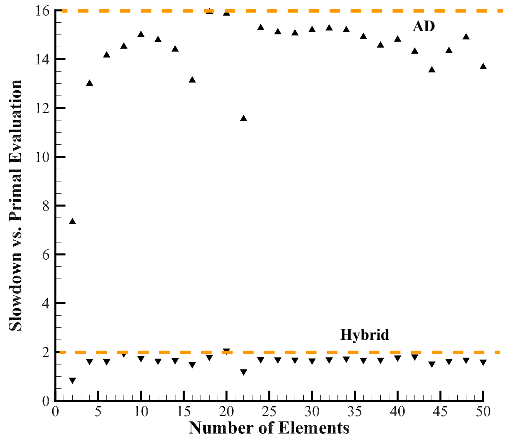
Table 14.5: Comparison of gradient values predicted by the forward difference method and algorithmic differentiation in forward and reverse mode for the most critical flutter damping value. Agreeing digits are marked in bold.

Method	Gradient Value (10^{-7})
Forward Difference	$-\mathbf{1.926}xxxxxxxxxxxxx$
AD, Forward Mode	$-\mathbf{1.9263576598010 3}23$
AD, Reverse Mode	$-\mathbf{1.92635765980103}42$

While no comparison between the complex step method and AD was conducted, the flutter gradients were verified against AD results in forward mode. The finite difference results again provide four digits of correlation, while the comparison against the forward mode results near machine precision correlation. As a result, the gradients obtained from UM/NAST can be used in an optimization setting with confidence.



(a) Run time



(b) Relative slow down

Figure 14.1: Comparison of the wall time required for a flutter search evaluation with different solution approaches (primal, AD, and hybrid-AD). The benchmark was run on Computer Configuration A (Appendix C).

14.4 Gradient Evaluation Performance

Large gradient-based optimization problems, in particular, require fast gradient evaluations. As such, I ran a series of benchmarks to assess the computational performance of the MDO-NAST-based gradient evaluations. The gradients were evaluated using the AD solver as well as the primal-AD hybrid method and the run times compared to that of the primal solution (Figure 14.1). The run times were averaged from 100 simulations. Figure 14.1a shows the absolute average run time of the individual solutions and Figure 14.1b the slow down factor compared to the primal solution. Clearly, the AD solution is more computationally costly than the primal solution, requiring between 12 and 16 times longer to obtain the solution including gradients. However, when applying the primal-AD hybrid method, the computational efficiency is greatly improved. While the hybrid method is still slower than the primal evaluation, it is less than two times slower, which is a very efficient result (slowdown factors of 3–4 are typically expected when determining gradients).

Furthermore, the scalability of the gradient evaluation with respect to the number

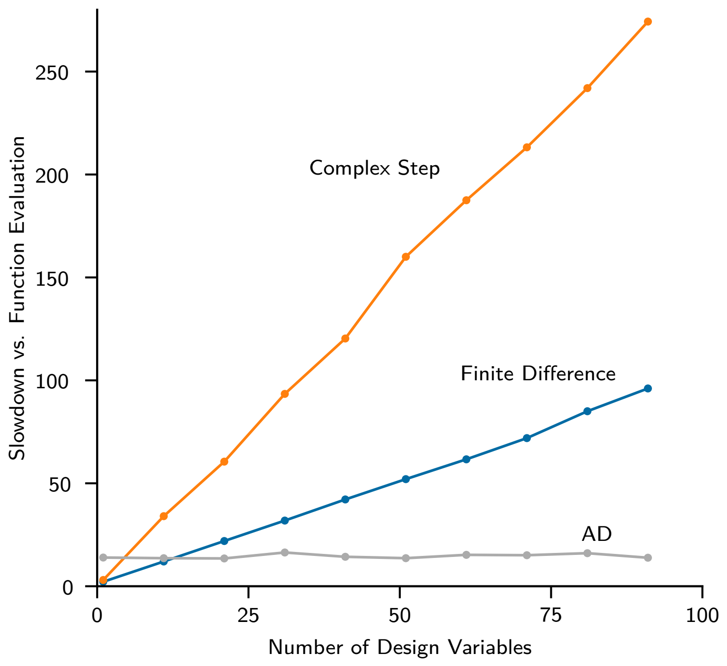


Figure 14.2: Comparison of the computational efficiency of the finited difference, complex step, and AD gradient evaluations as a function of the number of design variables.

of design variables is important for the application to large optimization problems encountered in MDO. To show the computational efficiency of the AD-based method, I benchmarked the computational expense of the gradient evaluation for a 100 element beam model using the finite difference, complex step and AD gradient evaluations (Figure 14.2). While the complex step and finite difference methods scale linearly with the number of design variables, the AD method in reverse mode remains constant, independent of the number of variables. The AD method outperforms the complex step results for any problem size larger than five variables and the finite difference results as of 10 variables. Practical optimization problems feature large number of design variables, therefore the AD-based method evaluates the gradients faster than the competing methods while maintaining high accuracy.

CHAPTER 15

Beam-Based Optimization Studies

Having verified the accuracy of the gradient solution within UM/NAST (Chapter 14), I conducted a series of beam-based optimization studies utilizing the flutter constraint formulated in Chapter 5. Based on the BWB, these studies aim to investigate the effect of a geometrically nonlinear flutter constraint on an aircraft design optimization problem.

A fuel burn minimization was formulated for the BWB using the simplified mass, drag, and fuel burn models. Two aeroelastic constraints were implemented using UM/NAST: a bending curvature (static aeroelastic) constraint and the proposed geometrically nonlinear flutter (dynamic aeroelastic) constraint. The components for the multidisciplinary optimization problem were implemented in OpenMDAO [92, 93, 94] and optimized using the SciPy optimize package [95] with the SLSQP optimizer. To better understand the effect of the aeroelastic constraints, an optimization was conducted with the bending constraint only and another including the proposed geometrically nonlinear flutter along with the strength constraint.

15.1 Cross Section Properties

As the existing BWB model consists of a beam representation, the cross section properties of the baseline configuration cannot easily be translated to a parametric model. Because of this, the wing cross section is treated as a simple, rectangular wing

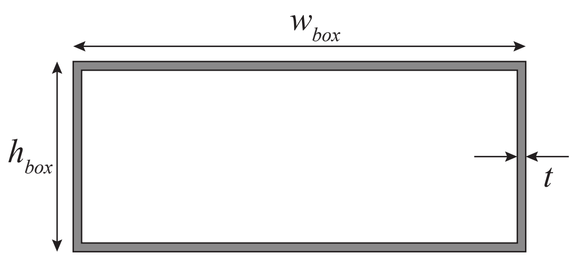


Figure 15.1: Surrogate wing box cross section.

box. The cross section properties of the wing box are determined analytically as a function of the wing box width, height and thickness.

The wing box width is coupled to the planform shape by the relative wing box size η_{box}, such that:

$$w_{box} = \eta_{box}\, c_{wing} \tag{15.1}$$

The total structural mass of the aircraft is obtained from:

$$m_{struc} = b_{wing}\, \mu_{wing} + m_{body} \tag{15.2}$$

15.2 Flight Envelope

An illustrative flight envelope was defined for the BWB in terms of altitude over speed and angle of attack. Three bounds are defined for the BWB: the stall boundary, the service ceiling, and the maximum permissible Mach number (Figure 15.2).

Typically, the service ceiling is defined by the ability of the aircraft to climb at a determined rate. For this work the service ceiling was set without analysis and serves purely to generate a representative flight envelope shape. The stall boundary is found

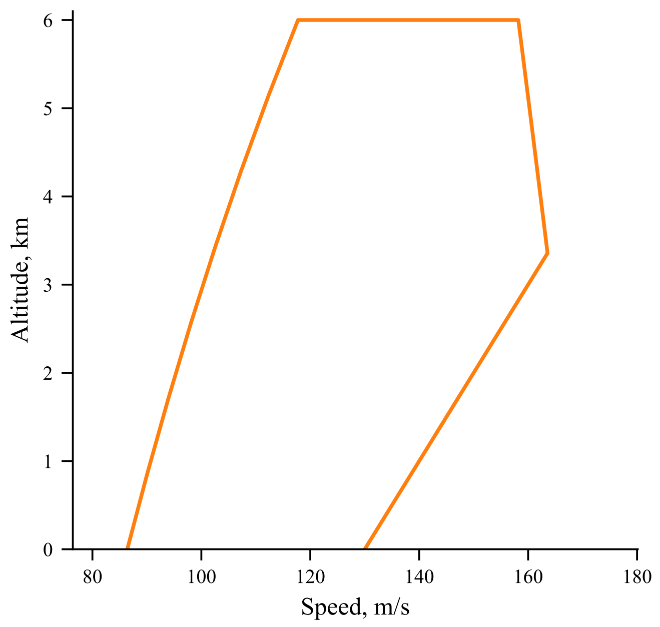


Figure 15.2: Notional flight envelope of the BWB.

from level flying conditions at maximum lift coefficient:

$$v_{stall} = \sqrt{\frac{2mg}{\rho_\infty C_{L,max} S}} \tag{15.3}$$

To determine the maximum Mach number boundary in terms of speed, the Mach number is multiplied by the speed of sound (function of altitude):

$$v_{max} = M_{max} a_\infty (h) \tag{15.4}$$

The flutter search process is conducted using spee and altitude information (in addition to the angle of attack). The flight envelope was linked to the trim angle of attack at cruise. Trim is controlled solely by the root angle of attack, with no control surfaces modeled in this case.

15.3 Drag Prediction

Drag of the entire aircraft is determined from the induced and friction drag components. The induced drag is calculated from the aspect ratio such that [96]:

$$C_{Di} = \frac{1}{0.95\, AR\, \pi} C_L^2 \tag{15.5}$$

where AR is the wing aspect ratio. The friction drag is accounted for by [96]:

$$C_{Df} = C_f \frac{S_{wet}}{S} \tag{15.6}$$

where:

$$C_f = \frac{0.074}{Re^{0.2}} \tag{15.7}$$

The resulting total drag coefficient is the sum of the friction and induced drag coefficients:

$$C_D = C_{Di} + C_{Df} \tag{15.8}$$

15.4 Fuel Burn Prediction

The fuel burn is determined from the aircraft range. The Breguet range equation[1] can be written as [98, 99, 100]:

$$R = \frac{v_c \, C_L}{C_D \, sfc} \ln\left(\frac{W_1}{W_2}\right) \tag{15.9}$$

$$= \frac{M_c \, a_c \, C_L}{C_D \, sfc} \ln\left(\frac{W_1}{W_2}\right) \tag{15.10}$$

The initial weight, W_1, is the combined weight of the structure and fuel, while the weight at the end of the trip, W_2, is just the structural weight (no fuel reserves considered), i.e.,

$$W_1 = W_s + W_f \tag{15.11}$$

$$W_2 = W_s \tag{15.12}$$

The fuel weight and fuel mass are thereby determined by rearranging the range equation:

$$W_f = W_s \left(e^{R \frac{C_D \, sfc}{C_L \, v_c}} - 1\right) \tag{15.13}$$

$$m_f = \frac{W_f}{g} \tag{15.14}$$

It should be noted, that the drag coefficient is taken from the drag prediction and the structural mass from the parametric mass model.

[1]While the equation derives its name from Louis Breguet, as Cavcar notes [97], the origin of the equation can be traced to multiple sources.

15.5 Optimization Including Static Constraint

First, a fuel burn minimization was formulated including a bending constraint mimicking a strength constraint (Eq. 15.15). This is achieved by aggregating the largest bending curvature (from UM/NAST) along the wing using KS constraint aggregation. The constraint is evaluated across all samples of the flight envelope, such that a second KS aggregation (similar to the proposed flutter constraint) is needed to obtain a scalar bending constraint. A span limitation was imposed to mimic a ground handling or gate constraint and a minimum chord constraint was imposed to avoid excessively slender wing configurations. A fuel volume constraint was included to ensure that the predicted fuel required for the mission can be stored within the wing box. Finally, a trim constraint is imposed. The design variables for this problem are the wing aspect ratio AR, wing surface area, body root angle of attack α_{root}, the size of the wing box relative to the wing chord η_{box}, the location of the wing box center x_{box}, and the thickness of the body and wing box skins t_{body} and t_{wing}.

$$\begin{aligned}
&\text{minimize:} && m_f \\
&\text{with respect to:} && \mathrm{x} = \left[AR, S, \alpha_{root}, \eta_{box}, x_{box}, t_{body}, t_{wing}\right]^T \\
&\text{subject to:} && \frac{V_{fuel}}{V_{box}} \leq 1 \\
& && \eta_{box} \leq 0.9 \\
& && x_{front\,spar} \geq 0 \\
& && x_{front\,spar} \leq 1 \\
& && x_{rear\,spar} \geq 0 \\
& && x_{rear\,spar} \leq 1 \\
& && KS\left(KS\left(\kappa_y\right)\right) \leq \kappa_{max} \\
& && L = W
\end{aligned} \tag{15.15}$$

Figures 15.3 and 15.5 and Table 15.1 show the results of this optimization problem. For this optimization the strength and fuel volume constraints are active. Figure 15.3 shows the iteration history of optimization problem.

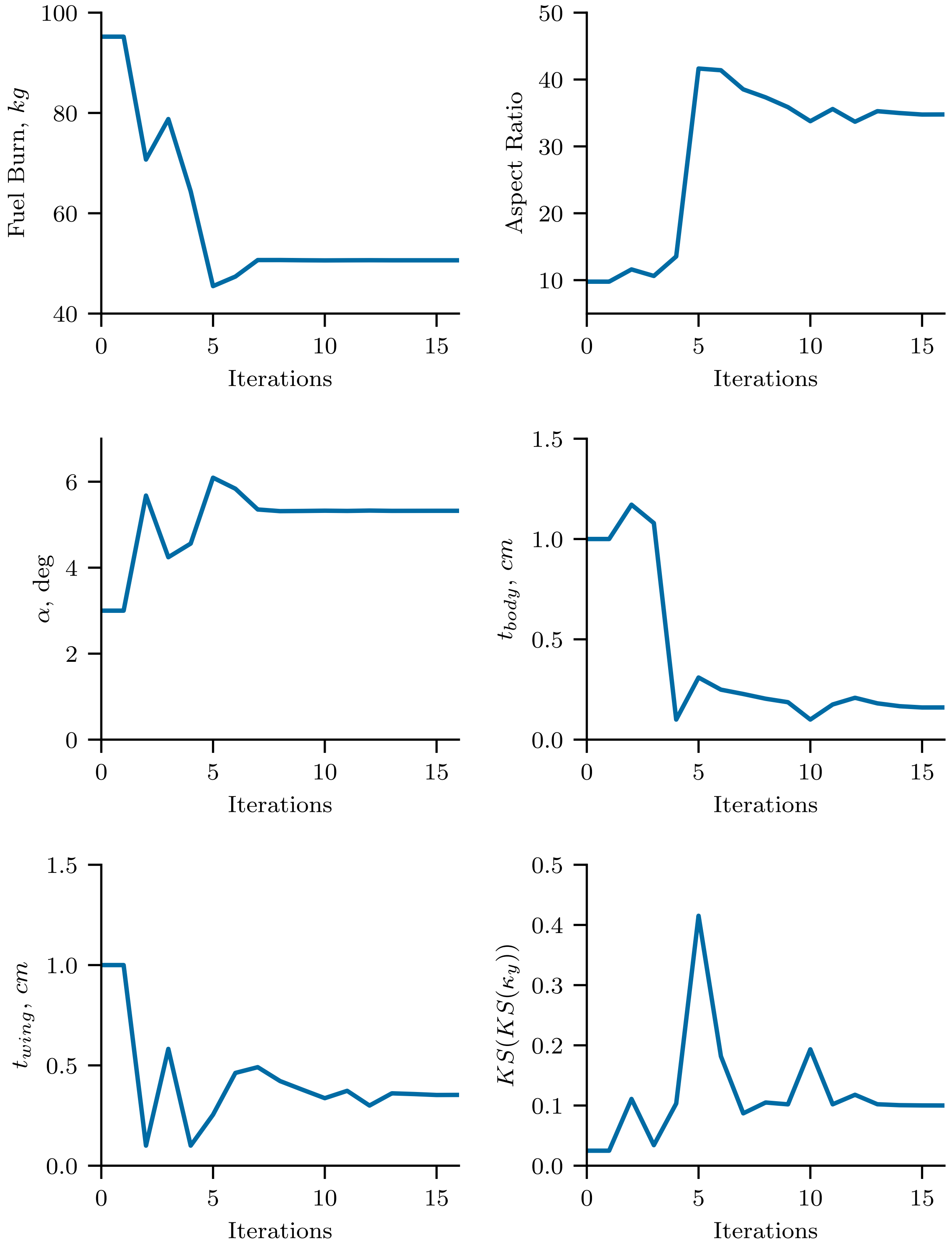


Figure 15.3: Iteration history of the optimization including the strength constraint.

15.6 Optimization Including Flutter Constraints

Next, a flutter constraint was added to the optimization problem (Eq. 15.16) yielding:

$$
\begin{aligned}
\text{minimize:} \quad & m_f \\
\text{with respect to:} \quad & \mathrm{x} = \left[AR, S, \alpha_{root}, \eta_{box}, x_{box}, t_{body}, t_{wing}\right]^T \\
\text{subject to:} \quad & \frac{V_{fuel}}{V_{box}} \leq 1 \\
& \eta_{box} \leq 0.9 \\
& x_{front\,spar} \geq 0 \\
& x_{front\,spar} \leq 1 \\
& x_{rear\,spar} \geq 0 \\
& x_{rear\,spar} \leq 1 \\
& KS\left(KS\left(\kappa_x\right)\right) \leq \kappa_{max} \\
& KS\left(KS\left(\mathrm{Re}\left(\zeta\right)\right)\right) < -0.2 \\
& L = W
\end{aligned}
\qquad (15.16)
$$

The results to this optimization problem are presented in Table 15.1 and in Figures 15.4 and 15.5.

In this optimization the flutter constraint is active, while the strength constraint is inactive. While both configurations look similar at first glance, the inclusion of the flutter constraint has a profound impact on the optimal configuration. Noteably, while the addition of the geometrically nonlinear flutter constraint result in increased fuel burn, it is only a modest increase of 0.8% . However, despite the similar values in fuel burn, the aspect ratio is reduced by 13%, while the thickness distribution of the wing and body skins is substantially different between the two configurations. The

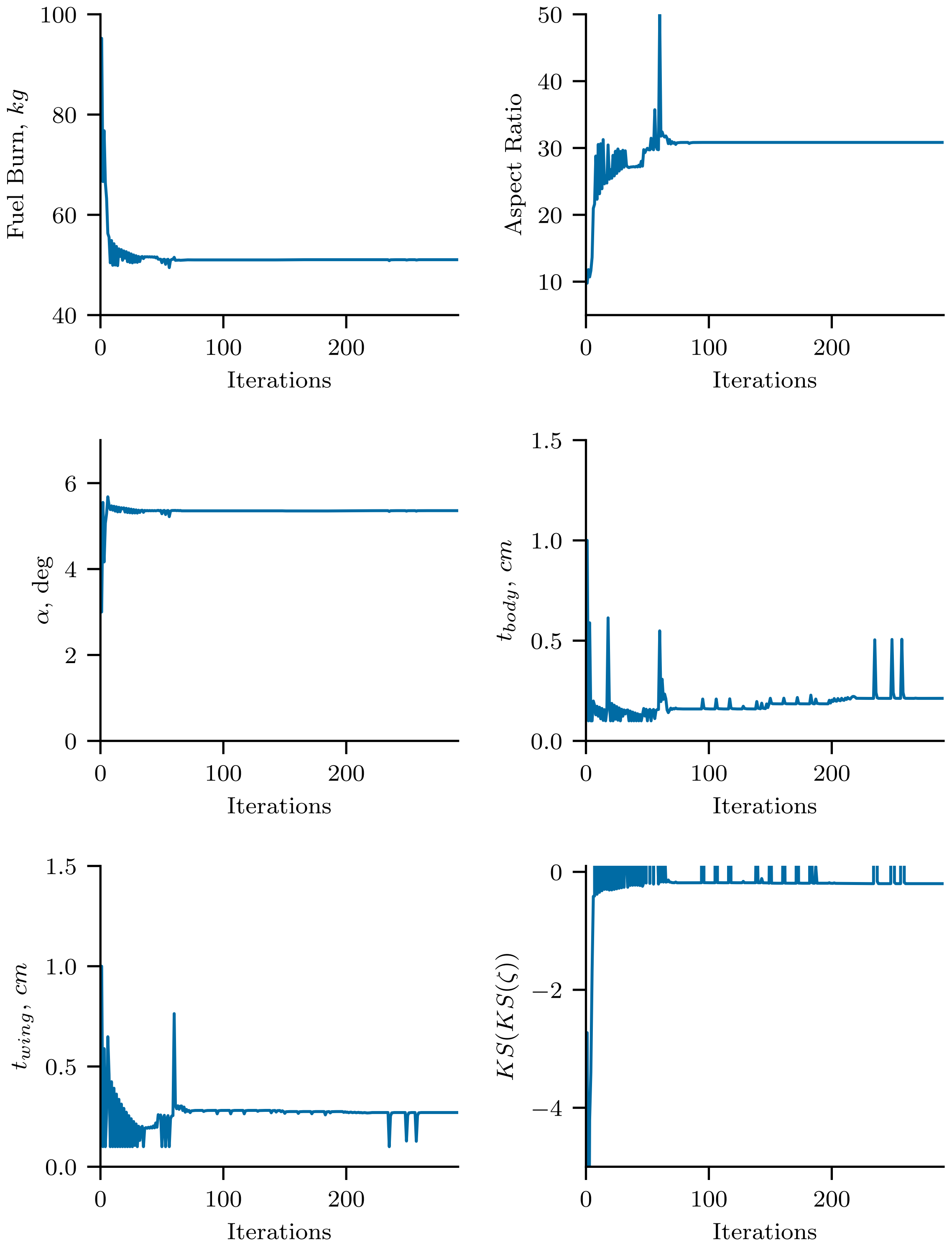


Figure 15.4: Iteration history of the optimization including the strength constraint.

Table 15.1: Optimal configurations for the BWB under strength and/or geometrically nonlinear flutter constraints.

Description	**Strength**	**Flutter**
m_f (kg)	50.62	51.04
Aspect Ratio	34.77	30.83
Wing Area (m^2)	3.058	3.015
α_{root} (deg)	5.32	5.35
η_{box}	0.899	0.9
x_{box}	0.549	0.549
t_{body} (cm)	0.16	0.21
t_{wing} (cm)	0.35	0.27

configuration without the flutter constraint has a larger wing skin than the body skin thickness. The inclusion of the flutter constraint, meanwhile, reduces the wing skin and increases the body skin thickness.

The high aspect ratio designs obtained here may not be attainable for other aircraft design problems. These high aspect ratio designs are likely a result of a low cruise speed ($M = 0.4$) and the minimization of fuel burn, which will favor higher aspect ratios, as well as the absence of a buckling constraint. However, this example highlights that including a geometrically nonlinear flutter constraint may yield a different design, while its exclusion may result in an infeasible configuration. Furthermore, the difference in the design may stem from a change in stiffness properties, while the overall planform may be similar. Finally, it is noteworthy, that while the configuration subject to only the strength constraint has a better fuel burn, the configuration including the flutter constraint results in a lower structural mass.

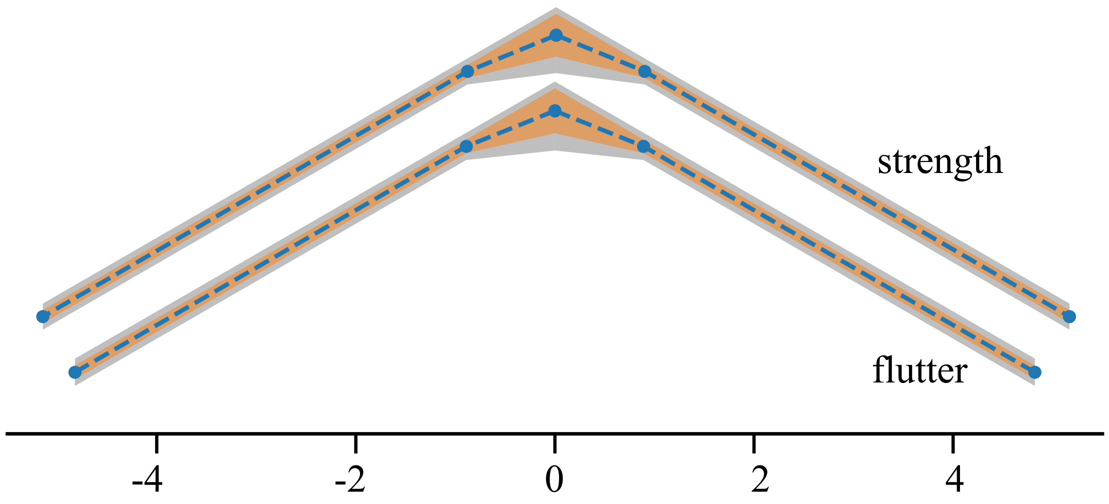


Figure 15.5: Planform comparison of baseline BWB including the strength and flutter constraints. The orange shading indicates the wing box geometry, while the blue dashed line represents the beam reference line.

CHAPTER 16

Multi-Fidelity Studies

Chapter 15 illustrates the need to include a geometrically-nonlinear flutter constraint by using a beam-based optimization problem. In general, however, aircraft MDO problems use higher-fidelity analyses which are incompatible with beam-based methods without intermediate steps. These intermediate steps, the mass and stiffness condensation processes and their respective gradients, were presented in Chapter 7. These condensation processes permit the use of a shell-based FEM model and CFD (although VLM will be used in this work) for the objective function and strength constraints, while utilizing the beam-based solution for the geometrically-nonlinear flutter constraint.

In this manner, the analyses are chosen based on their computational expense and their ability to resolve a given quantity of interest. While beam models cannot resolve local stresses, a higher-fidelity analysis (such as large shell models) may not be able to account for geometrical nonlinearities fast enough for optimizations and yield no additional accuracy. As a result, a multi-fidelity problem constitutes a “best of both worlds” approach. This chapter presents an application of a multi-fidelity problem that evaluates the objective function using higher-fidelity analyses and determines the feasibility of the flutter constraint using a geometrically nonlinear beam model.

16.1 Mass Condensation Verification

The mass condensation component is verified using a simple plate configuration (Figure 16.1). The plate properties are listed in Table 16.1. The component results are compared to analytical values for mass, inertia, and center of gravity position (Table 16.1). The values obtained from the mass condensation match the analytical values to machine precision.

Next, the accuracy of the gradient values obtained by the mass condensation component are quantified. To this end, a single mass element of the plate is perturbed using an imaginary disturbance ih. The reference gradient is then determined using the complex step method (Chapter 2):

$$g(x) \approx \frac{Im\left(f\left[x+ih\right)\right]}{h} \tag{16.1}$$

The derivative obtained using the complex step method is accurate to machine precision. The gradient results from the mass component and the corresponding reference results are listed in Table 16.2. The gradient obtained from the mass condensation component matches the complex step results to machine precision. As a result, the mass property gradients are accurate to machine precision.

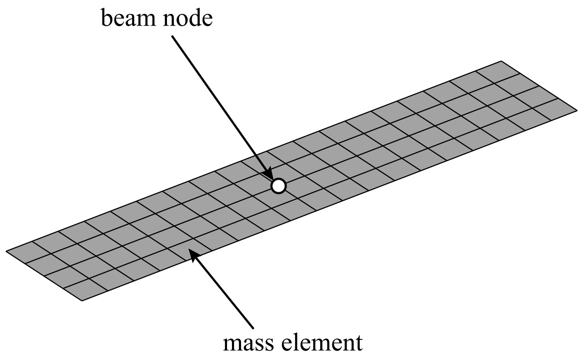


Figure 16.1: Plate example for testing the mass condensation process and verification of mass condensation gradients.

Table 16.1: Plate properties of the verification test case as well as component and analytical reference results for the mass condensation.

	Value	Reference Value
Plate Length, m	1.0	–
Plate Width, m	0.2	–
Plate Thickness, m	0.01	–
Plate Density, kg/m^3	2700.0	–
Verification Results		
Mass, kg	5.39999999999999	5.4
x_{cg}, m	0.10000000000000006	0.1
y_{cg}, m	0.5000000000000003	0.5
z_{cg}, m	0.0	0.0

Table 16.2: Comparison of the mass condensation component gradients with respect to mass element thickness with reference results using the complex step method.

	Component	Complex Step
$\frac{\partial m}{\partial t}$	7.105263157894723	7.105263157894723
$\frac{\partial I_{xx}}{\partial t}$	3.07533897069543	3.07533897069543
$\frac{\partial I_{yy}}{\partial t}$	0.039967105263157825	0.039967105263157825
$\frac{\partial I_{zz}}{\partial t}$	3.1153060759585878	3.1153060759585878
$\frac{\partial x_{cg}}{\partial t}$	-0.03289473684210529	-0.03289473684210529
$\frac{\partial y_{cg}}{\partial t}$	0.20775623268697985	0.20775623268697985
$\frac{\partial z_{cg}}{\partial t}$	0.0	0.0

16.2 Stiffness Condensation Verification

Next, the stiffness condensation process was verified. To determine the accuracy of the function evaluation, the equivalent beam stiffness properties of a 30-element straight beam with constant properties are determined using FEMtoBeam and compared against reference data [77]. The reference solution for the first element is:

$$k_1^{NAST} = \begin{bmatrix} 1.2615 \times 10^9 & 1.8566 \times 10^{-5} & -0.0866 & 3.1114 \\ 1.8566 \times 10^{-5} & 9.6953 \times 10^6 & -0.4323 & 0.0194 \\ -0.0866 & -0.4323 & 7.5533 \times 10^6 & 1.6162 \times 10^{-3} \\ 3.1114 & 0.0194 & 1.6162 \times 10^{-3} & 121.7089 \times 10^6 \end{bmatrix} \tag{16.2}$$

Over all elements and stiffness matrix components, the largest relative error compared against the reference data is less than 0.1%. It is worth noting, that the reference data for this comparison originates from an equivalent beam condensation process which uses the same theoretical formulation [45] as in this work, leading to a close match.

The accuracy of the stiffness condensation gradients was evaluated by comparing against complex step result. To this end, the 30-element beam was perturbed by an imaginary step for every element and every degree of freedom (Figure 16.2). The largest deviation between the predicted gradient and the complex step reference is smaller than 10^{-10}. While this is larger than the floating point working precision, it nonetheless constitutes a high-accuracy result for the gradients suitable for use in optimization problems.

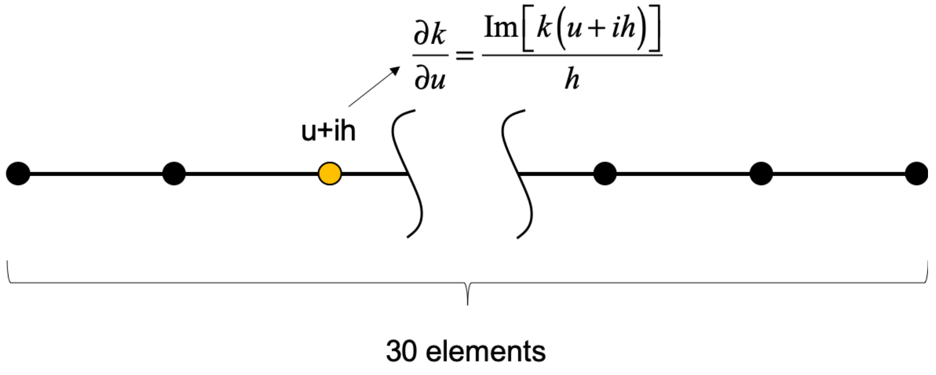

$\frac{\partial k}{\partial u} = \frac{\mathrm{Im}[k(u+ih)]}{h}$
u+ih
30 elements

Figure 16.2: Plate example for testing the mass condensation process and verification of mass condensation gradients.

16.3 uCRM Studies

I apply the multi-fidelity problem formulation to a high aspect ratio transport aircraft model (see Chapter 12) representative of potential future aircraft configurations. The multi-fidelity problem illustrates a roadmap for including a beam-based, geometrically nonlinear flutter constraint into a higher-fidelity optimization problem.

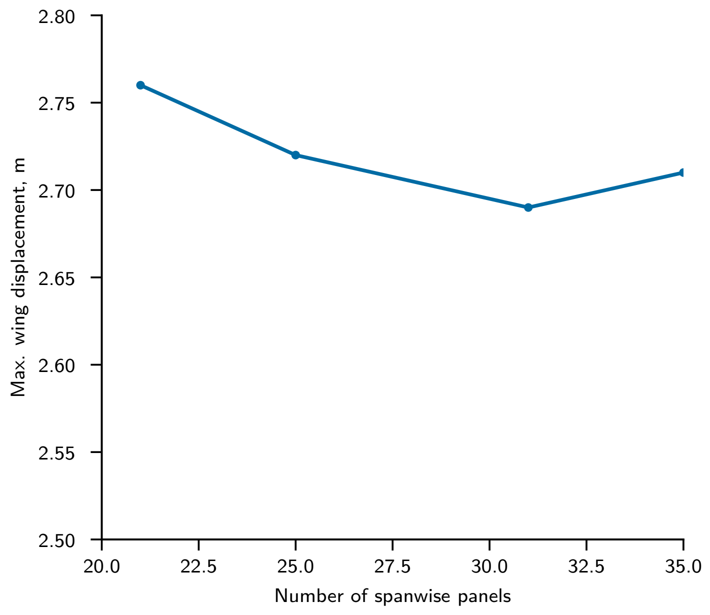


Figure 16.3: FSI convergence study for the uCRM 13.5 wing.

16.3.1 Aerostructural Convergence Studies

Before conducting the aerostructural optimization, I conducted a convergence study of the TACS/VLM FSI solution. The purpose of this study is to ensure that the mesh refinement of the VLM solver is sufficiently fine to yield a converged result.

For the convergence study I refined the spanwise number of VLM panels and observed the maximum tip displacement of the FSI solution. Figure 16.3 shows the convergence study of the uCRM 13.5 configuration. For the remaining studies, 31 spanwise panels are used as the difference between the maximum displacements is lower than 1%.

16.3.2 Aerostructural Optimization

Next, I conducted an aerostructural optimization of the uCRM 13.5 wing to serve as the baseline of the multi-fidelity studies. The optimization is formulated as a fuel

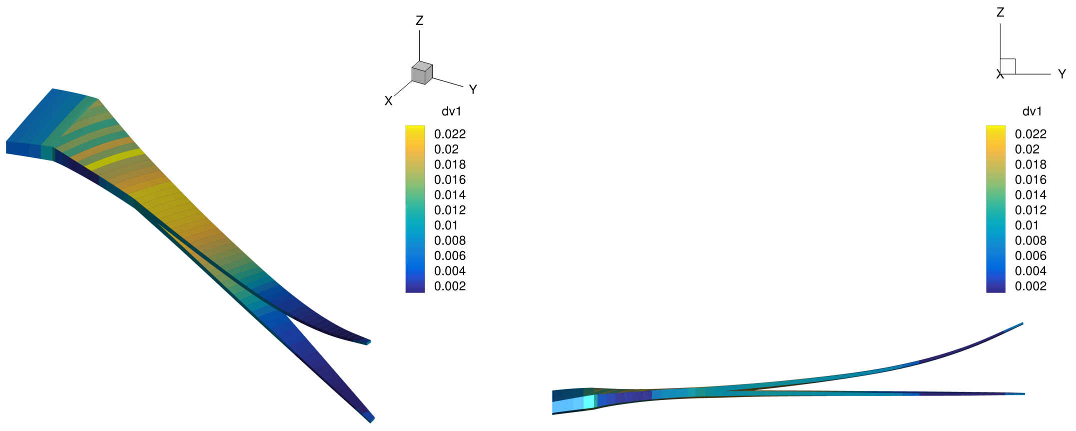


Figure 16.4: Thickness distribution and displacements of the uCRM 13.5 wing optimized with a von Mises stress constraint.

burn minimization problem subject to a von Mises yield constraint:

$$\begin{aligned} \text{minimize:} \quad & m_f \\ \text{with respect to:} \quad & \mathrm{x} = t_i \\ \text{subject to:} \quad & KS\left(\sigma_{Mises}\right) \leq \sigma_{yield} \end{aligned} \tag{16.3}$$

The design variables for this problem are the skin thicknesses of every wing rib and rib bay patch. Figure 16.5 shows the XDSM diagram of the FSI optimization problem. More than serving as the baseline solution, this optimization problem can be extended later by the flutter constraint.

The SLSQP optimizer from the SciPy optimize toolbox [95] was used and a uniform starting thickness of $t_i = 0.02\,\mathrm{m}$ was chosen. Figure 16.4 shows the optimized result of the uCRM 13.5 wingbox subject to a von Mises stress constraint (and no flutter constraint).

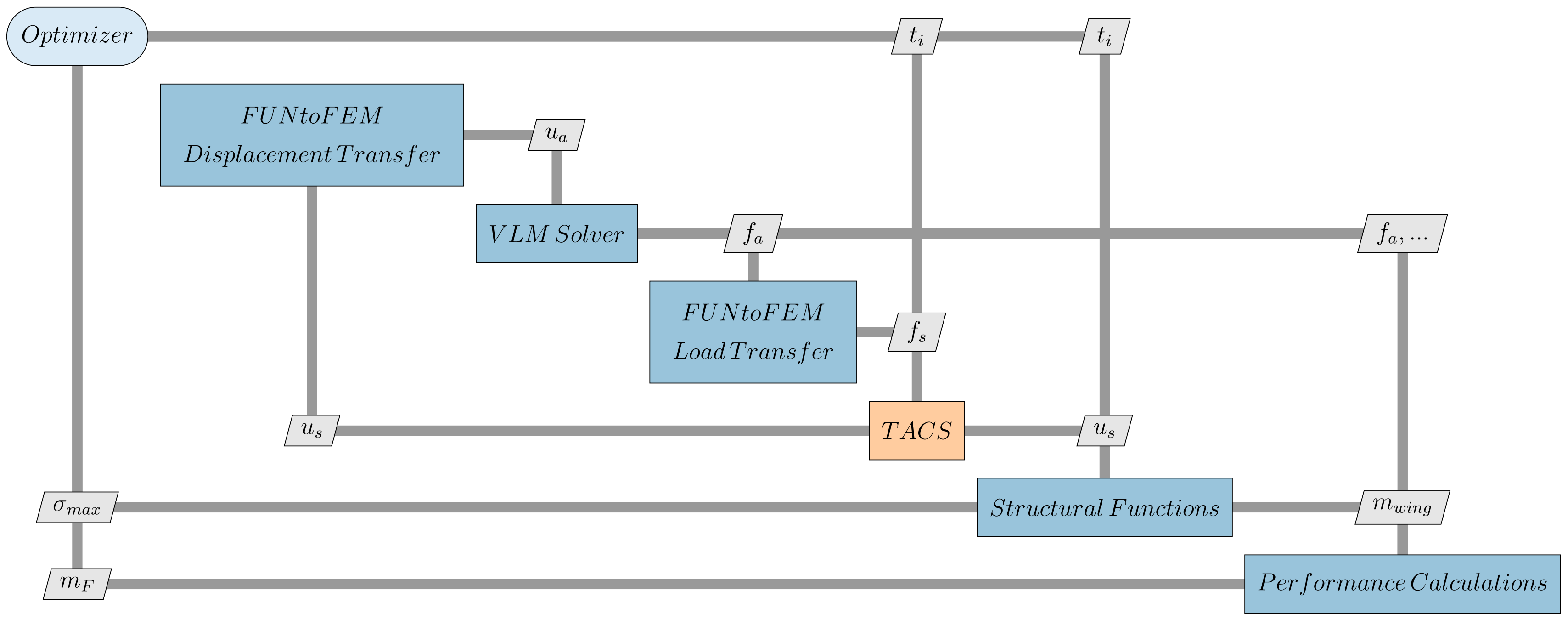


Figure 16.5: XDSM diagram of the FSI problem.

16.3.3 Multi-Fidelity Problem

Next, the multi-fidelity problem is assembled (16.6). The higher-fidelity FSI problem is retained while the flutter constraint is added. As described in Chapter 7, equivalent beam properties are obtained from the higher-fidelity structural mesh and transferred to UM/NAST for the flutter analyses. While UM/NAST is capable for accounting for planform changes as well, in the absence of an aerodynamic condensation process, this work focuses on the structural design variables. The flutter constraint is then evaluated using the KS-aggregated approach presented in Chapter 5. As such, a multi-fidelity optimization including the flutter constraint is represented as:

$$\begin{aligned}
&\text{minimize:} && m_f \\
&\text{with respect to:} && \text{x} = t_i \\
&\text{subject to:} && KS\left(\sigma_{Mises}\right) \leq \sigma_{yield} \\
& && KS\left(KS\left(\zeta_i\right)\right) \leq 0
\end{aligned} \tag{16.4}$$

In this work, the multi-fidelity problem is not evaluated as an optimization problem due to robustness issues in the VLM-FEM FSI solution resulting in failures to complete or converge the optimization. To illustrate the applicability of the flutter constraint within the multi-fidelity problem, the process in Figure 16.6 is run with the optimized uCRM 13.5 configuration. The panel thicknesses of the optimized configuration are introduced into the beam condensation to obtain the equivalent beam stiffness distribution (Figure 16.7).

The flutter constraint values before the second KS aggregation are depicted in Figure 16.8. The configuration optimized with the von Mises stress constraint (Section 16.3.2) is infeasible with respect to the flutter constraint. Moreover, a large percentage

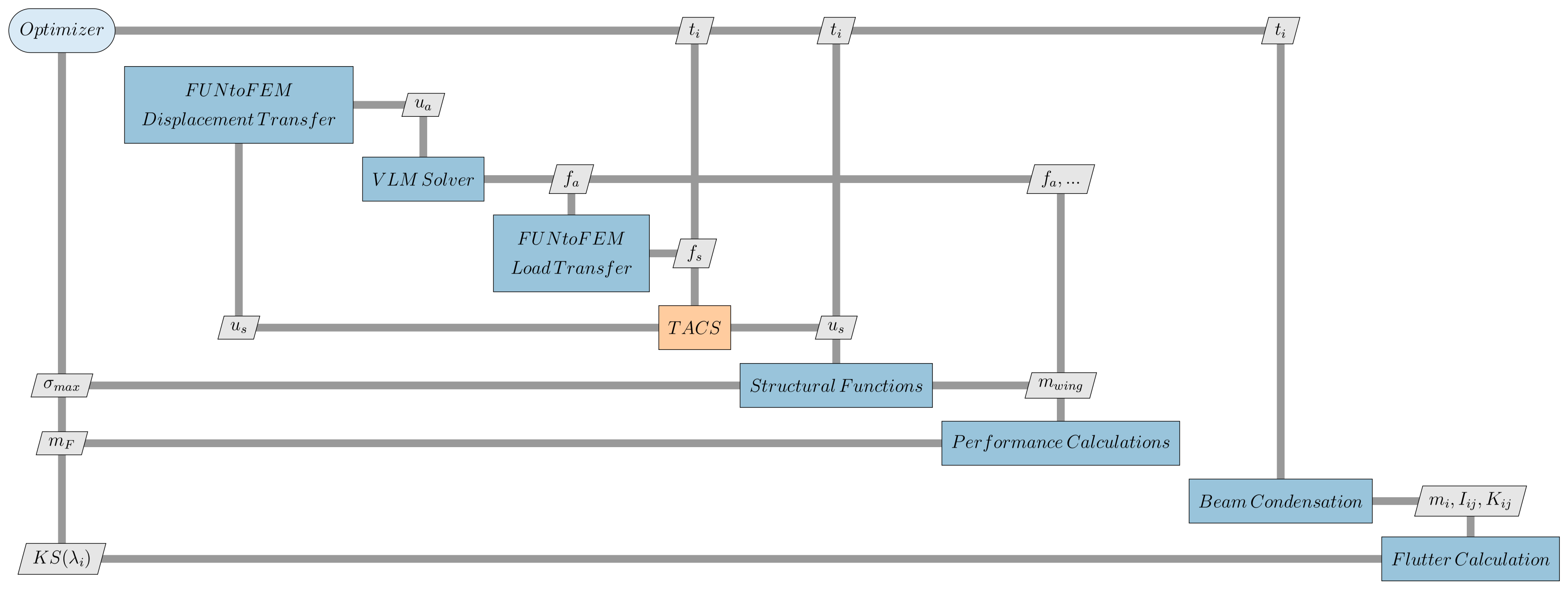


Figure 16.6: XDSM diagram of the multi-fidelity optimization problem.

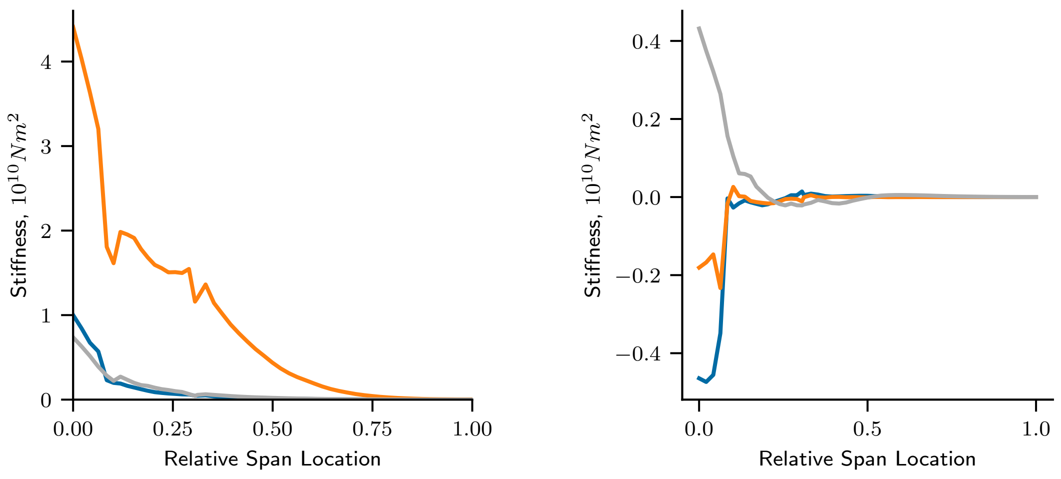


Figure 16.7: Equivalent beam stiffness properties for the optimized uCRM 13.5 configuration obtained from the stiffness condensation process. The diagonal terms (torsion [blue], in-plane bending [orange], and out-of-plane bending [gray]) are shown on the left. On the right, the off-diagonal stiffness terms are shown (torsion-out-of-plane bending [blue], torsion-in-plane bending [gray], and out-of-plane-in-plane bending [orange])

of the flight envelope encounters flutter, reinforcing the necessity of including the geometrically nonlinear flutter constraint for flexible vehicles such as the uCRM 13.5.

16.4 Future Work

While the optimized configuration in Section 16.3.2 has been shown to encounter flutter within the flight envelope, the proposed multi-fidelity problem offers a potential avenue to address flutter including geometrically nonlinear effects in large-scale MDO problems. The tools developed within this work enable both the function and gradient evaluation of the beam condensation process, and thereby are suitable for gradient-based optimization. However, the higher-fidelity methods (VLM-FEM solution) proved too unstable for the studies conducted within this thesis. As such, future work is required to obtain a more robust FSI solution.

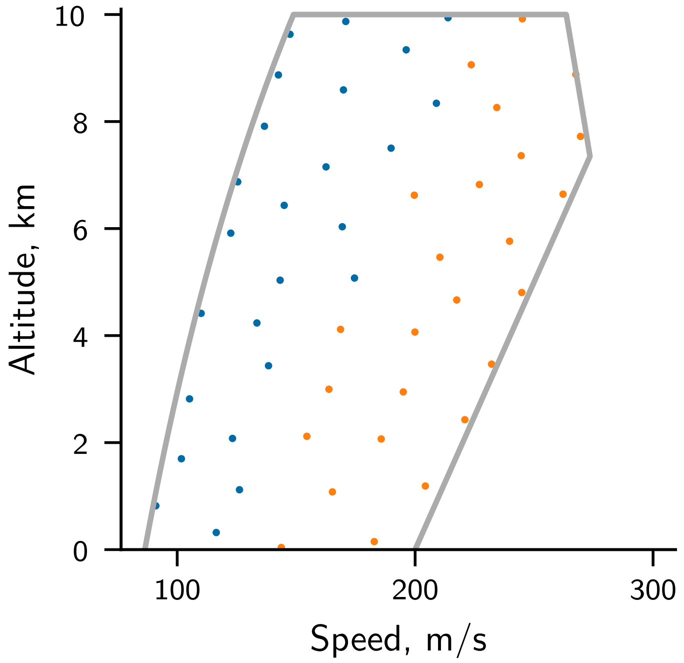


Figure 16.8: uCRM 13.5 flight envelope showing the KS aggregated damping values. The orange points denote unstable, while the blue show stable search points.

Part IV

Conclusions

CHAPTER 17

Concluding Remarks and Contributions

This thesis endeavored to include geometrically nonlinear effects into aircraft design optimization problems. To achieve this, contributions were accomplished in the UM/NAST framework, the efficient determination of nonlinear aeroelastic sensitivities, the interpretation and solution of nonlinear flutter problems, the application of a geometrically nonlinear flutter constraint, as well as including the beam-based constraint into a higher-fidelity MDO problem. The contributions presented within this work are:

1. Extended the interpretation for flutter problems including geometrical nonlinearities.
2. Developed high-accuracy methods for numerically linearizing the nonlinear equations of motion within the UM/NAST framework.
3. Created a computationally efficient algorithm for accurately predicting flutter including geometrical nonlinearities.
4. Extended a flutter constraint and applied it to geometrically nonlinear problems.
5. Extended UM/NAST into a design and optimization tool for geometrically nonlinear problems.

6. Developed an efficient method for recovering nonlinear aeroelastic sensitivities using AD.

7. Showed that geometrical nonlinear flutter constraints may drive the optimal configuration's design.

8. Proposed a roadmap for including beam-based, nonlinear analyses into higher-fidelity MDO problems.

The foundation of this work was created in the UM/NAST framework. Previously, this toolbox was designed for nonlinear aeroelastic analyses and simulations and was ill-suited for design problems. In this work, I conducted a large-scale rewrite of the UM/NAST framework, which transformed it from an analysis tool to one suitable for design problems. As a result, UM/NAST solutions are more computationally efficient than even the previous C++ implementations (version 2–3). This is true for serial applications, but the addition of parallelism to batch jobs enables even larger performance gains (dependent on the computer available). To facilitate gradient-based optimization problems, I added the ability to determine gradients within the framework. Finally, I created a modular system of enhancing UM/NAST capabilities including interfaces for external aerodynamic solvers, feedback controllers, simulated sensors, customized atmospheric models, etc. While not used for the research in this work, these interfaces find wide-spread use at A^2SRL and enable a wide range of research initiatives.

Furthermore, in applying AD to the UM/NAST code, I enabled the determination of sensitivities within the UM/NAST framework. These changes were conducted to facilitate gradient-based optimization problems. Because these require fast gradient evaluations, I investigated methods for improving the computational efficiency of AD gradient evaluations within UM/NAST. The hybrid-AD solution I proposed enables the computationally efficient determination of sensitivities without sacrificing accuracy.

Next, I investigated the interpretation of nonlinear flutter problems. Previous work relied on linear flutter interpretation methodologies to analyze and interpret nonlinear flutter problems. In this work, I presented problems by applying the linear flutter problem methodologies to nonlinear problems and proposed alternate methods of interpreting the nonlinear solutions. Based on this interpretative approach I unified the interpretation of linear and nonlinear flutter analyses and discussed the limitations of linear flutter interpretations for nonlinear problems, the difference in requirements for the flutter search process, and the applicability of the V-g diagram to nonlinear problems. I proposed extended V-g diagrams as an interpretative tool for nonlinear problems.

Based on the work on flutter interpretations, I formulated an enhanced flutter algorithm and implemented it within UM/NAST. The enhanced flutter algorithm enables parallelism during the flutter search process and uses a mode-tracked, surrogate-based approach to determining the flutter point. Using the AD improvements to the UM/NAST framework, I formulated new, higher accuracy linearization methods. Beyond its accuracy, the AD linearization shows significant speed improvements compared to the past method (based on the forward difference method). I showed that the proposed algorithm results in substantial accuracy improvements compared to the legacy flutter algorithm.

Furthermore, I formulated a flutter constraint, based on an existing methodology, to include geometrically nonlinear flutter analyses into aircraft design problems. The approach utilizes a sequential KS aggregation to obtain a conservative estimate of the most critical damping value. However, the nonlinear nature of the flutter problem results in different requirement to the flutter constraint. As a result, the aircraft flight envelope must be sampled. As such, I modified and applied an existing approach to constrained sampling to (non-hypercube) aircraft flight envelopes. The resulting flutter constraint therefore enforces feasibility for the entire flight envelope.

Next, I applied the geometrically nonlinear flutter constraint to a sample (beam-based) optimization problem. To investigate the effect of the flutter constraint, I applied a strength and a flutter constraint with geometrical nonlinearities in different optimization studies. The resulting configurations showed the most conservative wing planform to result from the nonlinear flutter constraint, while the application of only the strength constraint resulted in the least conservative configuration. This shows that the geometrically nonlinear flutter constraint may become necessary during optimization problems as aircraft become more flexible.

Finally, I developed and verified beam condensation tools based on existing methodologies to integrate the beam-based flutter constraint into a higher-fidelity MDO problem. Furthermore, I coupled an existing VLM tool with the shell-based FEM solver TACS to obtain a "high-fidelity" aerostructural solution. I integrated the beam condensation tools to present a roadmap for including the flutter constraint into a higher-fidelity optimization problem. As such, while the final result of this thesis is not a full-fledged MDO problem, it does offer an argument for the inclusion of geometrically nonlinear effects as well as a roadmap of how to include them as future aircraft become more flexible.

CHAPTER 18

Potential Future Work

This thesis developed methods and provides recommendations for working with geometrically nonlinear effects within aircraft MDO problems. A number of aspects remain to be investigated and pose possible avenues of future research. This chapter discusses some of the possible areas of investigation, focusing on potential developments to the UM/NAST framework, investigations regarding the geometrically nonlinear flutter constraint, and the multi-fidelity problem including the flutter constraint.

18.1 UM/NAST

Much of the work presented in this thesis centers around the UM/NAST framework. Methods for obtaining gradients were introduced and verified. This section discusses possible future developments within UM/NAST. The framework has constituted one of the backbones at A^2SRL and therefore fulfills many different needs, ranging from feedback controller design to optimization problems. Because the total required or possible future developments are vast, I will focus solely on the topics related to this work.

While I mounted a concerted effort to improve the performance of the gradient methods, a number of changes to the application of AD can yield further improvements. Currently, AD is applied across all subroutines via templating. While this allows for implementational flexibility, the current method will result in a large number of entries to the AD tape. CoDiPack allows the developer to locally define functions and their respective jacobians to reduce this overhead. While I was aware of this during the development of this work, the implementation proved too time consuming to be reasonably completed for this thesis. This means that every linear algebra function call is differentiated down to every local function variable, which may result in a substantial increase in tape size. To alleviate this, future work could apply the reverse mode derivatives for common linear algebra routines derived by Giles [101] together with CoDiPack's external function helpers. The resulting reduced AD tape size would benefit both the time needed for gradient evaluations as well as the memory footprint, both of which come at a premium during MDO problems.

Another area of investigation may be the inclusion of control laws during the linearization and stability analyses. UM/NAST currently permits the inclusion of rigid body degrees of freedom as well as unsteady aerodynamics using Peters' finite state aerodynamics, however, it does not allow for a linearization including a feedback controller. While open loop linearization and stability studies certainly yield interesting and useful results, most modern and future aircraft feature feedback controllers and their contribution to or inhibition of flutter is valuable to the certification process and should be studied. Such studies would require a theoretical development of how to linearize the feedback controller within UM/NAST along with the obvious implementational aspects.

18.2 Flutter Constraint

The application of the geometrically nonlinear flutter constraint also offers possibilities for further studies. As I discussed in Chapter 5, the serial aggregation using KS functions is equivalent to a single KS aggregation over all constraints. The double KS approach used in this work primarily originates from organizational needs. However, a constant KS parameter is not necessary and using two KS aggregations with varying weigths should be investigated further. Furthermore, an adaptive KS approach could be considered in further work if larger number of search points are required.

Further studies could also be conducted into the effect of rigid body DOF as well as the inclusion of feedback controllers on the optimization problem and optimized configuration. As noted by Su and Cesnik [4] and referenced in Chapter 1, the inclusion of rigid body DOF in flutter analyses can dramatically alter the predicted flutter onset and mode. Including these effects could therefore yield a different optimized configuration. Similarly, the inclusion of a feedback controller during the linearization process (as discussed in the previous section) should be studied along with its effect on the optimum. Due to the coupled nature of the flutter problem, the controller may inhibit or exacerbate flutter onset. Including the controller into the constraint should therefore be studied for both a predefined feedback controller as well as setting controller parameters as design variables.

The flutter studies presented in this work utilized a very simple aerodynamic model based on strip theory with unsteady effects accounted by using Peters' finite state aerodynamics. This aerodynamic model sufficed for studying the required methodology. However, most transport aircraft travel at transonic speeds, at which these aerodynamic models are invalid. CFD may fulfill the requirements in terms of physical effects modeled, but would be far too computationally expensive to include into a flutter constraint. A reduced order model approach to unsteady analyses has been investigated by Wang, Fidkowski, and Cesnik [102]. The promise of their approach is to quickly model unsteady transonic flow problems using an artificial neural network. However, questions remain about how this approach can be utilized for a large scale optimization problem. At present, further investigation is necessary whether such a reduced order or other aerodynamic model should be used to account for unsteady transonic flow when applying the flutter constraint.

18.3 Multi-Fidelity Problem

The multi-fidelity problem including the geometrically nonlinear flutter constraint offers many avenues for further exploration. The studies presented in this work focused on method development. Further work is needed to robustly include the beam-based flutter constraint into a higher-fidelity optimization problem. Future studies should also increase the scale of the problem and include additional constraints, such as buckling. For example, this work did not include geometrical design variables. Additional design variables that control the wing planform and twist are a natural next development step. The inclusion of a true trim constraint is another. Other nonlinear aeroelastic constraints, such as limit cycle oscillations or control effectiveness are natural future avenues of investigation.

However, further development is needed on a key component that ties together the varying levels of fidelity: the geometrical manipulation. A tool for manipulating the vehicle geometry was not needed in the presented studies because the design variables were limited to skin thicknesses. For further studies a tool for modifying the structural and aerodynamic meshes would be required. More than manipulate only one geometry, however, the tool would need to be able to account for the geometrical coupling of the varying fidelities. For example, a change in the aerodynamic shape alters not only the shell-based structural model, but also the reference beam location for the beam-based flutter model. A particularly interesting tool in this context is the Engineering Sketch Pad (ESP). ESP can model both the higher-fidelity components and lower-fidelity representations (such as VLM or beam meshes) by an intent driven design system. further work is required to couple ESP with mphys and UM/NAST. The resulting design system, however, would be intriguing as it may permit the use of higher level CAD-based design variables and constraints and bridge the gap between a numerical optimization model and later production CAD models.

18.4 Aircraft Design Studies

In addition to methodological studies that naturally grow out of this work, so too do further aircraft design optimization studies. The uCRM models used in this work constitute current and next generation aircraft configurations. Further studies could be conducted into the performance of composite and tow-steered wing structures compared to the baseline Aluminum configurations. Additionally, nonconventional configurations may benefit from the multifidelity approach. Aircraft such as joined (Prandt) wing configurations [103] encounter nonlinear effects. The proposed multifidelity approach promises the ability to explore unconventional configurations with added confidence in the design due to the inclusion of a flutter constraint.

Finally, the advent of electric propulsion in aviation has presented new challenges and solutions. Distributed propulsion systems endeavor to solve some of the challenges and create higher aspect ratio designs. However, flutter analyses for conventional wings do not take the additional dynamic pressure of the propulsion or the gyroscopic effects into account. Further research is needed to account for these effects and to accurately predict flutter for these forthcoming designs.

APPENDIX A

Dependencies of UM/NAST Quantities

Table A.1: Dependency table for UM/NAST quantities.

	ε	$\dot{\varepsilon}$	$\ddot{\varepsilon}$	β	$\dot{\beta}$	λ	ζ	u
M_{FF}	●							
M_{FB}	●							
M_{BF}	●							
M_{BB}	●							
C_{FF}	●	●		●				
C_{FB}	●	●		●				
C_{BF}	●	●		●				
C_{BB}	●	●		●				
K_{FF}	●							
R_F	●	●	●	●	●	●	●	●
R_B	●	●	●	●	●	●	●	●
C_{GB}							●	
Ω_ζ				●				
F_1	●	●		●				
F_2	●	●		●				

APPENDIX B

Sequential Aggregation using Kreisselmeier-Steinhauser Functions

The Kreisselmeier-Steinhauser (KS) function is defined as:

$$KS\left(g\left(x\right)\right)=\frac{1}{\bar{\rho}}\ln\left(\sum_{j=1}^{m}e^{\bar{\rho}g_{j}(x)}\right)\tag{B.1}$$

Assuming that the parameter $\bar{\rho}$ is the same across both aggregations, the sequential application of the KS functions results in:

$$KS\left(KS\left(g\left(x\right)\right)\right)=\frac{1}{\bar{\rho}}\ln\left(\sum_{j=1}^{m}e^{\bar{\rho}\frac{1}{\bar{\rho}}\ln\left(\sum_{k=1}^{n}e^{\bar{\rho}g_{j}(x)}\right)}\right)\tag{B.2}$$

$$=\frac{1}{\bar{\rho}}\ln\left(\sum_{j=1}^{m}\sum_{k=1}^{n}e^{\bar{\rho}g_{k}(x)}\right)\tag{B.3}$$

This can be rewritten as:

$$KS\left(KS\left(g\left(x\right)\right)\right)=\frac{1}{\bar{\rho}}\ln\left(\sum_{i=1}^{N}e^{\bar{\rho}g_{i}(x)}\right)\tag{B.4}$$

$$=KS\left(g\left(x\right)\right)\tag{B.5}$$

As such, the aggregation using KS functions in series is equivalent to a single KS aggregation over all constraints. Despite this, there are scenarios in which a sequential aggregation may make sense. For example, in the case of a flutter constraint, the initial aggregation may find the most critical damping value at a search point, with the second aggregation giving the scalar constraint. The sequential aggregation therefore permits a parallel evaluation of the search points.

APPENDIX C

Computer Configurations

C.1 Configuration A

2 × Intel Xeon E5-2650 (2.0 GHz)

64 GB RAM

Ubuntu 18.04

C.2 Configuration B

4 × Intel Xeon E7 (2.1 GHz)

256 GB RAM

Ubuntu 18.04

C.3 Configuration C

Intel Core i5-8259U (2.3 GHz base clock, 3.8 GHz Turbo Boost)

8 GB RAM

Intel Iris Plus Graphics 655 1536 MB

macOS 10.15 “Catalina”

APPENDIX D

AD Gradients from Linearization-Based Solvers

One of the linearization schemes in UM/NAST uses AD to obtain the linearized A matrix such that:

$$\dot{y} = Ay \tag{D.1}$$

using the relation:

$$A = \frac{\partial \dot{y}}{\partial y} \tag{D.2}$$

As such, obtaining gradients from a linearization-based solver therefore requires second-order derivatives ($\frac{\partial A}{\partial y \partial x}$) to avoid the chain rule being broken during the AD evaluation. This requires special implementation for the linearization solver. To illustrate this, this chapter presents a simplified problem and the AD implementation used to obtain the design sensitivities. The state velocity $\dot{y}$ is modeled as a quadratic function of the state variable y:

$$\dot{y} = y^2 \tag{D.3}$$

The function of interest, for this problem, is defined as:

$$f = \frac{\partial \dot{y}}{\partial y} x^3 + \dot{y} \tag{D.4}$$

$$= 2x^3 y + y^2 \tag{D.5}$$

with the design variable x. The analytical gradients are given by:

$$A = \frac{\partial \dot{y}}{\partial y} = 2y \tag{D.6}$$

$$\frac{\partial f}{\partial x} = 6x^2 y \tag{D.7}$$

The object-oriented implementation of the example linearization problem and gradient recovery, evaluated at $x = 4.0$ and $y = 3.0$, is listed below:

```cpp
// standard C++ headers
#include <cmath>
#include <iostream>

// CoDiPack headers
#include "codi.hpp"

// use specific functions without the namespace within this execution unit
using std::cout;
using std::endl;
using std::pow;
using codi::RealReverseGen;
using codi::RealReverse;

// create a second-order reverse type definition
typedef RealReverseGen<RealReverse> O2Reverse;



/*
    A stand-in class for the linearization solver.
*/
template<typename Type>
class Linearization
{
public:
    // Constructor
    Linearization() {};

    // Destructor
    ~Linearization() {};

    // set the state variable
    void SetState(double y) { y_ = y; };

    // get the state velocity
    Type GetYdot() { return ydot_.value(); };

    // get the linearized value
    Type GetLinearized() { return a_; };

    // Solve linearization
    void SolveLinearization()
    {
        // get the RealReverseGen<Type> tape
        RealReverseGen<RealReverse>::TapeType &tape = RealReverseGen<RealReverse>::getGlobalTape();

        RealReverseGen<RealReverse> y = y_;

        // set the state variable gradient to 1.0 (as input of interest)
        y.gradient() = 1.0;

        // activate the tape used to determine df/dy and register the inputs (in
        // this case the state variable)
        tape.setActive();
        tape.registerInput(y);

        // evaluate the function and register it as an output to the tape (2)
        ydot_ = pow(y, 2);
        tape.registerOutput(ydot_);
        ydot_.gradient() = 1.0;

        cout << "ydot(y) (double): " << pow(y_, 2) << endl;
        cout << "ydot(y) (AD): " << ydot_ << endl << endl;

        // set the tape (2) to passive
        tape.setPassive();
        tape.evaluate();

        // evaluate the gradient and reset the tape
        a_ = y.gradient();
        tape.reset();

        cout << "dydot/dy (double): " << 2 * y << endl;
        cout << "dydot/dy (AD): " << a_ << endl << endl;
    };

protected:
    // state variable
    double y_;

    // state velocity variable
    RealReverseGen<Type> ydot_;

    // linearized variable
    Type a_;

private:
};


/*
    A stand-in class for a solver that uses the linearization.
*/
class Solver
{
public:
    // Constructor
    Solver() {};

    // Destructor
    ~Solver() {};

    // set design variable
    void SetDesignVar(double x) { x_ = x; };

    // set state variable
    void SetState(double y) { y_ = y; };

    // Function of interest
    template<typename Type>
    Type SolveFunction(Type x)
    {
        // create a linearization solver
        Linearization<Type> lin;

        // set the state variable
        lin.SetState(y_);

        // solve the linearization
        lin.SolveLinearization();

        // get the state velocity
        Type ydot = lin.GetYdot();

        // get the linearized value
        Type a = lin.GetLinearized();

        return a * pow(x, 3) + ydot;
    };


    // Gradient of the function of interest
    double SolveGradient()
    {
        // get the RealReverse tape
        RealReverse::TapeType& tape = RealReverse::getGlobalTape();

        // assign the previously defined value of x double, to the gradient variable
        // and set the gradient of x to 1.0 (as input of interest to tape 1)
        RealReverse x = x_;
        x.gradient() = 1.0;

        // activate the tape and register inputs
        tape.setActive();
        tape.registerInput(x);

        // evaluate the function of interest and register it to the tape
        RealReverse f = SolveFunction(x);
        tape.registerOutput(f);
        f.gradient() = 1.0;

        // set the tape to passive
        tape.setPassive();
        tape.evaluate();

        cout << "f(x,y) (double): " << 2 * y_ * pow(x_, 3) + pow(y_, 2) << endl;
        cout << "f(x,y) (AD): " << f << endl << endl;

        // determine and return the gradient of the function of interest w.r.t.
        // the design variables
        return x.getGradient();
    };


protected:
    // design variable
    double x_;

    // state variable
    double y_;

private:
};



/*
    Program main function.
*/
int main()
{
    // set the values for x and y (as doubles first)
    double x = 4.0;
    double y = 3.0;

    // create the solver object
    Solver solver;

    // set the design and state variables
    solver.SetDesignVar(x);
    solver.SetState(y);

    // solve for the gradient
    double dfdx = solver.SolveGradient();

    cout << "df/dx (double): " << 6 * y * pow(x, 2) << endl;
    cout << "df/dx (AD): " << dfdx << endl << endl;

    return 0;
}
```

Because the linearization process involves an AD gradient evaluation, as does the gradient evaluation w.r.t. the design variables, a nested AD implementation is required:

```
RealReverseGen<RealReverse> ...
```

This also requires two separate tape evaluations. Inside the Linearization class, the SolveLinearization function contains the linearization calculations using the nested CoDiPack type. Notice that the tape reference inside this function is of the nested type as well. The resulting linearization variable a therefore is of the type *RealReverse*. By ensuring this, the chain rule evaluation remains uninterrupted during the determination of the design gradients. Finally, the outer tape evaluation determines the gradients of the function of interest with respect to the design variables.

For the example problem the listed code yields the following results:

	Analytical	AD
$\dot{y}$	9.0	9.0
A	6.0	6.0
$f(x, y)$	393.0	393.0
$\frac{\partial f}{\partial x}$	288.0	288.0

Bibliography

[1] Bradley, M. K. and Droney, C. K., "Subsonic Ultra Green Aircraft Research: Phase I Final Report," Tech. Rep. CR-2011-216847, NASA, April 2011.

[2] Bradley, M. K. and Droney, C. K., "Subsonic Ultra Green Aircraft Research: Phase II: N+4 Advanced Concept Development," Tech. Rep. CR-2012-217556, NASA, May 2012.

[3] Knoll, T. E., Brown, J. M., Perez-Davis, M. E., Ishmael, S. D., Tiffany, G. C., and Gaier, M., "Investigation of the Helios Prototype Aircraft Mishap," Tech. rep., NASA, January 2004.

[4] Su, W. and Cesnik, C. E. S., "Nonlinear Aeroelasticity of a Very Flexible Blended-Wing-Body Aircraft," *Journal of Aircraft*, Vol. 47, No. 5, 2010, pp. 1539–1553.

[5] Su, W. and Cesnik, C. E. S., "Strain-Based Analysis for Geometrically Nonlinear Beams: A Modal Approach," *Journal of Aircraft*, Vol. 51, No. 3, 2012, pp. 890–903.

[6] Cesnik, C. E. S., Senatore, P., Su, W., Atkins, E. M., and Shearer, C. M., "XHALE: A Very Flexible UAV for Nonlinear Aeroelastic Tests," *AIAA Journal*, Vol. 50, 2012, pp. 2820–2833.

[7] NASA, "Helios Mishap Photo Previews," Images obtained from: https://www.nasa.gov/centers/dryden/news/ResearchUpdate/Helios/Previews/index.html, accessed May 9, 2020.

[8] Garrigues, E., "A Review of Industrial Aeroelasticity Practices at Dassault Aviation for Military Aircraft and Business Jets," *Journal Aerospace Lab*, , No. 14, 2018, pp. 1–34.

[9] Haftka, R. T., "Automated procedure for design of wing structures to satisfy strength and flutter requirements," Tech. Rep. TN D-7264, NASA, July 1973.

[10] Sobieszczanski-Sobieski, J. and Haftka, R. T., "Multidisciplinary Aerospace Design Optimization: Survey of Recent Developments," *Structural Optimization*, Vol. 14, No. 1, 1997, pp. 1–23.

[11] Kenway, G. K. W. and Martins, J. R. R. A., "Multipoint High-Fidelity Aerostructural Optimization of a Transport Aircraft Configuration," *Journal of Aircraft*, Vol. 51, No. 1, 2014, pp. 144–160.

[12] Variyar, A., Economon, T. D., and Alonso, J. J., "Design and Optimization of Unconventional Aircraft Configurations with Aeroelastic Constraints," *55th AIAA Aerospace Sciences Meeting*, 2017, pp. 1–10.

[13] Webster, M., "fidelity," `https://www.merriam-webster.com/dictionary/fidelity`, accessed May 9, 2020.

[14] Jonsson, E., Riso, C., Lupp, C. A., Cesnik, C. E. S., Martins, J. R., and Epureanu, B. I., "Flutter and post-flutter constraints in aircraft design optimization," *Progress in Aerospace Sciences*, Vol. 109, 2019, pp. 100537.

[15] Hassig, H. J., "An approximate true damping solution of the flutter equation by determinant iteration," *Journal of Aircraft*, Vol. 8, No. 11, 1971, pp. 885–889.

[16] Wright, J. R. and Jonathan, E. C., *Introduction to Aircraft Aeroelasticity and Loads*, Wiley, 2nd ed., 2007.

[17] Hodges, D. H. and Pierce, G. A., *Introduction to Structural Dynamics and Aeroelasticity*, Vol. 15, Cambridge University Press, 2nd ed., 2011.

[18] Rodden, W. P., *Theoretical and computational aeroelasticity*, Crest Publishers, Burbank, CA, 2011.

[19] Chen, P. C., "Damping Perturbation Method for Flutter Solution: the g-Method," *AIAA Journal*, Vol. 38, No. 9, 2000, pp. 1519–1524.

[20] Eberhart, R. and Kennedy, J., "A new optimizer using particle swarm theory," *Systemic Control Design by Optimizing a Vector Performance Index*, 6th International Symposium on Micro Machine and Human Science, Nagoya, Japan, 1995.

[21] Goldberg, D. E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Longman Publishing, Boston, MA, 1989.

[22] Lyu, Z., Xu, Z., and Martins, J. R. R. A., "Benchmarking Optimization Algorithms for Wing Aerodynamic Design Optimization," *Proceedings of the 8th International Conference on Computational Fluid Dynamics*, Chengdu, Sichuan, China, July 2014.

[23] Yu, Y., Lyu, Z., Xu, Z., and Martins, J. R. R. A., "On the Influence of Optimization Algorithm and Starting Design on Wing Aerodynamic Shape Optimization," *Aerospace Science and Technology*, Vol. 75, April 2018, pp. 183–199.

[24] Haftka, R. T., "Parametric Constraints with Application to Optimization for Flutter Using a Continuous Flutter Constraint," *AIAA Journal*, Vol. 13, No. 4, 1975, pp. 471–475.

[25] Hajela, P., “A Root Locus-Based Flutter Synthesis Procedure,” *Journal of Aircraft*, Vol. 20, No. 12, 1983, pp. 1021–1027.

[26] Bhatia, K. G. and Rudisill, C. S., “Optimization of Complex Structures to Satisfy Flutter Requirements,” *AIAA Journal*, Vol. 9, No. 8, 1971, pp. 1487–1491.

[27] Rudisill, C. S. and Bhatia, K. G., “Second Derivatives of the Flutter Velocity and the Optimization of Aircraft Structures,” *AIAA Journal*, Vol. 10, No. 12, 1972, pp. 1569–1572.

[28] Gwin, L. and Taylor, R., “A general method for flutter optimization,” *14th Structures, Structural Dynamics, and Materials Conference*, Williamsburg, Virginia, 1973, pp. 1–6.

[29] Livne, E., Schmit, L., and Friedmann, P., “An integrated approach to the optimum design of actively controlled composite wings,” *30th Structures, Structural Dynamics and Materials Conference*, Mobile, AL, 1990.

[30] Livne, E., Schmit, L. A., and Friedmann, P. P., “Integrated structure/control/aerodynamic synthesis of actively controlled composite wings,” *Journal of Aircraft*, Vol. 30, No. 3, 1993, pp. 387–394.

[31] Ringertz, U. T., “On Structural Optimization with Aeroelasticity Constraints,” *Structural Optimization*, Vol. 8, No. 1, 1994, pp. 16–23.

[32] Stanford, B. K., Wieseman, C. D., and Jutte, C. V., “Aeroelastic Tailoring of Transport Wings Including Transonic Flutter Constraints,” *56th AIAA/ASME/ASCE/AHS/SC Structures, Structural Dynamics, and Material Conference*, Kissimmee, FL, 2015, pp. 1–22.

[33] Brooks, T. R., Kenway, G. K. W., and Martins, J. R. R. A., “Benchmark Aerostructural Models for the Study of Transonic Aircraft Wings,” *AIAA Journal*, Vol. 56, July 2018, pp. 2840–2855.

[34] Jonsson, E., Mader, C. A., Kennedy, G. J., and Martins, J. R. R. A., “Computational Modeling of Flutter Constraint for High-Fidelity Aerostructural Optimization,” *2019 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, San Diego, CA, January 2019.

[35] Kreisselmeier, G. and Steinhauser, R., “Systemic Control Design by Optimizing a Vector Performance Index,” *Computer Aided Design of Control Systems: Proceedings of the IFAC Symposium, Zürich, Switzerland, 29-31 August 1979*, No. 1, IFAC, Zurich, Switzerland, 1979, pp. 1–10.

[36] Xie, C., Meng, Y., Wang, F., and Wan, Z., “Aeroelastic Optimization Design for High-Aspect-Ratio Wings with Large Deformation,” *Schock and Vibration*, Vol. 2017, 2017, pp. 16.

[37] Lukaczyk, T., Wendor, A., Boteroz, E., Macdonaldz, T., Momosez, T., Variyarz, A., Veghz, J., Colonnox, M., Economon, T., Alonsok, J., Orra, T., and Da Silvayy, C., "SUAVE: An Open-Source Environment for Multi-Fidelity Conceptual Vehicle Design," *16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Dallas, TX, June 22–26 2015.

[38] Drela, M., "Integrated Simulation Model for Preliminary Aerodynamic, Structural, and Control-Law Design of Aircraft," *40th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, St. Louis, MO, 1999.

[39] Bhatia, M. and Beran, P., "Design of Thermally Stressed Panels Subject to Transonic Flutter Constraints," *Journal of Aircraft*, Vol. 54, No. 6, 2017, pp. 2340–2349.

[40] Cesnik, C. E. S., Palacios, R., and Reichenbach, E. Y., "Reexamined Structural Design Procedures for Very Flexible Aircraft," *Journal of Aircraft*, Vol. 51, No. 5, 2014, pp. 1580–1591.

[41] Bryson, D. E. and Rumpfkeil, M. P., "Aerostructural design optimization using a multifidelity quasi-Newton method," *Journal of Aircraft*, Vol. 56, No. 5, 2019, pp. 2019–2031.

[42] Opgenoord, M. M. J., Drela, M., and Willcox, K. E., "Physics-Based Low-Order Model for Transonic Flutter Prediction," *AIAA Journal*, Vol. 56, No. 4, 2018, pp. 1519–1531.

[43] Opgenoord, M. M. and Willcox, K. E., "Aeroelastic Tailoring using Additively Manufactured Lattice Structures," *2018 Multidisciplinary Analysis and Optimization Conference*, Atlanta, GA, 2018.

[44] Stodieck, O., Cooper, J. E., Neild, S. A., Lowenberg, M. H., and Iorga, L., "Slender-Wing Beam Reduction Method for Gradient-Based Aeroelastic Design Optimization AIAA JOURNAL," 2018, pp. 1–17.

[45] Malcolm, D. J. and Laird, D. L., "Extraction of Equivalent Beam Properties from Blade Models," *Wind Energy*, Vol. 10, 2007, pp. 135–157.

[46] Martins, J. R. R. A., *Multidisciplinary Design Optimization*, University of Michigan, Ann Arbor, MI, 2017.

[47] Lyness, J. N. and Moler, C. B., "Numerical differentiation of analytic functions," *SIAM Journal on Numerical Analysis*, Vol. 4, No. 2, 1967, pp. 202–210.

[48] Lyness, J. N., "Numerical algorithms based on the theory of complex variable," *ACM National Meeting*, Thompson Book Co., Washington, D.C., pp. 125–133.

[49] Squire, W. and Trapp, G., "Using complex variables to estimate derivatives of real functions," *SIAM Review*, Vol. 40, No. 1, 1998, pp. 110–112.

[50] Martins, J. R. R. A., Sturdza, P., and Alonso, J. J., "The complex-step derivative approximation," *ACM Transactions on Mathematical Software*, Vol. 29, No. 3, 2003, pp. 245–262.

[51] Martins, J. R. R. A., "Complexify: A Header File for Using the Complex Step Method in C++," .

[52] Hascoet, L. and Pascual, V., *Tapenade 2.1 user's guide*, 2004.

[53] Peters, D. A., Karunamoorthy, S., and Cao, W., "Finite State Induced Flow Models. I - Two-Dimensional Thin Airfoil," *Journal of Aircraft*, Vol. 32, No. 2, 01 1995, pp. 313–322.

[54] Teixeira, P. C. and Cesnik, C. E. S., "Inclusion of Propeller Effects on Aeroelastic Behavior of Very Flexible Aircraft," *International Forum on Aeroelasticity and Structural Dynamics*, Como, Italy, 2017, pp. 1–18.

[55] Teixeira, P. C. and Cesnik, C. E. S., "Propeller Effects on the Dynamic Response of HALE Aircraft," *AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Kissimee, FL, 2018.

[56] Brown, E., *Integrated Strain Actuation In Aircraft With Highly Flexible Composite Wings*, Ph.D. thesis, University of Michigan, Ann Arbor, Michigan, 2003.

[57] Shearer, C. M., *Coupled Nonlinear Flight Dynamics, Aeroelasticity, and Control of Very Flexible Aircraft*, Ph.D. thesis, University of Michigan, Ann Arbor, Michigan, 2006.

[58] Su, W., *Coupled Nonlinear Aeroelasticity and Flight Dynamics of Fully Flexible Aircraft*, Ph.D. thesis, University of Michigan, Ann Arbor, Michigan, 2008.

[59] Dillsaver, M., *Gust Response and Control of Very Flexible Aircraft*, Ph.D. thesis, University of Michigan, Ann Arbor, Michigan, 2013.

[60] Jones, J., *Development of a Very Flexible Testbed Aircraft for the Validation of Nonlinear Aeroelastic Codes*, Ph.D. thesis, University of Michigan, Ann Arbor, Michigan, 2017.

[61] Pang, Z., *Modeling, Simulation and Control of Very Flexible Unmanned Aerial Vehicle*, Ph.D. thesis, University of Michigan, Ann Arbor, Michigan, 2018.

[62] Kitson, R. C., *Fluid-Structure-Jet Interaction Effects on High-Speed Vehicles*, Ph.D. thesis, University of Michigan, Ann Arbor, Michigan, 2018.

[63] Teixeira, P. C., *Propeller Effects on Very Flexible Aircraft*, Ph.D. thesis, University of Michigan, Ann Arbor, Michigan, 2019.

[64] Kitson, R. C. and Cesnik, C. E. S., "Modelling of High Aspect Ratio Active Flexible Wings for Roll Control," *International Forum on Aeroelasticity and Structural Dynamics*, Saint Petersburg, Rusia, 2015.

[65] Skujins, T. and Cesnik, C. E. S., "Reduced-Order Modeling of Unsteady Aerodynamics Across Multiple Mach Regimes," *Journal of Aircraft*, Vol. 51, No. 6, 2014, pp. 1681–1704.

[66] Allemang, R. J., *Investigation of Some Multiple Input/Output Frequency Response Function Experimental Modal Analysis Techniques*, Ph.D. thesis, University of Cincinnati, Cincinnati, 1980.

[67] Allemang, R. J. and Brown, D. L., "A Correlation Coefficient for Modal Vector Analysis," *International Modal Analysis Con- ference*, 1982, pp. 110–116.

[68] Wallis, J., *A Treatise of Algebra, both Historical and Practical. Shewing the Original, Progress, and Advancement thereof, from time to time, and by what Steps it hath attained to the Heighth at which it now is*, Oxford: Richard Davis, 1685.

[69] Bradie, B., *A friendly introduction to numerical analysis*, Pearson Prentice Hall, Upper Saddle River, New Jersey, 2006.

[70] Henshaw, M. J. d. C., Badcock, K. J., Vio, G. A., Allen, C. B., Chamberlain, J., Kaynes, I., Dimitriadis, G., Cooper, J. E., Woodgate, M. A., Rampurawala, A. M., Jones, D., Fenwick, C., Gaitonde, A. L., Taylor, N. V., Amor, D. S., Eccles, T. A., and Denley, C. J., "Non-linear Aeroelastic Prediction for Aircraft Applications," *Progress in Aerospace Sciences*, Vol. 43, No. 4-6, 2007, pp. 65–137.

[71] Jonsson, E., Kenway, G., and Martins, J. R. R. A., "Development of Flutter Constraints for High-fidelity Aerostructural Optimization," *35th AIAA Applied Aerodynamics Conference*, AIAA, Denver, CO, 2017, p. 26.

[72] Lambe, A. B., Martins, J. R. R. A., and Kennedy, G. J., "An evaluation of constraint aggregation strategies for wing box mass minimization," *Structural and Multidisciplinary Optimization*, Vol. 55, No. 1, 2017, pp. 257–277.

[73] Raspanti, C., Bandoni, J., and Biegler, L., "New Strategies for Flexibility Analysis and Design Under Uncertainty," *Computers & Chemical Engineering*, Vol. 24, No. 9-10, 2000, pp. 2193–2209.

[74] Golchi, S. and Loeppky, J. L., "Monte Carlo based Designs for Constrained Domains," *ArXiv e-prints*, Vol. 7, 2015, pp. 1–28.

[75] Feinberg, J. and Langtangen, H. P., "Chaospy: An Open Source Tool for Designing Methods of Uncertainty Quantification," *Journal of Computational Science*, Vol. 11, November 2015, pp. 46–57.

[76] Heath, T. L., *A History of Greek Mathematics*, Vol. 2, Oxford University Press, 1921, pp. 321–323.

[77] Sanghi, D. and Cesnik, C. E. S., "Enhanced Fem2Stick Formulation," University of Michigan CASE VFA Tech. rep., 2018.

[78] Sagebaum, M., Albring, T., and Gauger, N. R., "High-Performance Derivative Computations using CoDiPack," *arXiv preprint arXiv:1709.07229*, 2017.

[79] Hogan, R. J., "Adept C ++ Software Library: User Guide," 2016.

[80] Economon, T. D., Palacios, F., Copeland, S. R., Lukaczyk, T. W., and Alonso, J. J., "SU2: An Open-Source Suite for Multiphysics Simulation and Design," *AIAA Journal*, Vol. 54, No. 3, 2016, pp. 828–846.

[81] Sagebaum, M. and Gauger, N., "MeDiPack Documentation," 2016, `https://www.scicomp.uni-kl.de/medi/`.

[82] Rosenbrock, H., "An Automatic Method for finding the Greatest of Least Value of a Function," *The Computer Journal*, Vol. 3, No. 3, 1960, pp. 175–184.

[83] Gray, J. S., Hwang, J. T., Martins, J. R. R. A., Moore, K. T., and Naylor, B. A., "OpenMDAO : An open-source framework for multidisciplinary design, analysis, and optimization," *Structural and Multidisciplinary Optimization*, Vol. 59, No. 4, 2019, pp. 1–39.

[84] Yildirim, A., Mader, C., Martins, J. R. R. A., Kennedy, G., Gray, J., Stanford, B., and Jacobson, K., "Modular Aerostructural Analysis and Optimization with the OpenMDAO Framework," *AIAA Aviation Forum*, American Institute of Aeronautics and Astronautics, Reno, NV (presentation only), 2020.

[85] Kennedy, G. J. and Martins, J. R. R. A., "A parallel finite-element framework for large-scale gradient-based design optimization of high-performance structures," *Finite Elements in Analysis and Design*, 2014.

[86] Mader, C. A., Kenway, G. K. W., Yildirim, A., and Martins, J. R. R. A., "ADflow: An Open-Source Computational Fluid Dynamics Solver for Aerodynamic and Multidisciplinary Optimization," *Journal of Aerospace Information Systems*, Vol. 0, No. 0, 0, pp. 1–20.

[87] Jacobson, K., Kiviaho, J. F., Smith, M. J., and Kennedy, G., "An Aeroelastic Coupling Framework for Time-accurate Analysis and Optimization," *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Kissimmee, FL.

[88] Jasa, J. P., Hwang, J. T., and Martins, J. R. R. A., "Open-source coupled aerostructural optimization using Python," *Structural and Multidisciplinary Optimization*, Vol. 57, No. 4, April 2018, pp. 1815–1827.

[89] "Google C++ Style Guide," 2020, `https://google.github.io/styleguide/cppguide.html`.

[90] Albring, T. A., Sagebaum, M., and Gauger, N. R., *Efficient Aerodynamic Design using the Discrete Adjoint Method in SU2*.

[91] Lockyer, A. J., Drake, A., Bartley-Cho, J., Vartio, E., Solomon, D., and Shimko, T., "High Lift Over Drag Active (HiLDA) Wing," Tech. Rep. AFRL-VA-WP-TR-2005-3066, 2005.

[92] Gray, J. S., Hearn, T. A., Moore, K. T., Hwang, J., Martins, J., and Ning, A., "Automatic Evaluation of Multidisciplinary Derivatives Using a Graph-Based Problem Formulation in OpenMDAO," *15th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, American Institute of Aeronautics and Astronautics, Atlanta, Georgia, 2014.

[93] Hwang, J. T., *A Modular Approach to Large-Scale Design Optimization of Aerospace Systems*, Ph.D. thesis, University of Michigan, 2015.

[94] Martins, J. R. R. A. and Poon, N. M. K., "On Structural Optimization Using Constraint Aggregation," *Proceedings of the 6th World Congress on Structural and Multidisciplinary Optimization*, Rio de Janeiro, Brazil, May 2005.

[95] Jones, E., Oliphant, T., Peterson, P., et al., "SciPy: Open Source Scientific Tools for Python," 2001, http://www.scipy.org/.

[96] Anderson, J., *Introduction to Flight*, McGraw-Hill, New York, NY, 8th ed., 2011.

[97] Cavcar, M., "Bréguet Range Equation?" *Journal of Aircraft*, Vol. 43, No. 5, 2006, pp. 1542–1544.

[98] Devillers, R., *La Dynamique de l'Avion*, University of Michigan Library, Ann Arbor, MI, 1920.

[99] Coffin, J. G., "A Study of Airplane Range and Useful Loads," Tech. Rep. NACA-TR-69, 1920.

[100] Breguet, L., "Calcul du Poids de Combustible Consummé par un Avion en Vol Ascendant," *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, Vol. 177, 1923, pp. 870–872.

[101] Giles, M. B., "Collected matrix derivative results for forward and reverse mode algorithmic differentiation," *Lecture Notes in Computational Science and Engineering*, Vol. 64 LNCSE, No. 08, 2008, pp. 35–44.

[102] Wang, Q. Z., Cesnik, C. E. S., and Fidkowski, K., "Multivariate Recurrent Neural Network Models for Scalar and Distribution Predictions in Unsteady Aerodynamics," *AIAA Scitech 2020 Forum*, AIAA, 2018.

[103] Wolkovitch, J., "The joined wing – An overview," *Journal of Aircraft*, Vol. 23, No. 3, 1986, pp. 161–178.