8-2019

Numerical Analysis and Spanwise Shape Optimization for Finite Wings of Arbitrary Aspect Ratio

Joshua D. Hodson
Utah State University

NUMERICAL ANALYSIS AND SPANWISE SHAPE OPTIMIZATION

FOR FINITE WINGS OF ARBITRARY ASPECT RATIO

By

Joshua D. Hodson

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

In

Mechanical Engineering

Approved:

Douglas F. Hunsaker, Ph.D. Major Professor	Geordie Richards, Ph.D. Committee Member
Robert E. Spall, Ph.D. Major Professor	James J. Joo, Ph.D. Committee Member
Barton L. Smith, Ph.D. Committee Member	Richard S. Inouye, Ph.D. Vice Provost for Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2019

Dedicated to my father

Lee James Hodson

1939-2018

ABSTRACT

Numerical Analysis and Spanwise Shape Optimization

for Finite Wings of Arbitrary Aspect Ratio

by

Joshua D. Hodson, Doctor of Philosophy

Utah State University, 2019

Major Professors: Dr. Douglas Hunsaker and Dr. Robert Spall

Department: Mechanical and Aerospace Engineering

Improvements to the current state of the art in wing shape optimization for morphing wing applications are presented, with a focus on low-fidelity analysis methods for preliminary design. An existing aerodynamic analysis tool based on lifting line theory is the foundation upon which this work builds, and several software development efforts are presented that enhance the capabilities of this tool relative to wing shape optimization. An automatic differentiation tool is integrated with the aerodynamic analysis tool to facilitate accurate and efficient derivative calculations for gradient-based optimization. A light-weight optimization framework written in Python is presented that is capable of efficiently searching the design space using popular gradient-based optimization techniques and parallelization of independent objective function evaluations. Several example optimization problems are presented using this toolset, and a method for visualizing the design space of morphing wings using this toolset is also presented and discussed.

The morphing wing application of primary interest to the current work, the Variable Camber Compliant Wing developed at the United States Air Force Research Laboratory, has an aspect ratio that falls below the acceptable range of aspect ratios for lifting line theory analysis. This has led to the development of a new method for applying lifting line theory to low-aspect-ratio lifting surfaces. A thorough review of Prandtl’s classical lifting line theory is first presented, followed by several low-aspect-ratio proposals from the slender wing theory of Jones to the lifting surface theories of Birnbaum, Blenk, and others. A new formulation for slender wing theory is presented that provides new insights into the appropriate limits for a formulation capable of spanning the entire range of aspect ratios from slender to infinite. A new empirical relation is proposed that satisfies the limits at both extremes and agrees well with high-order inviscid panel code results. A method is then presented for implementing this empirical relation in both analytical and numerical lifting line algorithms. A comparison of results computed using this method to experimental results for the Variable Camber Compliant Wing is also given.

(327 pages)

PUBLIC ABSTRACT

Numerical Analysis and Spanwise Shape Optimization

for Finite Wings of Arbitrary Aspect Ratio

by

Joshua D. Hodson, Doctor of Philosophy

Utah State University, 2019

Major Professors: Dr. Douglas Hunsaker and Dr. Robert Spall

Department: Mechanical and Aerospace Engineering

This work focuses on the development of efficient methods for wing shape optimization for morphing wing technologies. Existing wing shape optimization processes typically rely on computational fluid dynamics tools for aerodynamic analysis, but the computational cost of these tools makes optimization of all but the most basic problems intractable. In this work, we present a set of tools that can be used to efficiently explore the design spaces of morphing wings without reducing the fidelity of the results significantly. Specifically, this work discusses automatic differentiation of an aerodynamic analysis tool based on lifting line theory, a light-weight gradient-based optimization framework that provides a parallel function evaluation capability not found in similar frameworks, and a modification to the lifting line equations that makes the analysis method and optimization process suitable to wings of arbitrary aspect ratio. The toolset discussed is applied to several wing shape optimization problems. Additionally, a method for visualizing the design space of a morphing wing using this toolset is presented. As a result of this work, a light-weight wing shape optimization method is available for analysis of morphing wing designs that reduces the computational cost by several orders of magnitude over traditional methods without significantly reducing the accuracy of the results.

ACKNOWLEDGMENTS

I have had several mentors throughout the course of this research, and I would like to express my gratitude to each for helping me along in this journey: to Dr. Hope Forsmann at the Idaho National Laboratory for her guidance and expertise in nuclear reactor thermal hydraulics and the Fortran programming language, to Dr. James Joo at the U.S. Air Force Research Laboratory for his help in understanding morphing wing systems and their role in the future of aerospace systems, and to Drs. Stephen Whitmore and Warren Phillips at Utah State University for helping me realize my passion for aerospace. Also to Dalon Work and Tyson Etherington for their advice, encouragement, and friendship.

Three professors in particular have had an exceptional impact on my education. First, Dr. Robert Spall, who has been guide and mentor to me from my first day as a freshman undergraduate student to my last day as a PhD candidate 20 years later, and who first introduced me to the worlds of computational fluid dynamics, aerodynamics, and high performance computing. I wish him all the best as he sets sail this summer. Second, Dr. Barton Smith, who has been a mentor for nearly as long and who reminds me that there is a real world outside my numerical simulations, without which my numerical simulations would have no meaning. I also want to thank Dr. Smith for connecting me with the RELAP5-3D team at Idaho National Laboratory. Finally, Dr. Douglas Hunsaker, who among all his other responsibilities as a new faculty member at USU gave me a majority share of his time and resources. I am forever indebted to these three men for their mentorship and commitment to my success.

I would like to thank my wife Megan and my kids Wesley, Abigail, Preston, and Aliya, for supporting me on a daily basis as I furthered my education. They have each made sacrifices so that I could pursue my goal of earning a doctoral degree, and I thank them for sticking by me throughout it all. I also want to thank my mother for her continual words of encouragement, which on a few occasions came at times when I was ready to quit but gave me the boost I needed to forge ahead; and my father for his examples of hard work and perseverance that have motivated me to test the limits of my abilities and always strive to do my best.

I would also like to thank those organizations that provided financial support for this work. Portions of this research have been funded under a grant from the Office of Naval Research Sea-Based Aviation Program (grant no. N00014-16-1-2074) and fellowships from the Department of Energy Nuclear Energy University

Program, the U.S. Air Force Research Laboratory's Summer Faculty Fellowship Program, and the USU School of Graduate Studies Dissertation Fellowship Program.

Finally, I thank my Heavenly Father for His guidance and sustainment throughout the course of this work. I acknowledge His hand in this and all things.

Joshua D. Hodson

CONTENTS

Page

ABSTRACT iv

PUBLIC ABSTRACT vi

ACKNOWLEDGMENTS vii

LIST OF TABLES xiii

LIST OF FIGURES xiv

LIST OF ACRONYMS xvii

NOMENCLATURE xviii

1 INTRODUCTION 1

2 DUAL NUMBER AUTOMATIC DIFFERENTIATION FOR WING SHAPE OPTIMIZATION 4

2.1 Introduction 4

2.2 Derivative Calculation Methods 5

2.2.1 Symbolic Differentiation 5

2.2.2 Finite Differencing 9

2.2.3 Automatic Differentiation 11

2.2.4 Discussion 14

2.3 Methods and Tools for Forward-Mode Automatic Differentiation 15

2.4 Dual Number Automatic Differentiation 17

2.4.1 Dual Number Theory 18

2.4.2 The Dual Number Data Type 19

2.4.3 Integration with Fortran Algorithms 20

2.4.4 Special Considerations when Integrating DNAD with Existing Fortran Algorithms 23

2.5 Automatic Differentiation of a 1D Scalar Transport Equation Solver 26

2.5.1 Mathematical Model 26

2.5.2 Numerical Model 28

2.5.3 Automatic Differentiation of the 1D Scalar Transport Problem using DNAD 29

2.5.4 Results and Discussion 30

2.6 Automatic Differentiation of MachUp 36

2.6.1 The MachUp Numerical Lifting Line Solver 37

2.6.2 Modifications to the MachUp Source Code 39

2.6.3 Results and Discussion 41

3 WING SHAPE OPTIMIZATION USING A NUMERICAL LIFTING LINE ALGORITHM AND DUAL NUMBER AUTOMATIC DIFFERENTIATION 46

3.1 Introduction 46

3.2 Optix...47

3.2.1 Optimization Method...47

3.2.2 Code Structure 48

3.2.3 Parallel Execution of Independent Function Evaluations 49

3.2.4 Linear and Quadratic Line Searching 49

3.2.5 Limitations...50

3.3 Wing Shape Optimization in Inviscid Flow 51

3.3.1 Optimized Planform Shapes for Minimum Induced Drag 52

3.3.2 Optimized Geometric Twist Distributions for Minimum Induced Drag 54

3.3.3 Optimized Aerodynamic Twist Distributions for Minimum Induced Drag............56

3.4 Wing Shape Optimization for Viscous Flow 58

3.4.1 Optimized Planform Shapes for Minimum Total Drag 60

3.4.2 Optimized Geometric Twist Distributions for Minimum Total Drag.....................61

3.4.3 Optimized Aerodynamic Twist Distributions for Minimum Total Drag 63

3.5 Visualization of the Design Space for Wing Shape Optimization...................................64

4 ANALYTICAL LIFTING LINE METHOD FOR WINGS OF ARBITRARY ASPECT RATIO 68

4.1 Introduction 68

4.2 Lift Generated by a Finite Wing..69

4.3 Classical Lifting Line Theory...70

4.4 Slender Wing Theory 77

4.5 Lifting Surface Theory 81

4.6 A Unifying Formulation of Lifting Line, Slender Wing, and Lifting Surface Methods 82

4.7 Results and Discussion 92

5 NUMERICAL LIFTING LINE METHOD FOR WINGS OF ARBITRARY ASPECT RATIO 103

5.1 Introduction 103

5.2 The Phillips and Snyder Numerical Lifting Line Method 104

5.3 Modifications to the Numerical Lifting Line Method for Wings of Arbitrary Aspect Ratio 108

5.4 Results and Discussion 110

6 SUMMARY AND CONCLUSIONS 124

REFERENCES 127

APPENDICES 137

A MODIFIED DNAD SOURCE CODE 138

B ONE-DIMENSIONAL SCALAR TRANSPORT SOLVER IN FORTRAN 168

B.1 User Interface 168

B.2 Physics Module 169

B.3 I/O Module 172

C SUMMARY OF CODE CHANGES TO APPENDIX B FOR DNAD INTEGRATION 181

C.1 Changes to `adpsolver` Module 181

C.2 Changes to `adpio` Module 181

D SUMMARY OF CODE CHANGES TO MACHUP FOR DNAD INTEGRATION 185

D.1 Changes to `loads_m` Module 185

D.2 Changes to `plane_m` Module 185

D.3 Changes to `special_functions_m` Module 191

D.4 Changes to `view_m` Module 195

D.5 Changes to `wing_m` Module 196

D.6 Changes to `airfoil_m` Module 198

D.7 Changes to `atmosphere_m` Module 198

D.8 Changes to `dataset_m` Module 199

D.9 Changes to `math_m` Module 199

D.10 Changes to `section_m` Module 200

D.11 Changes to `myjson_m` Module 201

E EXAMPLE MACHUP INPUT FILES 209

E.1 Example JSON-Formatted Input File 209

E.2 Example Parameter File for a NACA 2412 Airfoil in Inviscid Incompressible Flow 209

E.3 Example Profile File for a NACA 2412 Airfoil 210

F OPTIX SOURCE CODE 211

G INPUT FILES AND PYTHON SCRIPTS FOR WING SHAPE OPTIMIZATION 225

G.1 Main Optimization Execution Script 225

G.2 Objective Function (`obj_fcn.evaluate`) 226

G.3 Objective Function with Gradient Calculations (`obj_fcn.evaluate_with_gradient`) 228

G.4 Helper Function for Extracting Variable Names (`get_list_of_vars`) 230

G.5 Main MachUp Input File 230

H AERODYNAMIC CALCULATIONS USING PANAIR 233

I MACHUP FUNCTIONS FOR WRITING PANAIR INPUT FILES 239

J EXAMPLE PANAIR INPUT FILE 245

K PRALINES SOURCE CODE 250

K.1 Pralines Main Program (`main.f90`) 250

K.2 Interface Module (`liftinglineinterface.f90`) 250

K.3 Planform Class Module (`class_Planform.f90`) 262

K.4 Module of Setter Functions (`liftinglinesetters.f90`) 268

K.5 Solver Module (`liftinglinesolver.f90`) 276

K.6 Output Module (`liftinglineoutput.f90`) 284

K.7 Matrix Solver Module (`matrix.f90`) 292

K.8 Utilities Module (`utilities.f90`) 293

L PROOF OF EQUATION (5.2.5) 300

M TABULATED PROPERTIES OF THE NACA X410 FAMILY OF AIRFOILS 302

CURRICULUM VITAE 306

LIST OF TABLES

Table Page

2.1 Methods for Declaring Floating-Point Variables in Fortran 26

3.1 Aerodynamic coefficients for the NACA X412 family of airfoils in inviscid flow 52

3.2 Minimum induced drag results for untwisted wings with optimized planforms 53

3.3 Minimum induced drag results for geometric-twist-optimized rectangular wings..........55

3.4 Minimum induced drag results for aerodynamic-twist-optimized rectangular wings 58

3.5 Aerodynamic coefficients for the NACA X412 family of airfoils in viscous, incompressible flow at a Reynolds number of $Re = 10^6$ 59

3.6 Drag coefficients for initial rectangular wing and baseline elliptic wing in viscous flow. 60

3.7 Minimum total drag results for untwisted wings with optimized planforms...........60

3.8 Minimum total drag results for geometric-twist-optimized rectangular wings. 62

3.9 Drag components for geometric-twist-optimized rectangular wing with 11 control points. 62

3.10 Minimum total drag results for aerodynamic-twist-optimized rectangular wings. 63

3.11 Drag components for aerodynamic-twist-optimized rectangular wing with 11 control points. 63

4.1 Summary of limits for R_1 and R_2 92

4.2 Comparison of methods for calculating the wing lift slope of a finite elliptic wing 93

4.3 Comparison of resistance values R_1 from methods for calculating the wing lift slope of a finite elliptic wing 94

4.4 Comparison of resistance values R_2 from methods for calculating the wing lift slope of a finite elliptic wing 95

M.1 Airfoil Coefficient Data for the NACA 2410 Airfoil 302

M.2 Airfoil Coefficient Data for the NACA 4410 Airfoil 303

M.3 Airfoil Coefficient Data for the NACA 6410 Airfoil 304

M.4 Airfoil Coefficient Data for the NACA 8410 Airfoil 305

LIST OF FIGURES

Figure Page

2.1 An example of forward-mode AD using the function $f(x_1, x_2) = x_1 \sin(x_2)$ 12

2.2 An example of reverse-mode AD using the function $f(x_1, x_2) = x_1 \sin(x_2)$ 13

2.3 Header lines added to a Fortran module to include DNAD capabilities. 21

2.4 Example source code, compiler commands, and code execution (a) before and (b) after DNAD integration. Differences are highlighted in bold. 22

2.5 Example source code for customizing output of dual number objects. 23

2.6 Example of inconsistent data types used in a common block in different program units. 25

2.7 Example input file for the Fortran program given in Appendix B. 29

2.8 Closed-form and numerical solutions to the 1D transport equation. 31

2.9 Percent error in numerical solution at x = 0.75 for various grid resolutions. 31

2.10 Comparison of $\partial\phi / \partial u$ computed using DNAD and Eq. (2.5.8). 32

2.11 Comparison of $\partial\phi / \partial\Gamma$ computed using DNAD and Eq. (2.5.9). 32

2.12 Comparison of $\partial\phi / \partial C$ computed using DNAD and Eq. (2.5.10). 33

2.13 Comparison of errors in $\partial\phi / \partial u$ computed using DNAD and finite differencing. 34

2.14 Comparison of errors in $\partial\phi / \partial\Gamma$ computed using DNAD and finite differencing. 34

2.15 Comparison of errors in $\partial\phi / \partial C$ computed using DNAD and finite differencing. 35

2.16 Run-time benchmarks for the adpsolver code using DNAD and finite differencing. 36

2.17 Percent error in lift and drag coefficients for various grid densities relative to the solution computed using $n = 1280$ nodes per semispan. 41

2.18 Comparison of errors in $\partial C_L / \partial\alpha$ using DNAD and finite differencing relative to the DNAD solution computed using 1280 nodes per semispan. 42

2.19 Comparison of errors in $\partial C_{D_i} / \partial\alpha$ using DNAD and finite differencing relative to the DNAD solution computed using 1280 nodes per semispan. 42

2.20 Run-time benchmarks for MachUp using DNAD and finite differencing. 44

3.1 Optimized planforms for minimum induced drag. 54

3.2 Optimized geometric twist distributions for minimum induced drag. 56

3.3 Optimized aerodynamic twist distributions for minimum induced drag. 58

3.4 Optimized planforms for minimum total drag. 61

3.5 Optimized geometric twist distributions for minimum total drag. 62

3.6 Optimized aerodynamic twist distributions for minimum total drag. 64

3.7 Contours of (a) induced and (b) parasitic drag for finite wing sections. Results are included for rectangular wings with $A = 8$, $C_L = 0.5$, and optimized aerodynamic twist for variable *N*. 65

4.1 Discrete system of overlapping horseshoe vortices on a finite wing. 71

4.2 Velocity induced at point *P* by a semi-infinite vortex filament. 72

4.3 Comparison of lift slope calculation methods applied to a finite elliptic wing 96

4.4 Comparison of lift slope calculation methods applied to a finite rectangular wing 97

4.5 Comparison of lift slope calculation methods applied to a tapered wing with $R_T = 0.75$ 97

4.6 Comparison of spanwise lift coefficient distributions for elliptic wings. 99

4.7 Comparison of spanwise lift distributions for elliptic wings. 99

4.8 Comparison of spanwise lift distributions for rectangular wings. 100

4.9 Comparison of spanwise lift distributions for tapered wings with $R_T = 0.75$. 100

4.10 Sample calculations of the aspect ratio efficiency factor using Pralines and Panair. 101

5.1 Discrete system of side-by-side horseshoe vortices on a finite wing. 104

5.2 Velocity induced at point *P* by an arbitrary vortex segment $\overline{OR}$. 105

5.3 Comparison of wing lift slope calculations for finite elliptic wings. 108

5.4 Comparison of analytical and numerical calculations for wing lift slope of elliptic wings. 110

5.5 Comparison of spanwise lift distributions for elliptic wings. 111

5.6 Comparison of spanwise lift distributions for rectangular wings. 112

5.7 Comparison of spanwise lift distributions for tapered wings with $R_T = 0.75$. 112

5.8 Sample calculations of the aspect ratio efficiency factor using MachUp and Panair. 113

5.9 Comparison of numerical and experimental lift coefficients for the CAD-0 test. 114

5.10 Comparison of numerical and experimental lift coefficients for the CAD-2 test. 115

5.11 Comparison of numerical and experimental lift coefficients for the CAD-8 test. 115

5.12 Comparison of numerical and experimental lift coefficients for the VCCW-2 test. 117

5.13 Comparison of numerical and experimental lift coefficients for the VCCW-8 test. 117

5.14 Comparison of numerical and experimental lift coefficients for the VCCW-CU test. 118

5.15 Comparison of numerical and experimental lift coefficients for the VCCW-CD test. 118

5.16 Comparison of numerical and experimental lift coefficients for the VCCW-LW test. 119

5.17 Comparison of numerical and experimental drag coefficients for the CAD-0 test. 119

5.18 Comparison of numerical and experimental drag coefficients for the CAD-2 test. 120

5.19 Comparison of numerical and experimental drag coefficients for the CAD-8 test. 120

5.20 Comparison of numerical and experimental drag coefficients for the VCCW-2 test. 121

5.21 Comparison of numerical and experimental drag coefficients for the VCCW-8 test. 122

5.22 Comparison of numerical and experimental drag coefficients for the VCCW-CU test. 122

5.23 Comparison of numerical and experimental drag coefficients for the VCCW-CD test. 123

5.24 Comparison of numerical and experimental drag coefficients for the VCCW-LW test. 123

H.1 Lift distribution with a 20×20 mesh for an elliptic wing with $A=4$ and $\bar{t}_{\max}=4\%$ 234

H.2 Lift distributions as a function of average chord length with an 80×80 mesh for an elliptic wing with $A=4$ and $\bar{t}_{\max}=4\%$ 235

H.3 Mesh refinement and extrapolated results computed for an elliptic wing with $A=4$ and $\bar{t}_{\max}=4\%$ 237

H.4 Lift coefficient results computed for elliptic wings of different airfoil thicknesses with $A=4$ and extrapolated results for $\bar{t}_{\max}=0\%$ 237

H.5 Comparison of wing lift coefficients for wings with straight quarter-chord and straight mid-chord. 238

H.6 Comparison of spanwise lift coefficients for wings with straight quarter-chord and straight mid-chord. 238

LIST OF ACRONYMS

AD	Automatic Differentiation
AFRL	Air Force Research Laboratory
AVX	Advanced Vector Extensions
BFGS	Quasi-Newton optimization method of Broyden, Fletcher, Goldfarb, and Shanno
CCSA	Conservative Convex Separable Approximation
CFD	Computational Fluid Dynamics
DNAD	Dual Number Automatic Differentiation
DOE	Design of Experiments
JSON	JavaScript Object Notation
NACA	National Advisory Committee for Aeronautics
OO	Operator Overloading
RMS	Root-Mean-Square
SCT	Source Code Transformation
SIMD	Single Instruction, Multiple Data
SSE	Streaming SIMD Extensions
STL	Stereolithography file format
VCCW	Variable Camber Compliant Wing
VWT	Vertical Wind Tunnel

NOMENCLATURE

A	Aspect ratio of a finite wing
a	Wing lift slope
a_0	Section lift slope
b	Wingspan
C	Source term proportionality constant
c	Chord length
$\bar{c}$	Average chord length
C_D	Total wing drag coefficient ($C_{D_i} + C_{D_p}$)
c_d	Total section drag coefficient ($c_{d_i} + c_{d_p}$)
c_{d_0} , c_{d_1} , c_{d_2}	Constant, linear, and quadratic terms in section parasitic drag equation
C_{D_i}	Wing induced drag coefficient
c_{d_i}	Section induced drag coefficient
C_{D_p}	Wing parasitic drag coefficient
c_{d_p}	Section parasitic drag coefficient
C_L	Wing lift coefficient
c_l	Section lift coefficient
c_r	Section resultant aerodynamic force coefficient
e_A	Aspect ratio efficiency factor
e_s	Span efficiency factor
f, g	Arbitrary functions
$\mathbf{f}$	Vector function response of a fluid system in a coupled multiphysics problem
$\mathbf{H}$	Hessian matrix
h	Infinitesimal perturbation parameter
$\mathbf{I}$	Identity matrix
$\mathbf{J}$	Jacobian of a subsystem in a couple multiphysics problem

$\mathfrak{J}$	Global Jacobian of a coupled multiphysics problem
L	Wing lift force
l	Section lift force
m'	Additional apparent mass
n	Number of iterations in an iterative solution process; also the grid size in a discretized model
N	Number of independent variables in an arbitrary function; also the length of a gradient vector
$\mathcal{O}$	Order of accuracy operator
p	Arbitrary performance function
R_1 , R_2 , R_{tot}	Resistance values in a parallel circuit
R_e	Effective downwash resistance value
R_i	Induced downwash resistance value
R_T	Taper ratio
Re_c	Reynolds number based on chord length
S_w	Planform area
$\mathbf{s}$	Vector function response of a structural system in a coupled multiphysics problem
u, v, w	Cartesian velocity components
$\mathbf{u}_n$, $\mathbf{u}_a$	Unit vectors in the normal and axial directions, respectively
V_∞	Freestream velocity
w	Downwash; also intermediate variable used in automatic differentiation
w_e	Effective downwash
w_i	Induced downwash
x, y, z	Cartesian coordinates; also chordwise, spanwise, and normal coordinates, respectively
x	Arbitrary independent variable
$\bar{z}_{c_{\max}}$	Percent maximum camber
α	Section angle of attack
α_e	Effective angle of attack

α_i	Induced angle of attack
α_{L0}	Zero-lift angle of attack
α_{wing}	Wing geometric angle of attack
Γ	Diffusivity; also vortex strength
Δ	Finite perturbation of an arbitrary parameter (e.g. Δx)
$\Delta\alpha_{\text{geometric}}$	Change in local angle of attack due to geometric twist
$\Delta\alpha_{\text{aerodynamic}}$	Change in local angle of attack due to aerodynamic twist
ζ , η , θ	Change of variables
ρ_∞	Freestream density
ϕ	Scalar field quantity
$\Omega_{\max}$	Maximum twist value of a finite wing
∇	Gradient operator

1 INTRODUCTION

Aerodynamic shape optimization is an important step in the design process of modern aircraft. This step allows designers to tailor the aerodynamic features of an aircraft to meet a specific set of mission requirements in the most effective way possible. Optimization can be performed early in the design process to identify what features of a design are most important and toward the end of the design process to refine certain aspects of the design based on mission requirements. In many cases, high-fidelity Computational Fluid Dynamics (CFD) analyses are used as the objective functions in these optimization analyses (for example, see Refs. [1–4]). While CFD analyses provide significant insight into the specific flow characteristics of a design, they come at a relatively high cost due to the complex computational meshes and substantial computing resources required. This is especially true for optimization studies in which many cases must be run to evaluate performance changes with respect to design variables. As a result, CFD is not always the best solution for early-stage optimization when insights into trends and interactions between design parameters are more important than highly-accurate performance characteristics.

The computational challenge of using full CFD simulations for aerodynamic optimization are compounded when the application is a morphing wing. In an effort to improve aircraft efficiency through all phases of flight, several morphing-wing technologies are currently in development (for example, see Refs. [5–8]). In order to take full advantage of the benefits provided by morphing-wing technologies, performance characteristics and control derivatives for the wing in multiple morphed configurations must be readily available. Due to the large number of possible configurations for a morphing wing with even just a few degrees of freedom, the efficiency of aerodynamic and structural computations is a prodigious concern.

Several alternatives to full CFD simulations are available for wing optimization. The first practical method dates back to the early 20th century when Lanchester [9] and Prandtl [10,11] developed what is known today as classical lifting line theory. While this was the first mathematical model able to predict lift distributions over a 3D wing with reasonable accuracy, the original formulation is restricted to analyses of a single finite wing with a straight quarter-chord and moderate-to-high aspect ratio in an incompressible, inviscid flow. This method has been used to minimize induced drag [12,13] and maximize lift [14] of various wing planforms.

Low-fidelity methods such as the vortex lattice method and vortex panel method represent another alternative. These potential flow methods are widely used in industry and academia, and their development can be found in common aerodynamic textbooks. For a well-cited overview of these methods, see Katz and Plotkin [15]. These methods can be used to predict lift, induced drag, and pressure distributions over complex geometries but are generally unable to account for viscous effects and airfoil thickness.

A more recent development in aerodynamic analysis is a modern numerical lifting line method presented by Phillips and Snyder [16]. This method is similar to classical lifting line theory but uses the more general 3D vortex lifting law. The numerical formulation allows for the analysis of a system of interacting lifting surfaces with arbitrary camber, twist, sweep, and dihedral. It can also account for viscous effects through the use of 2D airfoil section coefficients. This model should not be confused with the vortex lattice method when a single element is used in the chordwise direction. Although the placement of the horseshoe vortices is similar in both methods, the fundamental equations are significantly different. For example, the vortex lattice method with a single chordwise element places a control point at the three-quarter-chord position and closes the formulation by requiring the normal velocity relative to the local camber line at the control point to be zero. On the other hand, the numerical lifting line method of Phillips and Snyder [16] uses the 2D section lift produced by the local airfoil section to calculate the 3D vortex lift of the finite wing. The latter approach is the numerical equivalent to the analytical approach taken by Prandtl [10,11].

It is worth noting that numerical methods for solving the lifting line equation from classical lifting line theory have been presented by McCormick [17] and Anderson et al. [18]. These methods show accurate results for wings below stall, but they neglect the downwash produced by the bound vorticity and are therefore limited in application to isolated straight wings without sweep or dihedral.

In this work, the numerical lifting line method of Phillips and Snyder [16] has been used as a foundation upon which a system is built for analyzing and optimizing morphing wings. We begin in Chapter 2 with a presentation of methods for computing derivatives and a demonstration of the process of automatic differentiation (AD) in a numerical lifting line algorithm. In Chapter 3 we discuss the development of an optimization framework for aerodynamic analysis and apply the derivative calculations from Chapter 2 to several wing shape optimization problems. We also demonstrate a method for using lifting line calculations to visualize the design space of morphing wings, which can assist in understanding and improving the wing shape optimization process and results. Chapter 4 presents a detailed overview of classical lifting line theory along with a discussion of its limitations relative to aspect ratio. We discuss the works of several researchers directed at improving predictions for low-aspect-ratio wings and present several new analytical developments that extend the validity of classical lifting line theory to wings of arbitrary aspect ratio. Chapter 5 presents a method for implementing the results of Chapter 4 in the numerical lifting line method of Phillips and Snyder [16], and concludes with a comparison of numerical results to experimental data for a low-aspect-ratio morphing wing.

2 DUAL NUMBER AUTOMATIC DIFFERENTIATION FOR WING SHAPE OPTIMIZATION

2.1 Introduction

The focus of this chapter is to present an implementation for accurately and efficiently computing derivatives within an aerodynamic analysis tool for the purpose of facilitating gradient-based wing shape optimization. Derivative calculations are an important consideration in any gradient-based optimization study. The gradient of the objective – a vector of partial derivatives of the objective with respect to the design variables – is used to determine the appropriate direction and magnitude for changes in the design variables in order to find a minimum (or maximum) in the objective. Constrained optimization algorithms usually require the calculation of the gradient for each constrained variable as well. The ability of gradient-based optimization algorithms to accurately and efficiently converge upon a local extremum is strongly influenced by the accuracy and efficiency of the derivative calculations upon which they rely. For many gradient-based optimization problems, the calculation of derivatives is often the costliest step in the optimization cycle (see Martins [19]), thus special care must be taken in obtaining these derivatives.

Fortunately, derivatives are important to many fields of application beyond optimization and therefore have a long history of research which can be drawn upon. The first formal presentations of derivatives are traditionally credited to Isaac Newton and Gottfried Leibniz in the late 17th century, though the fundamental concept of a derivative appeared much earlier (see Simmons [20]). Numerical analysis methods such as the finite difference (see Courant et al. [21]) and finite element (see Hrennikoff [22] and Courant [23]) methods arose from the need to solve sophisticated problems for which solutions could not be obtained through the methods of analytical calculus. The elliptic and hyperbolic partial differential equations used to describe boundary value and initial value problems are examples of these. For an examination of the history of the finite difference and finite element methods, see Thomée [24].

With the advent of modern computers came a new method for computing derivatives. Wengert [25] presented a technique for the “computation of numerical values of derivatives without developing analytical expressions for the derivatives.” Wengert’s method formed the foundation for what is now known as automatic (or algorithmic) differentiation (AD). Numerous publications exist on the subject, covering topics that range from theoretical research to software development to implementation in practical analysis applications. See, for example, Griewank and Walthers [26], Rall [27], and Corliss et al. [28]. A wide variety of open-source software tools for implementing AD methods in analysis software have been published, and numerous engineering software packages include AD capabilities. For a recent review of theoretical AD methods, see Martins and Hwang [29]. For a recent comparison of AD software tools for various languages, see Šrajer et al. [30].

In this chapter we apply a specific implementation of AD to MachUp, an open-source aerodynamics analysis tool based on the numerical lifting line method of Phillips and Snyder [16]. We do this to better facilitate accurate and efficient gradient calculations for wing shape optimization and other gradient-based optimization problems in which MachUp is used as the objective function. We begin with a general overview of the gradient calculation methods described above – namely, symbolic differentiation, finite differencing, and AD. We then proceed with a description of Dual Number Automatic Differentiation (DNAD), an open-source implementation of forward-mode AD in Fortran (see Yu and Blair [31] and Spall and Yu [32]). We discuss modifications to DNAD that were made to improve the flexibility and ease of integrating the tool with Fortran codes. A simple example is given illustrating the implementation process, and important considerations when implementing DNAD in existing Fortran codes are discussed. Finally, we present the process used to implement DNAD in MachUp.

2.2 Derivative Calculation Methods

Methods used for computing derivatives can be divided into three general categories as discussed in Sec. 2.1 – namely, symbolic differentiation, finite differencing, and AD. The three categories are presented here, and considerations for their use are discussed.

2.2.1 Symbolic Differentiation

Given our understanding of differential calculus, the most obvious method for computing derivatives is direct symbolic differentiation of the objective function. This involves application of the definition of the derivative as given in any elementary calculus textbook, namely

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{2.2.1}$$

where f is the objective function, x is the independent variable, and h is a perturbation parameter. Through symbolic differentiation, we obtain an explicit analytical equation for the derivative of a function which can be evaluated independent of the original function and is accurate to infinite precision. Algebraic manipulation of the derivative formula can be performed to express the formula in its simplest form. From a computational perspective, the “simplest form” is that which requires the fewest mathematical operations to evaluate. Thus, symbolic differentiation represents the most accurate and the most efficient method available for derivative computations.

For simple functions such as those found in elementary calculus textbooks, there are no disadvantages to using symbolic differentiation for derivative computations. On the other extreme, direct symbolic differentiation of practical engineering problems is rarely feasible. For example, consider a coupled multiphysics model in which fluid and structural analyses are performed iteratively to determine the response of a system. Results from one analysis are used to update the boundary conditions of the other analysis between iterations, and the number of iterations performed is controlled through the evaluation of residuals in the fluid and structural responses of the system. Expressing this function as a single formula that can be differentiated symbolically would require accumulating each mathematical operation in the process into a single expression. Since the number of iterations – and therefore the number of mathematical operations to be evaluated – is variable, the exact expression cannot be known *a priori*.

Even in the situation described above, it may still be possible to obtain symbolic derivatives of the objective function if the problem can be broken into smaller sub-functions for which symbolic derivatives are available. This approach uses the chain rule of differentiation to propagate derivative information through each sub-function evaluation. For each sub-function evaluation, the Jacobian matrix – which contains the partial derivatives of all function outputs with respect to each variable input – must be evaluated and stored. These Jacobian matrices can then be chained together to compute the gradient of the final objective function with respect to each original input.

To continue the coupled multiphysics example described above, let p be a single-value function that represents the performance of the coupled multiphysics system. Also, let $\mathbf{x} = \left(\mathbf{x}_f, \mathbf{x}_s\right)$ be the complete set of inputs required for the coupled multiphysics model, where $\mathbf{x}_f = \left(x_{f_1}, x_{f_2}, \ldots, x_{f_l}\right)$ is the set of l inputs required for the fluid analysis and $\mathbf{x}_s = \left(x_{s_1}, x_{s_2}, \ldots, x_{s_m}\right)$ is the set of m inputs required for the structural analysis. Then $p = p(\mathbf{x}) = p\left(\mathbf{x}_f, \mathbf{x}_s\right)$. The problem at hand is then to determine ∇p, where

$$\nabla p = \frac{\partial p}{\partial \mathbf{x}} = \left(\frac{\partial p}{\partial \mathbf{x_f}}, \frac{\partial p}{\partial \mathbf{x_s}} \right) = \left(\frac{\partial p}{\partial x_{f_1}}, \frac{\partial p}{\partial x_{f_2}}, \ldots, \frac{\partial p}{\partial x_{f_l}}, \frac{\partial p}{\partial x_{s_1}}, \frac{\partial p}{\partial x_{s_2}}, \ldots, \frac{\partial p}{\partial x_{s_m}} \right) \tag{2.2.2}$$

Now, let **f** and **s** be vector functions that represent the responses of the fluid and structural models, respectively. Then $\mathbf{f} = \mathbf{f}\left(\mathbf{x}_f, \mathbf{s}\right)$ and $\mathbf{s} = \mathbf{s}\left(\mathbf{x}_s, \mathbf{f}\right)$. If **f** is symbolically differentiable with respect to the input vectors $\mathbf{x}_f$ and **s**, then the Jacobian of **f**

$$\mathbf{J_f} = \begin{bmatrix} \dfrac{\partial \mathbf{f}}{\partial \mathbf{x}_f} & \dfrac{\partial \mathbf{f}}{\partial \mathbf{s}} \end{bmatrix} \tag{2.2.3}$$

can be determined analytically. Likewise, if **s** is symbolically differentiable with respect to the input vectors $\mathbf{x}_s$ and **f**, then the Jacobian of **s**

$$\mathbf{J_s} = \begin{bmatrix} \dfrac{\partial \mathbf{s}}{\partial \mathbf{x}_s} & \dfrac{\partial \mathbf{s}}{\partial \mathbf{f}} \end{bmatrix} \tag{2.2.4}$$

can also be determined analytically. Equations (2.2.3) and (2.2.4) can be used to assemble a global Jacobian $\mathfrak{J}$, which contains a row and column for each element in the $\mathbf{x}_f$, $\mathbf{x}_s$, $\mathbf{f}$, and $\mathbf{s}$ vectors. In general, this global Jacobian has the form

$$\mathfrak{J} = \begin{bmatrix} \dfrac{\partial \mathbf{x}_f}{\partial \mathbf{x}_f} & \dfrac{\partial \mathbf{x}_f}{\partial \mathbf{x}_s} & \dfrac{\partial \mathbf{x}_f}{\partial \mathbf{f}} & \dfrac{\partial \mathbf{x}_f}{\partial \mathbf{s}} \\ \dfrac{\partial \mathbf{x}_s}{\partial \mathbf{x}_f} & \dfrac{\partial \mathbf{x}_s}{\partial \mathbf{x}_s} & \dfrac{\partial \mathbf{x}_s}{\partial \mathbf{f}} & \dfrac{\partial \mathbf{x}_s}{\partial \mathbf{s}} \\ \dfrac{\partial \mathbf{f}}{\partial \mathbf{x}_f} & \dfrac{\partial \mathbf{f}}{\partial \mathbf{x}_s} & \dfrac{\partial \mathbf{f}}{\partial \mathbf{f}} & \dfrac{\partial \mathbf{f}}{\partial \mathbf{s}} \\ \dfrac{\partial \mathbf{s}}{\partial \mathbf{x}_f} & \dfrac{\partial \mathbf{s}}{\partial \mathbf{x}_s} & \dfrac{\partial \mathbf{s}}{\partial \mathbf{f}} & \dfrac{\partial \mathbf{s}}{\partial \mathbf{s}} \end{bmatrix} \tag{2.2.5}$$

where each term in Eq. (2.2.5) is a submatrix itself.

If we evaluate the fluid model first, we must provide an initial guess for the structural response, namely $\mathbf{s}_0$. Then $\mathbf{f}_1 = \mathbf{f}(\mathbf{x}_f, \mathbf{s}_0)$ and the global Jacobian for this step is

$$\mathfrak{J}_{\mathbf{f}_1} = \begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \dfrac{\partial \mathbf{f}_1}{\partial \mathbf{x}_f} & \mathbf{0} & \mathbf{0} & \dfrac{\partial \mathbf{f}_1}{\partial \mathbf{s}_0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix} \tag{2.2.6}$$

More generally, the global Jacobian for the *i*th evaluation of the fluid model can be written as

$$\mathfrak{J}_{\mathbf{f}_1} = \begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \dfrac{\partial \mathbf{f}_i}{\partial \mathbf{x}_f} & \mathbf{0} & \mathbf{0} & \dfrac{\partial \mathbf{f}_i}{\partial \mathbf{s}_{i-1}} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix} \tag{2.2.7}$$

Note that the third diagonal entry in Eqs. (2.2.6) and (2.2.7) is **0** as opposed to the identity matrix. This is because **f** is the dependent variable in the fluid model, so that the **f** in the numerators is one iteration ahead of the **f** in the denominators and we have $\partial \mathbf{f}_i / \partial \mathbf{f}_{i-1} = \mathbf{0}$ for the third diagonal entry.

The structural model is evaluated using results from the fluid model as input, so that the global Jacobian for the structural evaluation is

$$\mathfrak{J}_{\mathbf{s}_1} = \begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \dfrac{\partial \mathbf{s}_i}{\partial \mathbf{x}_s} & \dfrac{\partial \mathbf{s}_i}{\partial \mathbf{f}_i} & \mathbf{0} \end{bmatrix} \tag{2.2.8}$$

Similar to the global Jacobian for the fluid model, the fourth diagonal entry in Eq. (2.2.8) is **0** because **s** is the dependent variable here. Then the **s** in the numerator is one iteration ahead of the **s** in the denominator and we have $\partial \mathbf{s}_i / \partial \mathbf{s}_{i-1} = \mathbf{0}$ for the fourth diagonal entry.

The global Jacobian for the entire coupled solution is found by chaining the global Jacobians, Eqs. (2.2.7) and (2.2.8), over each iteration. A coupled solution evaluated for *n* iterations will have a global Jacobian given by

$$\mathfrak{J}_n = \prod_{i=1}^{n} \mathfrak{J}_{\mathbf{s}_i} \mathfrak{J}_{\mathbf{f}_i} \tag{2.2.9}$$

Equation (2.2.2) can now be written as

$$\nabla p = \left\lfloor \frac{\partial p}{\partial \mathbf{x}_f} \quad \frac{\partial p}{\partial \mathbf{x}_s} \quad \frac{\partial p}{\partial \mathbf{f}_n} \quad \frac{\partial p}{\partial \mathbf{s}_n} \right\rfloor \mathfrak{J}_n \tag{2.2.10}$$

While it would be tedious to symbolically evaluate Eq. (2.2.9) by hand, computer algebra systems can readily perform the evaluations and produce a symbolic formula for the evaluation of ∇p over *n* iterations. However, this symbolic formula may be of little use in practice as it is dependent on the number of iterations performed.

A more practical use of the methodology presented here is to evaluate the global Jacobians numerically during each step in the solution process and propagate a single aggregate Jacobian through each iteration. This approach results in numerical derivatives as opposed to symbolic expressions for the derivatives, but no additional assumptions or approximations are made in their evaluation.

An important distinction must here be made regarding the symbolic derivatives produced using this method compared to the derivatives that would be obtained from symbolically differentiating the governing mathematical model. While the derivatives discussed here are computed using symbolic derivatives of the individual analysis components, they are exact in terms of the numerical model and not the underlying mathematical model upon which the numerical algorithm is based. For further discussion on the implications of numerical modeling error in regard to derivative calculations, see Pakalapati et al. [33] and Sec. 2.4.4.

One potential benefit of using this method is that it can provide a measurement for the level of convergence in an iterative process such as the one described above. In our example, the last column of submatrices in the *i*th global Jacobian contains the derivatives of fluid and structural responses $\mathbf{f}_i$ and $\mathbf{s}_i$ with respect to the initial guess $\mathbf{s}_0$. As the solution approaches convergence, the responses should become independent of the initial guess, so that these derivatives should tend toward zero. Examining the magnitudes of these derivatives may offer insight into the level of convergence achieved at each iteration.

In summary, symbolic differentiation provides the most accurate and computationally efficient method of calculating derivatives. Unfortunately, symbolic derivatives are not readily available for most practical engineering problems. In some cases, the problem can be broken into smaller, symbolically differentiable components and the partial derivatives of the complete problem can be found through application of the chain rule of differentiation, either numerically or symbolically, without any loss in accuracy.

2.2.2 Finite Differencing

In 1928, Courant et al. [21] proposed a method for converting partial differential equations into simple algebraic functions by replacing the partial differentials with quotient approximations. Of particular interest to the authors were the boundary value and eigenvalue problems for elliptic differential equations and the initial value problem for hyperbolic and parabolic differential equations, for which there are few known analytical solutions and none of practical interest. Their development is general enough that it can readily be applied to other problems requiring the calculation of derivatives, and different formulations for the quotient approximations allow for variable levels of accuracy and complexity.

The simplest example of a finite difference approximation can be taken directly from the definition of the derivative given in Eq. (2.2.1). Removing the limit and replacing the infinitesimal perturbation parameter *h* with a finite step size Δx gives

$$\frac{\partial f}{\partial x} = \frac{f(x+\Delta x) - f(x)}{\Delta x} + \mathcal{O}(\Delta x) \tag{2.2.11}$$

which is the first-order-accurate forward difference formula. The first-order-accurate backward difference formula is similar but perturbed by $-\Delta x$, specifically

$$\frac{\partial f}{\partial x} = \frac{f(x) - f(x-\Delta x)}{\Delta x} + \mathcal{O}(\Delta x) \tag{2.2.12}$$

The second-order-accurate central difference formula is found by perturbing the functions in both positive and negative directions, which gives

$$\frac{\partial f}{\partial x} = \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} + \mathcal{O}(\Delta x^2) \tag{2.2.13}$$

This approach can be applied iteratively to obtain derivatives of higher order as well. For example, an approximation to the second derivative of *f* can be found by first finding approximations for $\partial f / \partial x$ from Eq. (2.2.13) centered at $x + \Delta x / 2$ and $x - \Delta x / 2$ using a perturbation of $\Delta x / 2$, and then using these derivative approximations as the perturbed function values in Eq. (2.2.13). This gives

$$\frac{\partial^2 f}{\partial x^2}(x) = \frac{\frac{\partial f}{\partial x}\left(x + \frac{\Delta x}{2}\right) - \frac{\partial f}{\partial x}\left(x - \frac{\Delta x}{2}\right)}{\Delta x} + \mathcal{O}(\Delta x^2) = \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{\Delta x^2} + \mathcal{O}(\Delta x^2) \tag{2.2.14}$$

Many other finite difference approximations are available. Other methods for deriving these formulas also exist. For example, Taylor series expansions of the function *f* can be used to obtain Eqs. (2.2.11)-(2.2.14) and many other finite difference formulas. In fact, the orders of accuracy listed in these equations were determined using Taylor series expansion. For more finite difference formulas and the methods used to derive them, see LeVeque [34].

While the finite difference method was originally developed specifically as a tool for solving partial differential equations, it can be used anytime the derivative of a function is needed and an approximation is acceptable. In general, the finite difference method can readily be applied to any numerical algorithm without the need to make internal modifications to the algorithm. This is done simply by running the algorithm multiple times at perturbed states and then applying the appropriate finite difference equation to the results. Because of this, the finite difference method is the most versatile derivative-approximation method available.

It is also the least accurate and least efficient method, however. All of the methods discussed here are limited in accuracy by machine precision and numerical modeling error; but the finite difference method is subject to truncation error as well. The finite difference method is also less efficient than other methods because full evaluations of the objective function are required for each perturbed function value in the finite difference formula. A function with N input variables would need to be evaluated $N+1$ times to obtain a first-order-accurate approximation of its gradient vector using Eqs. (2.2.11) or (2.2.12), or $2N$ times for a second-order-accurate approximation using Eq. (2.2.13).

2.2.3 Automatic Differentiation

As mentioned previously, AD is a method of numerically computing partial derivatives of an algorithm, accurate to machine precision, without the need to express the partial derivative symbolically. There are several different approaches to AD, but all stem from the same basic idea that every numerical algorithm can be broken down into a series of fundamental mathematical operations. If each mathematical operation in an algorithm is differentiable, then the entire algorithm can be differentiated through sequential applications of the chain rule of differentiation. This concept is quite similar to the example discussed in Sec. 2.2.1, but applied at a much finer level.

The different approaches to AD can be classified into two main groups. The first group, called forward-mode AD, performs derivative calculations in concert with the primary function evaluation. The chain rule of differentiation is used to propagate derivative calculations from one mathematical operation to the next, so that the derivatives of intermediate variables with respect to the independent variables are computed alongside the calculation of the intermediate variables themselves. This is the form of AD that was first developed by Wengert [25].

An example of forward-mode AD is given in Figure 2.1. The primary function $f(x_1, x_2) = x_1 \sin(x_2)$ is evaluated as shown, moving from top to bottom. An intermediate variable w_i is produced by each mathematical operation, and the derivatives of each intermediate variable with respect to the two independent variables are computed alongside the function values. Substitution using the definitions given, shown at the bottom of Figure 2.1, confirms that numerical results calculated using this process are consistent with the exact symbolic partial derivatives of the function.

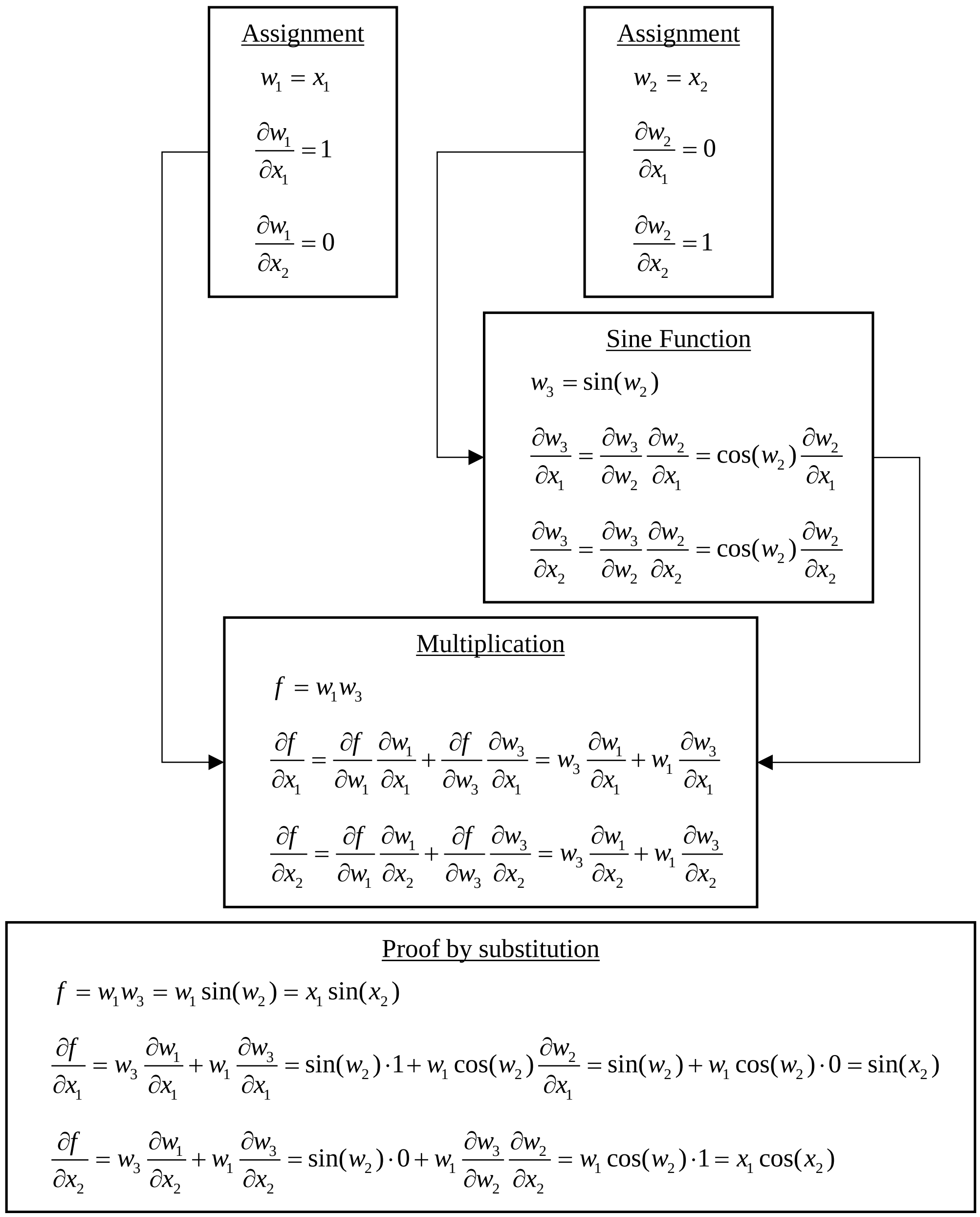


Figure 2.1 An example of forward-mode AD using the function $f(x_1, x_2) = x_1\sin(x_2)$.

An example of reverse-mode AD is given in Figure 2.2, again using the function $f(x_1, x_2) = x_1 \sin(x_2)$. The function evaluation proceeds as normal, but each partial derivative evaluation listed represents an addition to the computational graph. Once the function evaluation has been completed, the computational graph is traversed in reverse-order to produce the partial derivatives given in the right-hand box of Figure 2.2.

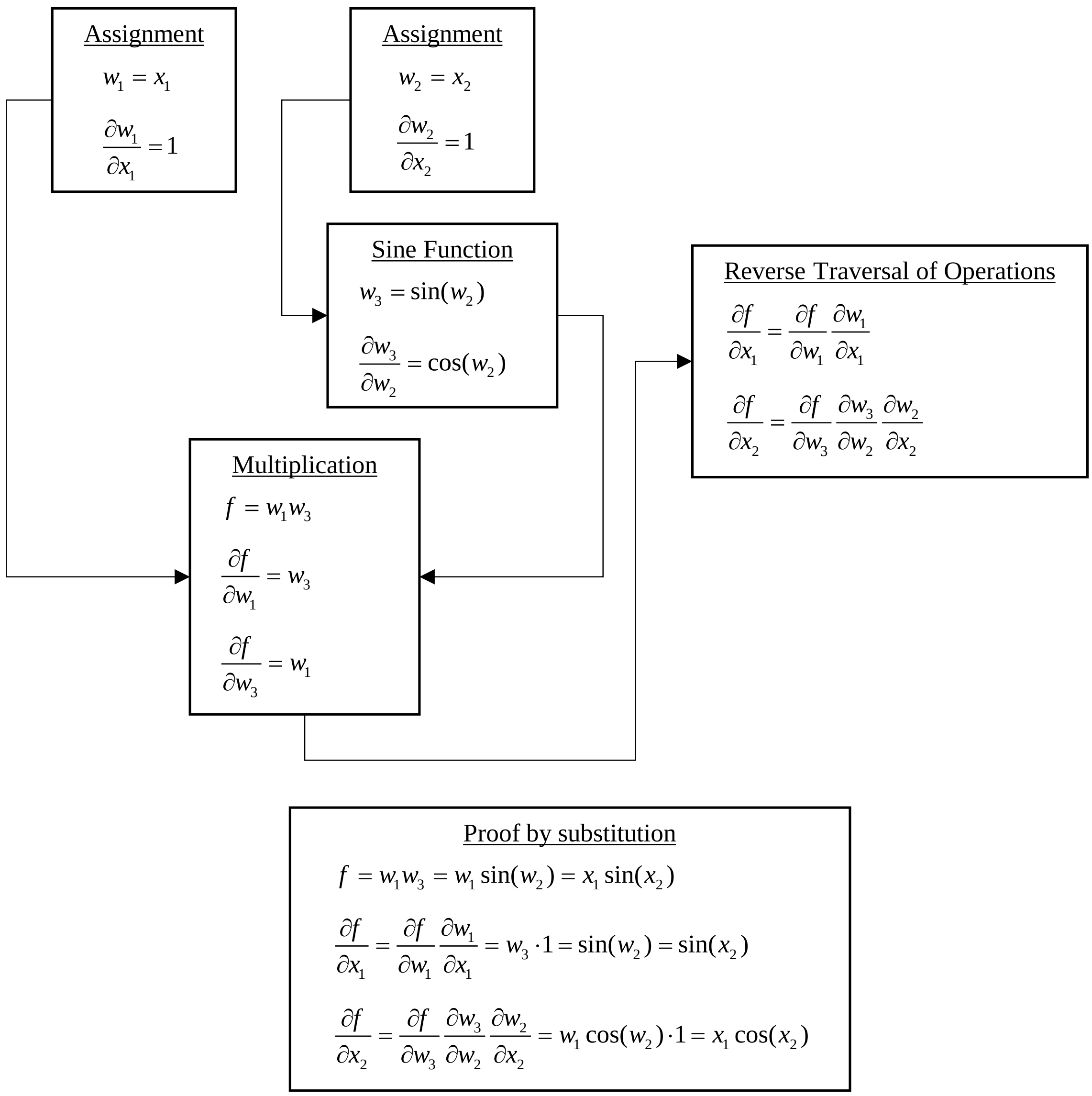


Figure 2.2 An example of reverse-mode AD using the function $f(x_1, x_2) = x_1 \sin(x_2)$.

Figures 2.1 and 2.2 were given to illustrate the overall processes of the respective methods, but many details are hidden so that a comprehensive comparison of the two methods cannot be made based on these examples alone. Fewer computations are required to arrive at the partial derivative solutions given in Figure 2.2 than were required to arrive at the same solution in Figure 2.1. This is only true, however, because the number of design variables (2) is larger than the number of output variables (1). In general, the complexity of the derivative calculations using forward-mode AD scales according to the number of design variables in the problem, while the complexity of the derivative calculations using reverse-mode AD scales according to the number of output variables for which gradients are needed.

Also not apparent from the examples illustrated in Figures 2.1 and 2.2 are the memory requirements for each method. Forward-mode AD algorithms must store the intermediate partial derivatives at each step in the solution process, so they require more memory than either symbolic differentiation or the finite difference method. Reverse-mode AD algorithms must store the entire computational graph, however, so that their memory requirements are considerably larger.

The effort required to implement the different modes in code is also not illustrated. In general, forward-mode AD is considered easier to implement and less intrusive to the original source code than reverse-mode AD. This is because, in reverse-mode AD, additional coding is required to build, manage, and traverse the computational graph. Additionally, each output variable for which derivatives are needed must be handled individually with existing reverse-mode AD methods, while forward-mode AD methods can be applied in such a way that a single version of the source code can compute derivatives of any output variable with respect to any input variable. This will be demonstrated later in this chapter. The reader is referred to the literature (see, for example, Refs. [26–28]) for more detailed discussions on the two AD methods.

2.2.4 Discussion

The purpose of the present chapter is to facilitate accurate and efficient derivative calculations within MachUp, as discussed in the introduction. Symbolic differentiation of the algorithm is infeasible due to the complexity of the nonlinear system of equations which form the basis of the numerical lifting line algorithm implemented by MachUp. Finite differencing introduces truncation error into the derivative calculations and is less efficient than AD methods. For these reasons, the author has selected AD for implementation in MachUp.

Of the available AD methods, reverse-mode AD is considered more efficient than forward-mode AD when the number of design variables is larger than the number of dependent variables. This is likely to be the case for any wing shape optimization problem in which MachUp is used to evaluate the objective function, but this advantage is expected to be small because of the limited number of inputs required to define even complex wing geometries in MachUp. The memory advantages of forward-mode AD are also not important to this effort because the memory requirements for most typical MachUp simulations are several orders of magnitude less than the available memory on most modern machines. The primary concerns in deciding between forward-mode and reverse-mode AD for the current work, then, are the amount of effort required for implementation in the existing source code and the flexibility of the resulting code in computing derivatives of various output variables with respect to various input variables. As discussed in Sec. 2.2.3, forward-mode AD holds advantages over reverse-mode AD in both respects.

2.3 Methods and Tools for Forward-Mode Automatic Differentiation

Two methods exist for implementing forward-mode AD in source code, namely source code transformation (SCT) and operator-overloading (OO). SCT is done by scanning the existing source code, identifying all floating-point variables and mathematical operations, adding additional variable declarations for the storage of partial derivative values, and adding additional lines of code for the computation of partial derivative values anytime a mathematical operation is performed. While to perform this process manually would be tedious at best, several algorithms have been developed to perform SCT automatically for a variety of programming languages. See, for example, Refs. [35–39]. SCT is possible in all programming languages, but tools for performing SCT are quite complex. Because the derivative computations appear explicitly in the differentiated source code, the resulting source code can be quite large when compared to the original undifferentiated code, and different source code is obtained depending on which input variables are made available for differentiation. As a result, several versions of the source code must be maintained in order to allow for any flexibility in the derivative calculations, which has negative implications regarding version control.

OO methods rely on the use of a custom data type with specifically-defined behavior for mathematical operations. The custom data type contains storage for a single floating-point variable and its derivative with respect to one or more independent variables. Any time a mathematical operator is applied to an instance of this custom data type, code is executed to evaluate both the mathematical operation and its analytical partial derivative(s) automatically. Not all programming languages support the polymorphic features required for implementing AD through OO, but most common languages used for scientific and engineering applications do (e.g. Fortran, Matlab, C++, Python, R, and Java).

As with SCT, several tools are available for implementing OO in existing source code, but implementations are quite varied in both functionality and ease-of-use. One common method for implementing OO, because of its overall simplicity, is the complex step method proposed by Lyness [40]. This method replaces all floating-point variables in an algorithm with the built-in complex data type that is standard in most modern programming languages. The independent variable with respect to which derivatives are desired is given an initial assignment that includes a small imaginary perturbation. Calculations for the algorithm then proceed under the normal rules of complex mathematics, assuming all operators in the algorithm are defined for complex numbers. Outputs of the algorithm will then contain the normal function value in the real part of the complex number and the first derivative of the function, multiplied by the small perturbation used, in the imaginary part of the complex number. To illustrate this method, consider a Taylor series expansion of the arbitrary function $f(x)$ using the perturbation *ih*,

$$f(x+ih) = f(x) + ih\frac{\partial f}{\partial x}(x) - \frac{h^2}{2!}\frac{\partial^2 f}{\partial x^2}(x) - \frac{ih^3}{3!}\frac{\partial^3 f}{\partial x^3}(x) + \ldots \tag{2.3.1}$$

Isolating the real part of Eq. (2.3.1) and solving for $f(x)$ gives

$$f(x) = \mathrm{Re}[f(x+ih)] + \frac{h^2}{2!}\frac{\partial^2 f}{\partial x^2}(x) - \ldots \tag{2.3.2}$$

so that $\mathrm{Re}[f(x+ih)]$ gives an approximation to the function $f(x)$ that is second-order accurate in *h*. Isolating the imaginary part of Eq. (2.3.1) and solving for the first derivative gives

$$\frac{\partial f}{\partial x} = \frac{\mathrm{Im}[f(x+ih)]}{h} + \frac{h^2}{3!}\frac{\partial^3 f}{\partial x^3}(x) + \ldots \tag{2.3.3}$$

so that $\mathrm{Im}[f(x+ih)]/h$ gives an approximation to the first derivative of *f* with respect to *x* that is also second-order accurate in *h*. On finite-precision machines, careful selection of the size of the perturbation parameter *h* can potentially eliminate the truncation errors in both Eqs. (2.3.2) and (2.3.3). See Martins et al.

[41] for a more detailed discussion on the complex step method and considerations for eliminating the truncation errors when using this method.

While the complex step method is relatively simple to implement and does not require the construction of a custom data type, it has several drawbacks that discourage its use over other OO methods. Martins et al. [41] showed a Fortran implementation of the complex step method to be significantly less efficient than even finite difference methods with only marginal improvement in accuracy. Also, the requirement for a perturbation step size introduces the potential for additional truncation error in the results that would not be incurred with a full OO implementation. Finally, the restriction to a complex number data type limits the number of independent variables with respect to which partial derivatives can be obtained to one per simulation.

Several full OO implementations of forward-mode AD have been developed for a variety of programming languages. Fortran implementations include ADF95 (see Straka [42]), AUTO_DERIV (see Stamatiadis et al. [43]), COSY INFINITY (see Berz et al. [44]), AD01 (see Pryce and Reid [45]), and DNAD (see Yu and Blair [31]). Of these, DNAD has been selected for use in this work. Several considerations were weighed in comparing these tools, including licensing, extensibility, level of complexity in the interface, and runtime performance. A thorough discussion of DNAD is given in the remainder of this chapter, including a presentation of dual number theory upon which DNAD is based, an examination of the algorithm, and the process of integrating DNAD with MachUp. For details on the other software packages listed, the reader is referred to Refs. [42–45].

2.4 Dual Number Automatic Differentiation

DNAD is essentially a derived data type that implements the rules of dual number theory programmatically, which can then be used in computational software to perform dual number calculations. In this section, we shall first present an overview of what dual number theory is. We shall then describe how this theory is implemented as a data type in DNAD. Finally, we shall discuss the process of integrating this derived data type into Fortran source code for automatic differentiation of an algorithm.

2.4.1 Dual Number Theory

The theory of dual numbers was first introduced by Clifford [46] for the purpose of describing biquaternions. It has since found application in several areas including the calculation of derivatives. Dual numbers extend the real number space by adjoining a nilpotent unit ε , which has the property

$$\varepsilon^n = \begin{cases} 1, & n = 1 \\ 0, & n \neq 1 \end{cases} \tag{2.4.1}$$

This is similar to how complex numbers extend the real number space using the imaginary unit *i*, but the differences in properties between the nilpotent unit and the imaginary unit result in distinct behavior between the two number sets. A dual number *d* can be written as

$$d = a + b\varepsilon \tag{2.4.2}$$

where *a* is the real component and *b* is the dual component of the dual number. A Taylor series expansion of the dual number perturbed about the real component by the dual unit ε gives

$$f(a+\varepsilon) = f(a) + f'(a)\varepsilon + f''(a)\frac{\varepsilon^2}{2!} + \ldots \tag{2.4.3}$$

or, applying the definition of the nilpotent unit given in Eq. (2.4.1),

$$f(a+\varepsilon) = f(a) + f'(a)\varepsilon \tag{2.4.4}$$

From Eq. (2.4.4) we see that the real component of $f(a+\varepsilon)$ gives the function value $f(a)$ and the dual component gives the first derivative of the function, $f'(a)$. Note that Eq. (2.4.4) is not a truncation of Eq. (2.4.3), so that the function value and its derivative obtained in this manner are exact.

Now consider two functions *f* and *g* represented as dual numbers $d_1 = f + f'\varepsilon$ and $d_2 = g + g'\varepsilon$. The product of these two dual numbers is

$$d_1 d_2 = (f + f'\varepsilon)(g + g'\varepsilon) = fg + (fg' + gf')\varepsilon + f'g'\varepsilon^2 = fg + (fg' + gf')\varepsilon \tag{2.4.5}$$

which gives the correct values for the function and its derivative as the real and dual components, respectively. Similarly, the quotient of these two dual numbers is

$$\frac{d_1}{d_2} = \frac{f + f'\varepsilon}{g + g'\varepsilon} \tag{2.4.6}$$

Multiplying top and bottom by the dual conjugate of the denominator, $g - g'\varepsilon$, gives

$$\frac{d_1}{d_2} = \frac{fg + (f'g - fg')\varepsilon - f'g'\varepsilon^2}{g^2 - g'\varepsilon^2} = \frac{f}{g} + \frac{f'g - fg'}{g^2}\varepsilon \tag{2.4.7}$$

which again gives the correct values for the function and its derivative as the real and dual components, respectively.

The dual number of a trigonometric function can be determined using another Taylor series expansion. For example, the function $f = \sin(x)$ can be expressed as a dual number as

$$d_3 = f(x+\varepsilon) = \sin(x) + \cos(x)x'\varepsilon - \sin(x)\frac{x''}{2!}\varepsilon^2 - \ldots \tag{2.4.8}$$

so that the real part – $\sin(x)$ – gives the original function and the dual part – $\cos(x)x'$ – gives the correct derivative. All other terms are zero by definition of the nilpotent unit.

This exercise can be repeated for all other continuously differentiable mathematical operations with consistent results. Additionally, the chain rule of differentiation allows multiple operations to be chained together to produce accurate expressions for the function and its derivative for functions of arbitrary complexity.

Retention of higher-order terms in ε allows for higher-order derivatives to be calculated in like manner to the first-order derivatives illustrated above. For an example of how dual numbers can be applied to compute second-order derivatives, see Fike and Alonso [47]. Only first-order derivatives will be treated in this work. However, future effort may extend DNAD to facilitate higher-order derivatives. This would facilitate accurate and efficient calculations of Hessian matrices, which can then be used to improve the performance of some gradient-based optimization algorithms.

2.4.2 The Dual Number Data Type

The dual number theory described in Sec. 2.4.1 has been implemented in Fortran by Yu and Blair [31] using a derived data type with operator overloading. The derived data type consists of a single floating-point variable that stores the primary function value, an array of floating-point values of arbitrary length containing any number of partial derivative values, and a series of overloaded operator functions defining the mechanics of dual number algebra. The DNAD source code is provided in Appendix A.

The source code given in Appendix A contains several modifications from the original version published by Yu and Blair. These modifications have been made in an effort to make the software more flexible and easier to use. In the original version, variables for precision and the length of the derivative vectors were defined explicitly in the source code. These definitions have been removed. Floating-point precision is now controlled through options specified at compile-time. For example, the DNAD module can be compiled for double-precision floating-point algebra using the `–fdefault-real-8` flag in the gfortran compiler or the `–r8` flag in the Intel Fortran compiler. Similarly, the length of the derivative vector can be specified using preprocessor directives. For example, a derivative vector length of 3 is defined by specifying `–Dndv=3` when invoking the preprocessor. By making these modifications, the floating-point precision and derivative vector length are now contained in the DNAD programming interface as opposed to being explicitly contained in the DNAD source code. Additionally, the source code has been reorganized and is now contained within a single source file. This is simply for convenience when compiling the module and linking it with Fortran algorithms. Note that the use of preprocessor directives requires that the preprocessor be invoked before the source code can be compiled. This is usually done when invoking the compiler by specifying a preprocessor flag (e.g. `–cpp` when using gfortran or `–fpp` when using the Intel Fortran compiler) or by capitalizing the file extension (e.g. *.F or *.F90 as opposed to *.f or *.f90).

Several additional operators have been added to the programming interface, including the intrinsic Fortran `tan`, `dtan`, `atan`, `atan2`, and `maxloc` functions. The `abs` operator overload has been modified to prevent excessive not-a-number (`NaN`) occurrences in the derivative values when the primary value is constant with respect to any of the independent variables. These changes make the DNAD module more versatile in its application to existing Fortran algorithms.

2.4.3 Integration with Fortran Algorithms

A streamlined process for integrating DNAD with Fortran algorithms has been developed. Most source files can be modified simply by adding the lines of code shown in Figure 2.3 to the beginning of the file. When other modifications to the source file are necessary, e.g. when logic must be added to handle input and/or output of the additional data contained in the dual number type, these modifications can also be contained inside compiler directives so that the normal behavior of the code can be retained when the DNAD module is not included in the compiling process.

```
#ifdef dnad
    use dnadmod
#define real type(dual)
#endif
```

Figure 2.3 Header lines added to a Fortran module to include DNAD capabilities.

In some situations, the programmer may need to exclude specific floating-point variables from conversion to dual numbers. For example, a named constant (usually defined with the `parameter` keyword) cannot be converted directly to a dual number because Fortran does not allow for implicit type conversion to derived data types. A simple trick has been successfully applied in these cases which takes advantage of the fact that the algorithms used to parse the preprocessor directives are case-sensitive, while the Fortran compiler itself is case-insensitive. Therefore, the commands shown in Figure 2.3 will only convert variables declared as `real` to `type(dual)`. Variables declared as uppercase `REAL` will remain unchanged.

As a simple example, consider the Fortran source code contained in the top box of Figure 2.4(a). This code prompts the user for a value for the radius (`r`) of a circle, which it then uses to compute the area of the circle. The compiler command used to generate an executable from this source code, using the open-source gfortran compiler, is given in the middle box of Figure 2.4(a). A single execution of the resulting executable is illustrated in the bottom box, where the user has specified a radius of 5.

In comparison, the modified source code to generate an executable capable of automatic differentiation using DNAD is shown in the top box of Figure 2.4(b). Only two changes have been made to the original code: the addition of four header lines from Figure 2.3 to include the DNAD module in the executable, and the change to uppercase `REAL` for the declaration of the parameter `pi`. The command needed to compile this code, given in the middle box of Figure 2.4(b), now specifies the DNAD source code file and defines the length of the derivative vectors. The `-Ddnad` option tells the preprocessor to execute the commands contained inside the `#ifdef` preprocessor directive at the top of the Circle program. Also note that the name of the source code file has been changed to use a capital "F" in the extension so that the preprocessor is automatically invoked. Execution of the resulting executable, shown in the bottom box of Figure 2.4(b), is again slightly different from that shown in the bottom box of Figure 2.4(a). In addition to specifying the radius of the circle, the user must also specify the partial derivative of the radius with respect to the

```
program Circle




    implicit none

    real, parameter :: pi=3.1416
    real :: r, a

    write(*,*) "Enter a radius: "
    read(*,*) r
    a = pi * r**2
    write(*,*) "Area = ", a
end program Circle
```

```
gfortran Circle.f90
```

```
./a.out
 Enter a radius
5
 Area =    78.5400009
```

a)

```
program Circle
#ifdef dnad
    use dnadmod
#define real type(dual)
#endif
    implicit none

    REAL, parameter :: pi=3.1416
    real :: r, a

    write(*,*) "Enter a radius: "
    read(*,*) r
    a = pi * r**2
    write(*,*) "Area = ", a
end program Circle
```

```
gfortran -Ddnad -Dndv=1 dnad.F90 Circle.F90
```

```
./a.out
 Enter a radius
5 1
 Area =    78.5400009    31.4159985
```

b)

Figure 2.4 Example source code, compiler commands, and code execution (a) before and (b) after DNAD integration. Differences are highlighted in bold.

independent variable of interest. In this case, the independent variable of interest is the radius, so that $\partial r / \partial r = 1$. In addition to outputting the area, the code has automatically output the derivative vector because a free-formatted output command was used. If different formatting is desired for the output, preprocessor directives can be used to customize the output. An example of this based on the CircleArea program is given in Figure 2.5.

Using the source code given in Figure 2.4(b) with the compiler command given in Figure 2.4(a) will produce an executable identical to the executable produced from the unmodified code. Thus, for this simple program, the same source code can now be used to generate both normal and differentiated versions of the software. It shall be shown later that this same result can be achieved for even complex algorithms. The simplification to a single source code for both normal and differentiated versions of the software is a significant advantage over other AD tools where separate source code repositories must be maintained.

```
program Circle
...
#ifndef dnad
    write(*,*) "Area = ", a
#else
    write(*, '(A, F12.7)') "Area = ", a%x
    write(*, '(A, F12.7)') "dA/dr = ", a%dx
#endif
end program Circle
```

Figure 2.5 Example source code for customizing output of dual number objects.

2.4.4 Special Considerations when Integrating DNAD with Existing Fortran Algorithms

Throughout the course of this work, several items have been identified that require special care in order to correctly implement DNAD with an existing Fortran code. Most of these items, with the exception of the first, relate to specific components of the Fortran programming language. Most of these components have been deprecated since the release of the Fortran 90 language standard and are generally discouraged from use in new source code. The abundance of legacy Fortran analysis codes that were written to language standards prior to Fortran 90, however, motivates the enumeration of these items.

The first item of note is the definition of "exact" when used to describe derivatives computed using AD. Most numerical algorithms use approximate methods – for example, interpolation methods, iterative root-finding methods, and finite volume and finite differencing approximations – to implement a mathematical model in code. Pakalapati et al. [33] noted that the derivatives computed using AD are derived directly from the numerical algorithm and have no knowledge of the underlying mathematical model from which the numerical algorithm was derived. As such, the derivatives are only exact in relation to the numerical model, and even then only to the limit of machine precision. The practice of referring to derivatives computed using AD as "exact" stems from the absence of truncation error in their calculations. However, it is important to keep in mind that derivative calculations computed using AD are still susceptible to mathematical modeling error, numerical modeling error, and round-off error.

The next item of concern for DNAD integration is the use of fixed-form source code. Prior to the Fortran 90 standard, all lines of Fortran code were required to conform to a set of fixed form rules. Fixed form source code is restricted to 72 characters per line of code, and the first six characters of each line are reserved. Continuation lines can be used to divide longer statements across multiple lines, but individual lines exceeding 72 characters in length are automatically truncated by the compiler. When preprocessor directives are used to modify fixed form source code (such as when `real` variable declarations are replaced with `type(dual)` – an increase of six characters – using the preprocessor directives listed in Figure 2.3), the resulting lines of code must conform to this same limit or be truncated.

With the publication of the Fortran 90 language standard, free form source code formatting was introduced which allows for up to 132 characters per line of code and removes the special reservation of the first six characters on each line. This added flexibility reduces the likelihood that modifying the source code through the preprocessor directives given in Figure 2.3 will cause inadvertent truncation of the source code. However, the possibility still exists. For fixed form source code, the code must be checked for any lines containing floating-point variable declarations that exceed 65 characters in width. For free form source code, these declaration lines cannot exceed 125 characters in width. Any lines exceeding these limits must be shortened or divided into multiple lines.

Another hurdle to the implementation of DNAD in Fortran codes is the use of the `common` statement to share information between program units. When data types of variables in a `common` statement are consistent between program units, the conversion of floating-point variables from `real` to `type(dual)` should work seamlessly. However, it is possible for the memory space assigned to a `common` statement to be declared as one data type in one program unit and a different data type in another program unit. This can lead to data misalignment when the data types of variables in the common block are changed. The code given in Figure 2.6 illustrates this problem. The same common block is declared in two separate subroutines but with inconsistent data types. The common block `cb`, when the two subroutines are compiled using a double-precision compiler flag (e.g. -fdefault-real-8 for the gfortran compiler or -r8 for the Intel Fortran compiler), will have a size of 8 bytes. However, if the DNAD header from Figure 2.3 is included and the subroutines are recompiled, the size of the common block `cb` in subroutine `sub1` will be $8(N+1)$, where N is the length of the derivative vector. This is incompatible with the size of the common block in subroutine `sub2` and will generate a compiler error. Even worse, if the name is omitted from the common block declarations so that the blank common block is used, the code will compile successfully but the character string `c` in subroutine `sub2` will only align with the first eight bytes of the dual number `x` in subroutine `sub1`. Any additional variables included in the common block will then be misaligned in the executable.

```fortran
subroutine sub1
    implicit none
    real :: x
    common /cb/ x ...
end subroutine sub1

subroutine sub2
    implicit none
    character(len=8) :: c
    common /cb/ c ...
end subroutine sub2
```

Figure 2.6 Example of inconsistent data types used in a common block in different program units.

A similar problem is encountered when codes use the `equivalence` statement to associate two variables within a program unit to the same memory location. Again, if a floating-point variable is associated with a non-floating-point variable in this manner, conversion to dual numbers will result in data misalignment and a potential bug in the resulting executable.

Both the `common` and `equivalence` statements have been deprecated in favor of modern replacements that, when used properly, ensure correct alignment of the underlying bit data and enforce consistency between program units automatically. New code development should use `modules` in place of `common` blocks and `transfer` statements in place of `equivalence` statements. When `common` and `equivalence` statements are encountered in legacy codes, the statements may need to be updated to their modern counterparts before successful differentiation of the codes can be achieved with the DNAD module.

Next we discuss concerns with using the Fortran `data` statement. The `data` statement is used to initialize variables to specific values at the beginning of a program unit. It is the recommended way for initializing large arrays of floating-point values prior to calculations and is therefore quite common in scientific and engineering applications. Unfortunately, custom constructors cannot be invoked implicitly within the Fortran programming language, so conversion of variables from `real` to `type(dual)` will also require a change to any `data` statements involving those variables. These changes can be wrapped inside preprocessor directives, but care must be taken to ensure consistency between initial values in both branches of the control.

The last concern to be discussed in this section is the multitude of ways in which floating-point variables can be declared in Fortran programs. Table 2.1 provides a summary of the different statements that can be used to declare floating-point variables. In this work the author has recommended using lowercase `real` for variables, uppercase `REAL` for constant parameters, and controlling precision through compiler options. However, all the options listed in Table 2.1 are valid statements and may be encountered in codes developed without DNAD in mind. Some codes may include several of the forms listed in Table 2.1. In order to correctly differentiate an algorithm via DNAD, all forms used in the code need to be accounted for in the conversion. This can be done by adding additional `#define` directives to the header lines given in Figure 2.3 or by modifying all floating-point variable declarations in the source code to use a single, consistent format.

Table 2.1 Methods for Declaring Floating-Point Variables in Fortran

Declaration	**Description**
`real`	Single-precision
`double precision`	Double-precision
`real*N`	*N*-byte floating-point value, where *N* is an integer
`real(kind=p)`	Floating-point value of kind *p*, where *p* is a kind value returned from the `selected_real_kind` function

2.5 Automatic Differentiation of a 1D Scalar Transport Equation Solver

In this section DNAD is applied to a 1D scalar transport equation solver implemented in Fortran. We begin with a description of the mathematical model for 1D transport of a scalar quantity which includes advection, diffusion, and production terms. We then describe the method used to implement this model numerically. Next, we discuss changes needed to obtain partial derivatives via the DNAD module. Finally, we present results computed using this model and comparisons with alternative methods for derivative calculations.

2.5.1 Mathematical Model

The Fortran algorithm is based on the mathematical model presented by Pakalapati et al. [33] and was written specifically for the purpose of demonstrating differentiation of a fluid dynamics algorithm using DNAD. The governing equation,

$$u\frac{\partial \phi}{\partial x} = \Gamma \frac{\partial^2 \phi}{\partial x^2} + C\phi \tag{2.5.1}$$

describes the 1D transport of a scalar field quantity ϕ due to advection, diffusion, and production, where *x* is the spatial coordinate, *u* is the velocity, Γ is the diffusivity, and *C* is a proportionality constant such that the source term is directly proportional to the local concentration ϕ.

Two boundary conditions are required to close Eq. (2.5.1). For certain boundary conditions, a closed-form solution can be obtained which will aid in evaluating the accuracy of derivatives computed using DNAD. A closed-form solution to Eq. (2.5.1) with boundary conditions $\phi(0)=1$ and $\phi(1)=0$ is given by Pakalapati et al. [33] as

$$\phi(x)=\frac{e^{\lambda^-}e^{\lambda^+ x}-e^{\lambda^+}e^{\lambda^- x}}{e^{\lambda^-}-e^{\lambda^+}} \tag{2.5.2}$$

where

$$\lambda^{\pm}=\frac{u\pm\sqrt{u^2-4\Gamma C}}{2\Gamma} \tag{2.5.3}$$

when $u^2-4\Gamma C>0$.

We desire the derivatives of the scalar field function ϕ with respect to the *x* coordinates and the three proportionality constants *u*, Γ, and *C*. Symbolic differentiation of Eq. (2.5.2) with respect to $\lambda^{\pm}$ gives

$$\frac{\partial\phi}{\partial\lambda^{\pm}}=\pm\frac{e^{\lambda^+}e^{\lambda^-}\left(e^{\lambda^+ x}-e^{\lambda^- x}\right)+xe^{\lambda^{\mp}}e^{\lambda^{\pm}x}\left(e^{\lambda^-}-e^{\lambda^+}\right)}{\left(e^{\lambda^-}-e^{\lambda^+}\right)^2} \tag{2.5.4}$$

Differentiation of $\lambda^{\pm}$ with respect to the three proportionality constants gives

$$\frac{\partial\lambda^{\pm}}{\partial u}=\pm\frac{\lambda^{\pm}}{\sqrt{u^2-4\Gamma C}} \tag{2.5.5}$$

$$\frac{\partial\lambda^{\pm}}{\partial\Gamma}=\pm\frac{C-u\lambda^{\pm}}{\Gamma\sqrt{u^2-4\Gamma C}} \tag{2.5.6}$$

$$\frac{\partial\lambda^{\pm}}{\partial C}=\mp\frac{1}{\sqrt{u^2-4\Gamma C}} \tag{2.5.7}$$

From the chain rule of differentiation, the partial derivatives of ϕ with respect to the three proportionality constants are

$$\frac{\partial\phi}{\partial u}=\frac{\partial\phi}{\partial\lambda^+}\frac{\partial\lambda^+}{\partial u}+\frac{\partial\phi}{\partial\lambda^-}\frac{\partial\lambda^-}{\partial u} \tag{2.5.8}$$

$$\frac{\partial\phi}{\partial\Gamma}=\frac{\partial\phi}{\partial\lambda^{+}}\frac{\partial\lambda^{+}}{\partial\Gamma}+\frac{\partial\phi}{\partial\lambda^{-}}\frac{\partial\lambda^{-}}{\partial\Gamma} \quad (2.5.9)$$

$$\frac{\partial\phi}{\partial C}=\frac{\partial\phi}{\partial\lambda^{+}}\frac{\partial\lambda^{+}}{\partial C}+\frac{\partial\phi}{\partial\lambda^{-}}\frac{\partial\lambda^{-}}{\partial C} \quad (2.5.10)$$

2.5.2 Numerical Model

The 1D transport problem can be solved numerically by discretizing the domain and applying finite difference approximations such as those discussed in Sec. 2.2.2 to the derivative terms in Eq. (2.5.1). For example, we can use the first-order-accurate backward difference formula given by Eq. (2.2.12) to approximate $\partial\phi/\partial x$ in the advection term and the second-order-accurate central difference formula given by Eq. (2.2.14) to approximate $\partial^2\phi/\partial x^2$ in the diffusion term. This gives

$$a_W\phi_{i-1}+a_P\phi_i+a_E\phi_{i+1}=0 \quad (2.5.11)$$

where

$$a_W=\begin{cases}\dfrac{\Gamma}{\Delta x^2}-\dfrac{u}{2\Delta x} & u\geq 0\\ \dfrac{\Gamma}{\Delta x^2} & u<0\end{cases} \quad (2.5.12)$$

$$a_E=\begin{cases}\dfrac{\Gamma}{\Delta x^2} & u\geq 0\\ \dfrac{\Gamma}{\Delta x^2}+\dfrac{u}{2\Delta x} & u<0\end{cases} \quad (2.5.13)$$

$$a_P=-\left(a_W+a_E+C\right) \quad (2.5.14)$$

Equation (2.5.11) is a tridiagonal system of linear equations that can be solved implicitly using, for example, the Thomas Algorithm (see Thomas [48]).

A basic Fortran implementation of the solution procedure described above is given in Appendix B. This implementation uses a card-style input format for specifying the boundary conditions and proportionality constants, as well as the number of grid points to use in the domain discretization. An example input file is given in Figure 2.7. Upon completion, the program generates a comma-delimited ASCII text file containing the solution of ϕ at each discretized *x*-coordinate. For comparison, the closed-form solution given by Eq. (2.5.2) has also been included in the Fortran code and can be solved by changing the solver method to

```
*job, name=adpsolver_example
*bcs, phi0=1.0, phi1=0.0
*props, u=1.0, gamma=0.1, c=-1.0
*grid, npts=41
*solver, method=implicit
```

Figure 2.7 Example input file for the Fortran program given in Appendix B.

`analytical`. Closed-form partial derivatives based on Eqs. (2.5.8)-(2.5.10) can then be calculated by adding a `derivatives` parameter to the `solver` card and setting its value to `yes`. Note that the closed-form solution is only valid when the boundary conditions are $\phi(0)=1$ and $\phi(1)=0$, and the `derivatives` parameter is only used when the solver method is set to `analytical`.

2.5.3 Automatic Differentiation of the 1D Scalar Transport Problem using DNAD

The process outlined in Sec. 2.4.3 was used to differentiate the 1D scalar transport problem via the DNAD module. Code modifications are summarized in Appendix C. No modifications to the `main` program unit were required because no floating-point variables are declared in this unit. Conversion of floating-point variables from `real` to `type(dual)` in the `adpsolver` and `adpio` modules was facilitated using the preprocessor directives given in Figure 2.3. No other modifications to the `adpsolver` module were needed. Several additional modifications were needed in the `adpio` module, however, in order to properly handle options for requesting derivatives in the input file and to include the derivative calculations in the output file.

First, three new static variables were declared in the `adpio` module to store information relative to the partial derivatives requested by the user. This is necessary in order to ensure that the derivative vectors are not overrun by too many partial derivative requests and to ensure that the derivative information is properly written to the output file.

Next, an additional case was added to the `parseCard` function so that a `*dnad` card can be processed, and an additional function (`setDNADField`) was added to process this card. The `*dnad` card is used to enumerate specific inputs with respect to which partial derivatives are to be computed. Available options supported by the `setDNADField` function include derivatives with respect to u, Γ, and C. An example card requesting derivatives with respect to all three of these variables is

```
*dnad, dv=u, dv=gamma, dv=c
```

The boundary conditions $\phi(0)$ and $\phi(1)$ are the only other floating-point inputs to this code. Support for derivatives with respect to these variables could easily be added with just a few more lines of code to the `setDNADField` function.

A minor change was made to the type declaration of the `stringToReal` function (line 342 in the original code). The `real` keyword was changed to all-caps (`REAL`) so that the return type of this function would not be converted to `type(dual)` by the preprocessor. This ensures that only a single floating-point value is parsed when processing the text for a parameter value, and only the primary variable value (not the partial derivative values) is modified by the function assignment.

The last set of changes made to the `adpio` module was to add an alternative version of the `writeData` function that includes statements for writing out the results with automatic derivatives. The `writeData` function that is included when the code is compiled is controlled by placing the two functions inside an `#ifndef dnad` preprocessor directive. This complete separation of code for the undifferentiated and differentiated versions of the algorithm allows for flexibility in how the derivative information is output to the results file without affecting the original format.

Preprocessor directives were used around each of the modifications described above, so that inclusion of the DNAD module in the compiled executable can be controlled entirely through preprocessor options specified at compile-time. Adding the `–Ddnad` and `–Dndv=<#>` options to the compiler command will activate the DNAD module and enable automatic differentiation within the resulting executable, while omitting these options from the compiler command will produce the same executable as would be generated by the original code contained in Appendix B.

2.5.4 Results and Discussion

Figure 2.8 compares numerical results of the 1D scalar transport problem described above using three different grid resolutions to the closed-form solution given by Eq. (2.5.2). For $x < 0.5$ the closed-form curve is relatively linear and the numerical results are in good agreement. For larger x values, the curve becomes more nonlinear and the accuracy of the numerical results is reduced. However, refinement of the grid demonstrates convergence of the numerical results toward the closed-form solution. Figure 2.9 shows the

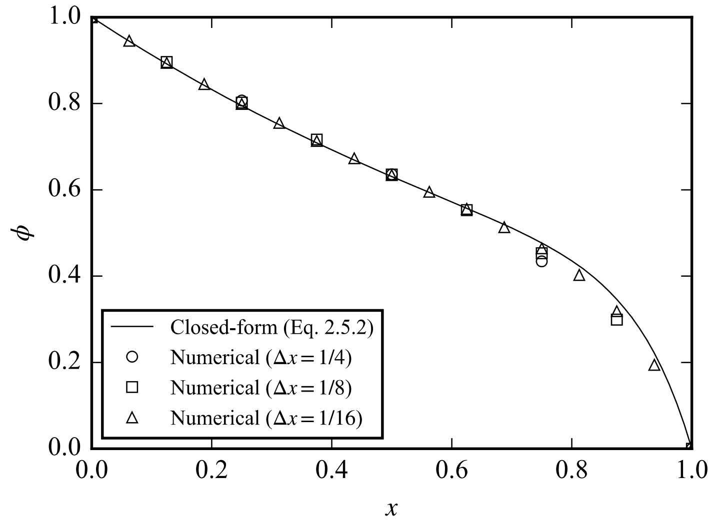


Figure 2.8 Closed-form and numerical solutions to the 1D transport equation.

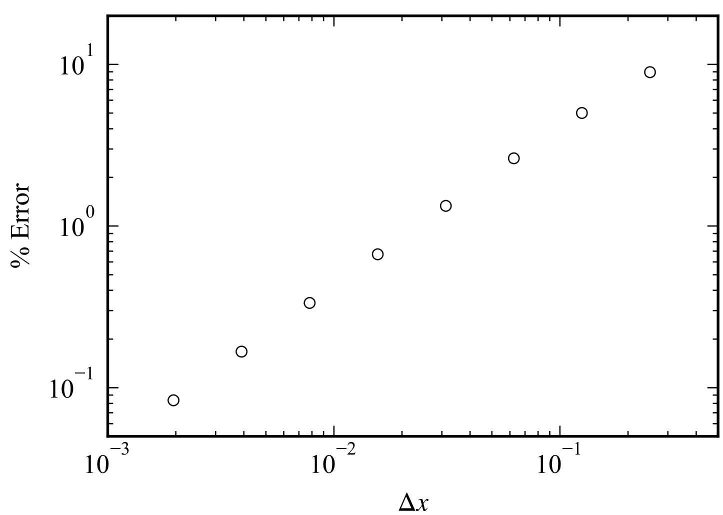


Figure 2.9 Percent error in numerical solution at *x* = 0.75 for various grid resolutions.

percent error of the solution at $x = 0.75$ for several grid sizes on a log-log plot. The slope of the data indicates the algorithm has a first-order convergence rate. This convergence rate corresponds with the order of the backward difference formula – see Eq. (2.2.12) – that was used in developing the numerical model.

Figures 2.10-2.12 provide a comparison of partial derivative results computed using both the DNAD module and the closed-form solutions given by Eqs. (2.5.8)-(2.5.10). Also included are DNAD-computed derivatives of Eq. (2.5.2). The DNAD-computed derivatives of Eq. (2.5.2) match Eqs. (2.5.8)-(2.5.10) to machine precision, confirming that these results are the numerical equivalent of symbolic differentiation of the governing equation. The DNAD-computed derivatives of the numerical model are not as accurate, and the error is shown to be inversely proportional to the step size Δx used in the analysis.

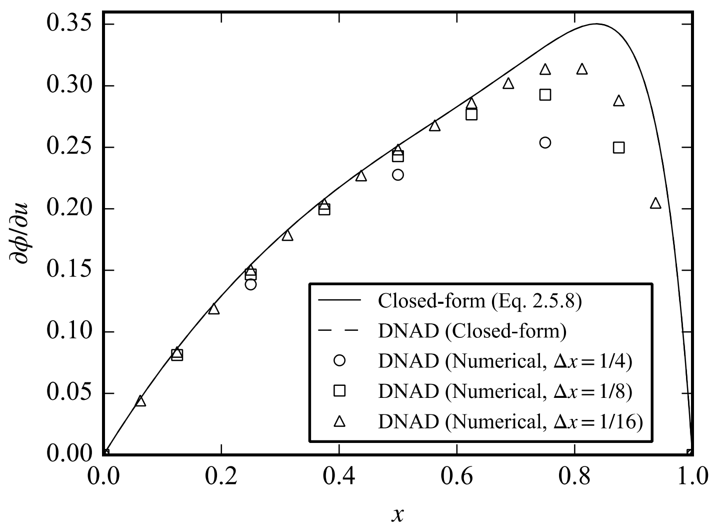


Figure 2.10 Comparison of $\partial\phi / \partial u$ computed using DNAD and Eq. (2.5.8).

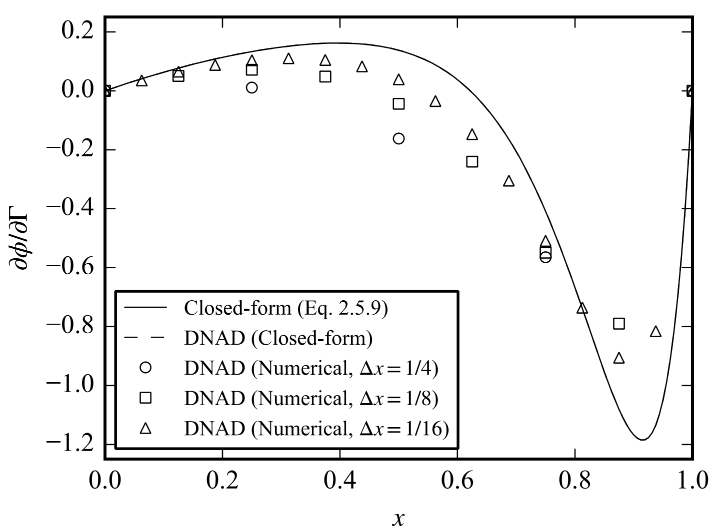


Figure 2.11 Comparison of $\partial\phi / \partial\Gamma$ computed using DNAD and Eq. (2.5.9).

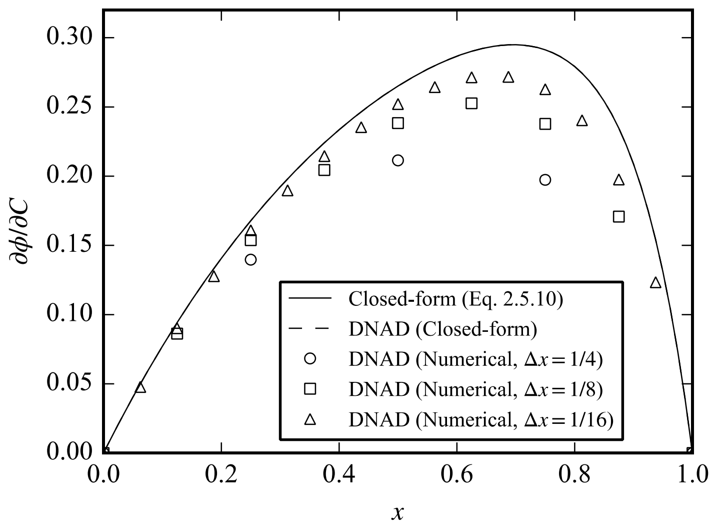


Figure 2.12 Comparison of $\partial\phi / \partial C$ computed using DNAD and Eq. (2.5.10).

Figures 2.13-2.15 compare the root-mean-square (RMS) errors of the DNAD calculations with RMS errors in corresponding solutions computed using finite difference methods. In general, all of the methods exhibit a first-order convergence rate, and for coarse grids ($\Delta x > 10^{-2}$) there is little difference in the RMS error values between the different methods. There are lower limits in $\Delta x \Delta x$, however, below which the finite difference methods no longer converge or even begin to diverge. These limits depend on the finite differencing method used and the finite differencing step size (e.g. Δu, $\Delta \Gamma$, or ΔC) used. DNAD derivatives do not have this limit and will continue toward the exact solution with decreasing Δx at the same approximate convergence rate as ϕ until reaching the accuracy of machine precision.

Some of the finite difference solutions shown in Figures 2.14 and 2.15 have smaller RMS errors than the corresponding DNAD solutions for a given Δx. In each of these cases, the truncation error due to discretization of the governing equation is in the opposite direction as the truncation error due to discretization of the partial derivative, so that the two error sources partially cancel, lowering the overall RMS error. The reductions in error are small and only occur at particular combinations of and the finite differencing step size ($\Delta \Gamma$ or ΔC), so that taking advantage of this situation during a typical analysis would be impractical.

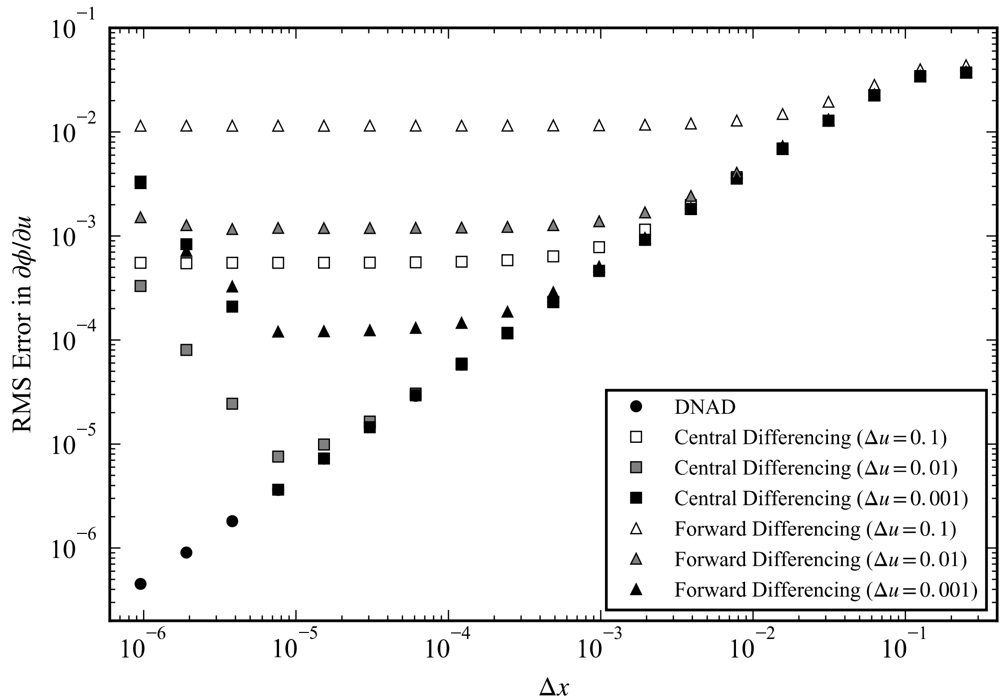


Figure 2.13 Comparison of errors in $\partial\phi/\partial u$ computed using DNAD and finite differencing.

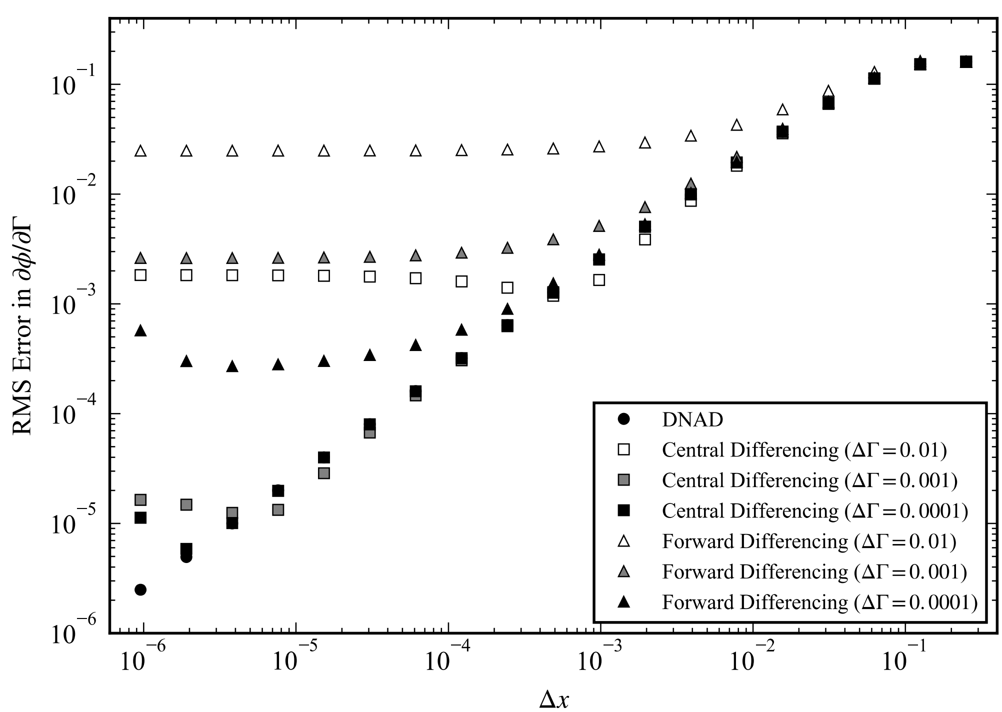


Figure 2.14 Comparison of errors in $\partial\phi/\partial\Gamma$ computed using DNAD and finite differencing.

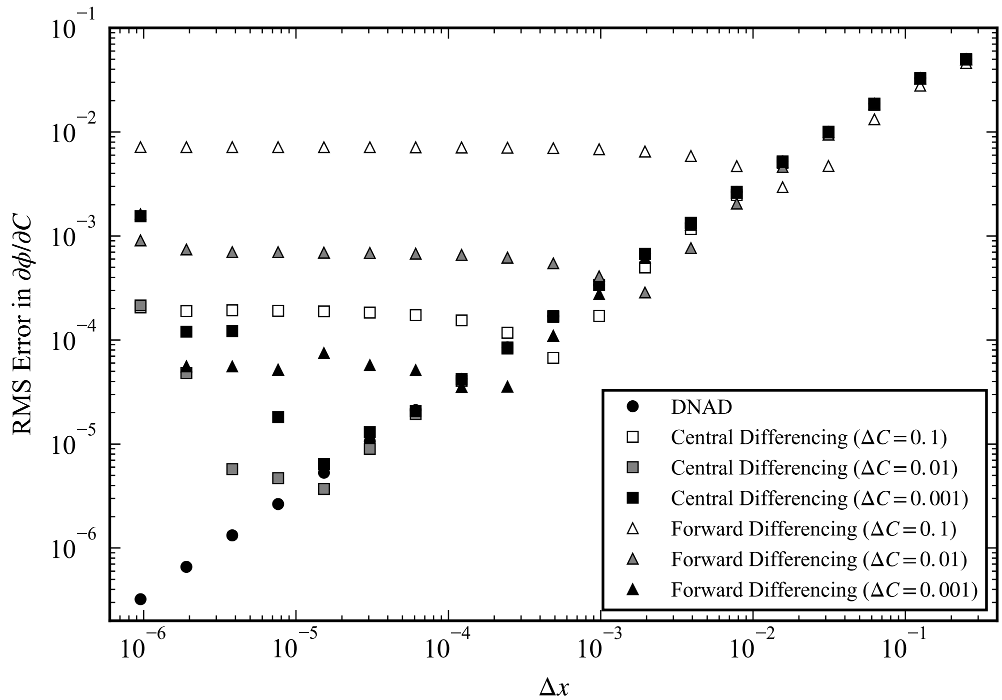


Figure 2.15 Comparison of errors in $\partial\phi / \partial C$ computed using DNAD and finite differencing.

Time benchmarks for the adpsolver code were collected using a standard desktop computer running Windows 10 Enterprise 64-bit with a third-generation Intel Core i7-3770 3.4 GHz processor, 16 GB of 1600 MHz DDR3 RAM, and a 1 TB 7200 RPM Serial ATA internal hard drive. The adpsolver code was compiled using version 6.3.0 of the GNU Fortran compiler.

The example input file given in Figure 2.7 was used as a baseline input file for all of the timed analyses, except that the number of grid points was changed and a `*dnad` card was added for the benchmarking analyses of the differentiated code. Figure 2.16 shows timing results for the differentiated code normalized by the time required to execute the undifferentiated code. Also included for comparison are the normalized times that would be required to compute the same derivatives using first-order forward differencing (Eq. (2.2.11)) and second-order central differencing (Eq. (2.2.13)).

The results presented here demonstrate that using DNAD can improve the accuracy and run-time efficiency of derivative calculations over finite differencing methods. The run-time cost using the DNAD

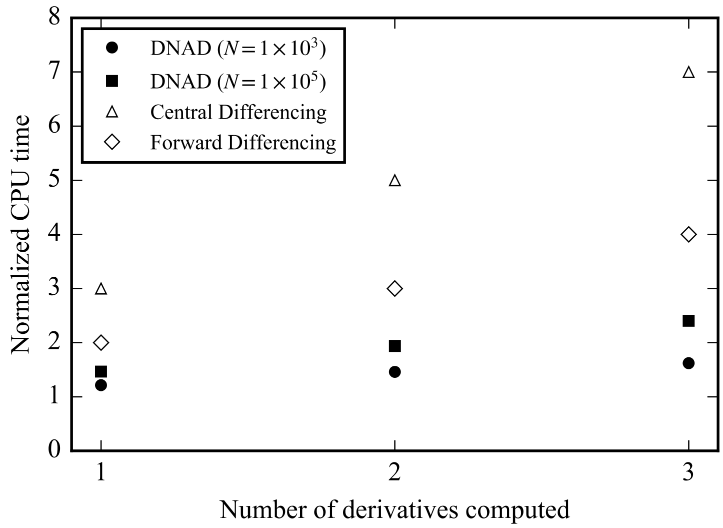


Figure 2.16 Run-time benchmarks for the adpsolver code using DNAD and finite differencing.

module is dependent on mesh size, though this dependency is small. For each of the data sets shown in Figure 2.16, the run-time cost increases almost linearly with the number of derivatives computed. The additional cost of each derivative computed using DNAD is less than half the cost of the primary function if a mesh size of $N = 10^5$ is used and less than one-fourth the cost of the primary function if a mesh size of $N = 10^3$ is used. Source code changes required to differentiate the adpsolver code using DNAD were minimal and mostly confined to I/O operations. The effort required to implement these changes was comparable to the effort required to find the appropriate perturbation step sizes for computing the derivatives using finite differencing. This is a relatively simple code with fewer than one thousand lines of code, however, so these same conclusions may not apply to more complicated software with larger code bases.

2.6 Automatic Differentiation of MachUp

In this section we discuss automatic differentiation of the numerical lifting line code MachUp using DNAD. This code was written without DNAD in mind, but it presents an ideal application for DNAD since relatively few inputs are required for an analysis and the results generated by MachUp are of value to wing design problems. The MachUp source code is developed and maintained by the USU Aero Lab[*] and is

[*] http://aero.go.usu.edu

available for download from github[*]. The entire source code is not included in this work, but Appendix D contains the modifications that were made for this effort. We begin this section with a discussion of the MachUp software. Next we discuss in detail the modifications that were made to the MachUp source code in order to enable automatic differentiation via DNAD. Finally, we demonstrate the accuracy and efficiency of the resulting derivative calculations through a limited set of results and performance measurements.

2.6.1 The MachUp Numerical Lifting Line Solver

MachUp is an open-source aerodynamics analysis tool for evaluating the performance characteristics of systems of finite wings. It is written in modern, object-oriented Fortran and interfaces with web applications and other software packages through human-readable input and output files. The core algorithm in MachUp is based on the numerical lifting line method of Phillips and Snyder [16]. Although rooted in potential flow theory, this method has the ability to include viscous effects in the analysis. It can be used to model multi-wing systems and wings with sweep and dihedral. The algorithm equates the 3D vortex theory of lift to the 2D section lift at discrete control points located along the quarter-chord of a wing to solve for the bound vorticity strength at these locations. This development produces a nonlinear system of equations that must be solved iteratively. A linear approximation is used as an initial guess, and Newton's method is applied to update the solution until changes in the calculated vortex strengths between iterations fall below a user-specified threshold. The algorithm has been used for a number of wing design studies, including studies of propeller-wing interactions [49] and ground effect [50].

The MachUp source code is organized into several modules which contain the data structures for different components of the analysis. For example, a `plane_t` object stores all of the data and methods needed to define and solve the system of horseshoe vortices for the complete model, including an array of `wing_t` objects that define the geometry for each individual lifting surface. Further, each `wing_t` object contains an array of `section_t` objects that each define the 2D airfoil section properties at one control point on the wing.

Input to MachUp is handled through ASCII text files that use the JavaScript Object Notation (JSON) format for data structures. This data-interchange format is often used in web applications and can easily

---

[*] https://github.com/usuaero/machup

interface with other common engineering tools such as Matlab and Python. In addition, these input files are human-readable and can be created and edited in any standard text editor. The JSON data structures are handled within MachUp through an open-source JSON parsing tool available on Github* and also included within the MachUp source code. A sample MachUp input file is shown in Appendix E. The input file defines a single wing with an aspect ratio of 8; an elliptic chord distribution; no sweep, dihedral, or geometric twist; and a uniform cross-section with properties corresponding approximately to those of a NACA 2412 airfoil in inviscid, incompressible flow. The airfoil properties and profile are defined in separate files that reside within the specified airfoil database folder. An example of these property and profile files for the NACA 2412 airfoil is given in Appendix E.

Commands to be executed are specified under the "`run`" record in the JSON input file. The example file in Appendix E lists multiple commands, all of which can be performed with a single execution of the software. The first command, "`targetcl`", solves the complete lifting line algorithm multiple times using Newton's method to adjust the angle of attack until the specified lift coefficient is achieved. The "`forces`" command generates a JSON output file that contains the aerodynamic performance coefficients calculated by the lifting line algorithm. The "`distributions`" command generates a comma-separated values (CSV) file that contains aerodynamic results for each wing section in the model. Finally, the "`stl`" command generates a stereolithography (STL) file that contains a geometric representation of the model for visualization. Note that any of the "run" commands listed can be temporarily disabled by setting the "`run`" parameter for that command to 0.

Viscous effects can be included in a MachUp analysis by specifying viscous properties for the 2D airfoils used in the analysis. For example, a 2D potential flow solution for flow around a NACA 2412 airfoil gives a zero-lift angle of attack of approximately $\alpha_{L0} = -0.0380\ \text{rad}$ and a section lift slope of approximately $a_0 = 6.86/\text{rad}$, but 2D viscous flow solutions give somewhat different values. XFOIL (see Refs. [51,52]) predicts a zero-lift angle of attack of approximately $\alpha_{L0} = -0.0372\ \text{rad}$ and a section lift slope of approximately $a_0 = 6.26/\text{rad}$ for a NACA 2412 airfoil in viscous flow at a Reynolds number of $\text{Re} = 1\text{E}6$.

* https://github.com/jacobwilliams/json-fortran

Similarly, the zero-lift moment coefficient and moment slope of an airfoil will also be different between inviscid and viscous flows and, for viscous flows, will depend on Reynolds number. Section parasitic drag is estimated in MachUp using the relation

$$c_{d_p} = c_{d_0} + c_{d_1} c_l + c_{d_2} c_l^2 \tag{2.6.1}$$

where the coefficients c_{d_0}, c_{d_1}, and c_{d_2} must be determined using a viscous analysis tool such as XFOIL or from experimental data. These coefficients are identically zero by definition for inviscid flows. MachUp sums the individual section parasitic drag components computed from Eq. (2.6.1) about the aircraft center of gravity to evaluate global viscous forces and moments as described by Phillips [53].

The example input file in Appendix E contains other parameters not discussed here, and still other parameters are supported by MachUp that have not been included in the example. A comprehensive description of each available parameter is beyond the scope of this work. The reader is instead referred to the MachUp documentation and source code available on Github.

2.6.2 Modifications to the MachUp Source Code

The original and modified lines of code required for DNAD integration in MachUp have been listed in Appendix D. Two of the MachUp source files – `main.f90` and `json.f90` – were left unmodified. A few other files – `airfoil.f90`, `dataset.f90`, and `section.f90` – required only the addition of the preprocessor directives from Figure 2.3.

After adding the preprocessor directives from Figure 2.3, several compiler errors were encountered due to improper handling of data type conversions between `real` and `type(dual)`. These compiler errors were resolved by manually inspecting the source code and ensuring that floating point variables were properly declared as either `real` or `REAL` as discussed in Sec. 2.4.3.

All of the remaining code changes dealt directly with I/O functions of the program. An intermediate layer between the third-party JSON interface and the computational portion of the MachUp source code already existed prior to DNAD integration (see `myjson.f90`), but was not fully implemented. Several lines of code still called directly into the `json_m` module, so that variables declared as `type(dual)` were not properly processed. These errors were resolved by completing the implementation of the `myjson_m` module and adding some additional functions specifically for handling variables of `type(dual)`. The preprocessor directive for converting all `real` variable declarations to `type(dual)` was not added to the `myjson_m` module (as was done with the other modules) so that functions for handling input and output of both data types are included in the compiled executable. Fortran `interfaces` were defined at the beginning of the `myjson_m` module so that the appropriate I/O function is called based on the type declarations of the arguments passed to the function. As with other DNAD-specific changes, the functions and interfaces for handling variables of type(dual) were enclosed in preprocessor directives so that they are only included in the compiled executable when the DNAD module is activated.

DNAD implementation in the manner described here affords some very significant advantages over other automatic differentiation methods. Because all JSON-specific I/O operations are processed through the `myjson_m` module, no variable-specific code is needed to make a floating-point variable available for derivative calculations. Instead, derivatives with respect to any floating-point variable contained in the JSON input file can be computed simply by converting the single value in the JSON input file to a two-value array. For example, consider again the example JSON input file given in Appendix E. The angle of attack and sideslip angle are specified under the “`condition`” keyword:

```
"condition": { "alpha": 0, "beta": 0 }
```

Derivatives with respect to angle of attack can be computed simply by replacing the above line with the following:

```
"condition": { "alpha": [0, 1], "beta": 0 }
```

Note, however, that angle of attack is specified in the input file in units of degrees, so that partial derivatives of output quantities will have units of deg^{-1}. Partial derivatives of lift coefficient, pitching moment coefficient, and other aerodynamic performance coefficients with respect to angle of attack are typically reported in units of rad^{-1}. This conversion can be performed after the analysis, but it can be handled more conveniently by specifying the operating conditions as:

```
"condition": { "alpha": [0, 57.2957795131], "beta": 0 }
```

so that, by nature of the chain rule of differentiation, all partial derivatives output by the code will have units of rad^{-1}.

The JSON output file, written upon successful execution of the MachUp “`forces`” command, also takes advantage of the `myjson_m` module so that partial derivative information is automatically written for all variables of `type(dual)`. As with the input file, no variable-specific code modifications were needed to enable output of partial derivative information, so that the partial derivative information is available for all variables written to a JSON output file automatically.

2.6.3 Results and Discussion

The order of convergence of MachUp can be estimated by considering how the solution changes as the number of nodes is increased. For this analysis, we use an elliptic wing with a straight quarter-chord, an average chord of 1, an aspect ratio of 8, a uniform cross-section with a section lift slope of 2π, and no geometric or aerodynamic twist. The wing is operating at an angle of attack of $\alpha = 1\deg$. Figure 2.17 shows the magnitude of the percent difference between the lift and drag computed for several grid densities relative to that computed using $n = 1280$, where n is the mesh size (i.e. number of nodes per semispan). The plot reveals an approximately second-order convergence rate for both lift and induced drag.

An elliptic wing with the same parameters described above was also used to evaluate the accuracy of derivative calculations. Derivatives of lift and drag with respect to angle of attack were evaluated using the DNAD module, forward differencing (Eq. (2.2.11)), and central differencing (Eq. (2.2.13)). Percent errors were computed relative to the DNAD solution using a grid size of 1280 nodes per semispan. Results are plotted in Figures 2.18 and 2.19.

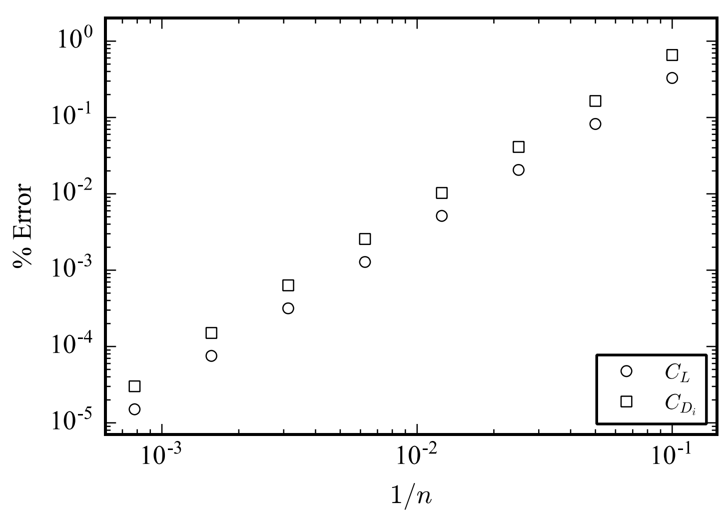


Figure 2.17 Percent error in lift and drag coefficients for various grid densities relative to the solution computed using $n = 1280$ nodes per semispan.

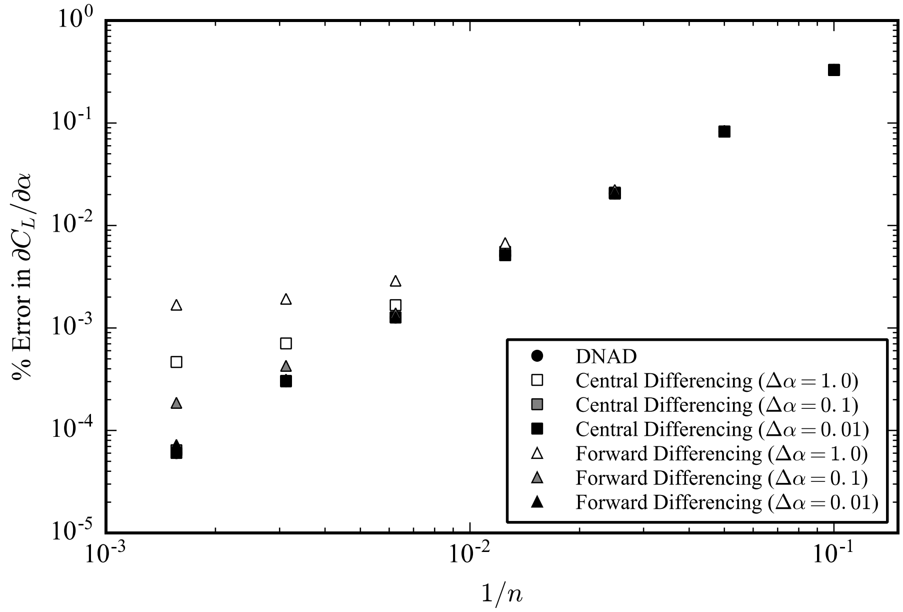


Figure 2.18 Comparison of errors in $\partial C_L / \partial \alpha$ using DNAD and finite differencing relative to the DNAD solution computed using 1280 nodes per semispan.

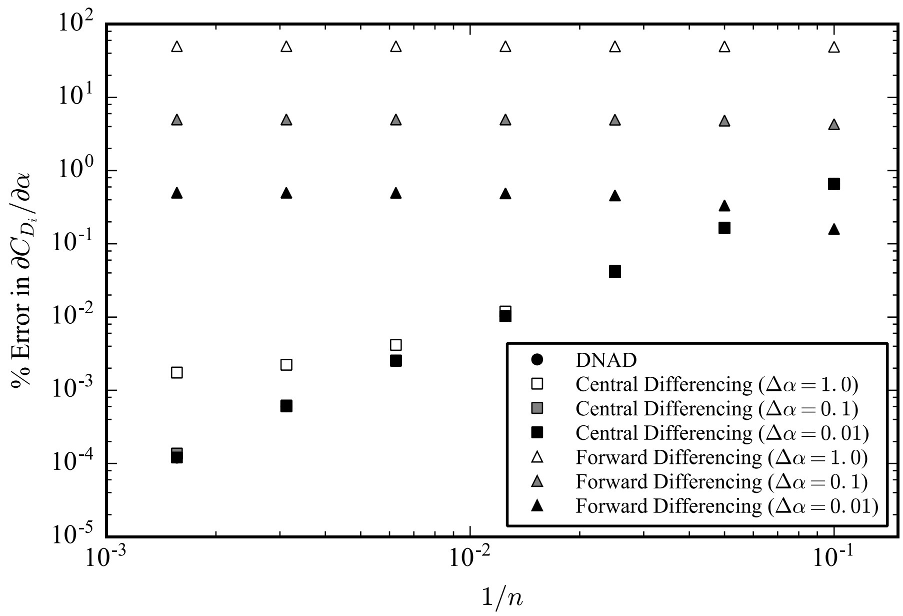


Figure 2.19 Comparison of errors in $\partial C_{D_i} / \partial \alpha$ using DNAD and finite differencing relative to the DNAD solution computed using 1280 nodes per semispan.

Derivatives of lift with respect to angle of attack computed using DNAD and the two finite difference formulas are nearly indistinguishable for grid sizes below 100 nodes per semispan or when a perturbation step size of $\Delta\alpha = 0.01$ is used. This is because the lift is nearly linear in angle of attack so that the finite difference approximations are able to closely represent the function. On the other hand, the behavior of induced drag is approximately quadratic in angle of attack. The forward difference method (Eq. (2.2.11)) does a relatively poor job of predicting $\partial C_{D_i} / \partial\alpha$, while the central difference method (Eq. (2.2.13)) is still nearly as accurate as the DNAD calculations, especially for small $\Delta\alpha$. There seems to be little advantage in terms of accuracy to using DNAD over the second-order central difference method in these calculations when an appropriate step size is used. However, determination of an appropriate step size is still an added requirement to the finite difference method, and other derivative calculations whose primary functions are neither linear nor quadratic may not be satisfactorily represented by either finite difference method. DNAD calculations are not limited by either of these issues.

Classical lifting line theory [10,11] provides closed-form solutions to the partial derivatives of lift and induced drag with respect to angle of attack, namely

$$\frac{\partial C_L}{\partial\alpha} = a = \frac{a_0}{1 + a_0 / \pi A} \tag{2.6.2}$$

$$\frac{\partial C_{D_i}}{\partial\alpha} = \frac{2C_L}{\pi A}\frac{\partial C_L}{\partial\alpha} \tag{2.6.3}$$

While the numerical lifting line algorithm implemented in MachUp is closely related to this theory, results for $\partial C_L / \partial\alpha$ and $\partial C_{D_i} / \partial\alpha$ computed with MachUp do not converge exactly to the solutions given by Eqs. (2.6.2) and (2.6.3). This is because Prandtl [10,11] aligned the trailing wake with the chord line of the wing in his derivation, while Phillips and Snyder [16] aligned the trailing wake with the freestream. This change in formulation results in a difference of less than 0.002% in both $\partial C_L / \partial\alpha$ and $\partial C_{D_i} / \partial\alpha$ between the results of Eqs. (2.6.2) and (2.6.3) and those of MachUp with a grid size of 1280 nodes per semispan. This difference is why the percent errors plotted in Figures 2.18 and 2.19 must be computed relative to the fine-grid MachUp results and not the closed-form solutions.

As with the adpsolver code, time benchmarks for MachUp were collected using a standard desktop computer running Windows 10 Enterprise 64-bit with a third-generation Intel Core i7-3770 3.4 GHz processor, 16 GB of 1600 MHz DDR3 RAM, and a 1 TB 7200 RPM Serial ATA internal hard drive. The MachUp code was compiled using version 6.3.0 of the GNU Fortran compiler.

The example input file given in Appendix E was used as a baseline input file for all of the benchmarking analyses, except that the "`targetcl`" and "`stl`" commands were disabled and DNAD calculations were requested by changing selected floating point inputs to 2-value lists and compiling the code using the appropriate DNAD preprocessor commands. Also, two different grid sizes were used: 40 and 80 nodes per semispan. Figure 2.20 shows timing results for the differentiated code normalized by the time required to execute the undifferentiated code. Also included for comparison are the normalized times that would be required to compute the same derivatives using first-order forward differencing (Eq. (2.2.11)) and second-order central differencing (Eq. (2.2.13)).

The performance of MachUp with DNAD integration is shown to be highly dependent on the number of nodes per semispan (n) used in the simulation. This is because, for small n, the majority of execution time is spent on problem setup and I/O operations which are not heavily influenced by the DNAD module. As grid size increases, the percentage of execution time spent on computations such as LU decomposition and forward and backward substitution also increases, so that the performance impact of the DNAD module is more heavily felt.

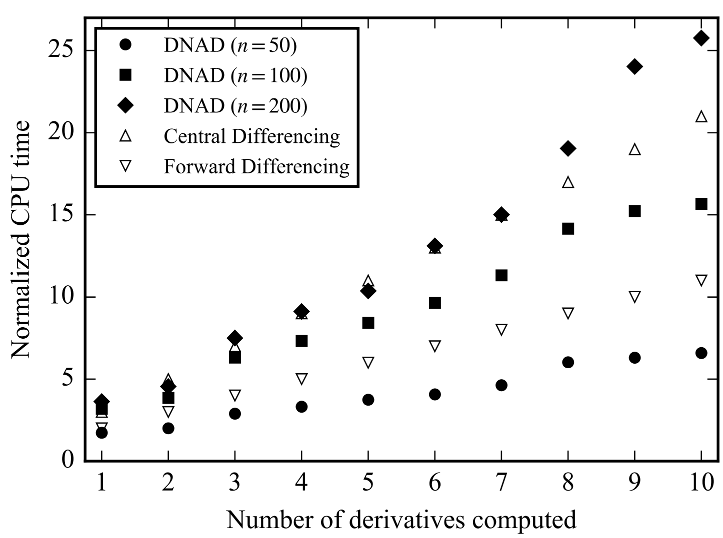


Figure 2.20 Run-time benchmarks for MachUp using DNAD and finite differencing.

We note disproportionately large increases in the DNAD execution times shown in Figure 2.20 when going from 2 derivatives to 3 and from 7 to 8. This has been attributed to single instruction, multiple data (SIMD) vectorization capabilities of the hardware. Vectorization extensions such as Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) use SIMD registers to process a single instruction on multiple data more efficiently than if the instruction was processed on each piece of data individually. The size of the SIMD registers sets limitations on how many pieces of data can be processed with a single vectorized instruction, so that behavior such as that seen in Figure 2.20 when going from 2 derivatives to 3 and from 7 to 8 is expected.

As a result of the above observations, the answer to whether DNAD calculations are more advantageous than finite difference methods depends on the size of the model and the purpose of the analysis. If the entire model has fewer than 400 nodes and fewer than 8 design variables for which derivatives are required, DNAD presents an appealing solution in terms of both runtime efficiency and accuracy. For larger models and problem setups with 8 or more design variables, the finite differencing approach may be satisfactory. For problems with less than 400 nodes and 8 or more design variables, a reverse-mode AD method may be the most efficient option, but integration of a reverse-mode AD tool and verification of this is beyond the scope of the current work.

3 WING SHAPE OPTIMIZATION USING A NUMERICAL LIFTING LINE ALGORITHM AND DUAL NUMBER AUTOMATIC DIFFERENTIATION

3.1 Introduction

Aerodynamic shape optimization is the process of designing the outer mold line (OML) of an aerodynamic structure such that specific aerodynamic performance requirements are met in the most efficient manner possible. As mentioned in Chapter 1, CFD analyses can be used to generate high-fidelity aerodynamic performance predictions, but the computational cost of CFD makes it ill-suited for use as the objective function in aerodynamic shape optimization problems. Lyu et al. [4] states that visualization of design spaces for these types of problems is not possible with current CFD capabilities and hardware limitations. This leads aerodynamicists to consider lower-fidelity aerodynamic tools that trade accuracy for runtime efficiency. This approach can be especially useful early in the design process when insights into trends and interactions between design parameters are more important than highly-accurate performance characteristics.

One such tool is MachUp, an implementation of the numerical lifting line method of Phillips and Snyder [16]. This tool was discussed in Sec. 2.6, where a method for automatic differentiation of the software was presented. In this chapter, we discuss the use of MachUp as the objective function in an aerodynamic shape optimization problem. Note that the modified version of MachUp discussed in Sec. 2.6, which uses DNAD for derivative calculations, is used here. We first present Optix, an open-source optimization framework being developed in the Utah State University AeroLab and available through the groups Github page[*]. The version of the code considered in this work is included in Appendix F. We then present solutions to several inviscid wing shape optimization problems that were computed using MachUp and Optix, and compare the results to known analytical solutions developed using classical lifting line theory [10,11]. Similar wing shape optimization problems using viscous airfoil properties are then presented. Finally, we present and discuss a wing design contour plot that was generated using MachUp and serves to visualize the complete design space for the viscous wing shape optimization problems discussed previously.

[*] https://github.com/usuaero/optix

3.2 Optix

Optix is an open-source, gradient-based optimization framework written in Python. It has the capability of solving a wide range of nonlinear optimization problems and offers some unique features that set it apart from similar optimization frameworks. Because Optix is written entirely in Python, it is easy to interface with and can readily be extended and customized to fit a particular problem. Optix is also cross-platform and highly portable. It can be run on any machine where Python is available, and NumPy (see Oliphant [54]) is the only package dependency outside of the Python Standard Library. Any combination of software available on the host machine can be executed within the objective function, so that the objective function – typically the most computationally expensive part of an optimization analysis – can be written in a high-performance computing language such as Fortran or C++. The following subsections highlight the key features of Optix.

3.2.1 Optimization Method

Optix implements the BFGS method of Broyden [55], Fletcher [56], Goldfarb [57], and Shanno [58]. This is a quasi-Newton method that uses historical calculations of the objective function and its gradient to approximate a Hessian matrix, which in turn is used to estimate a search direction. According to Nocedal and Wright [59] the BFGS method is among the most popular quasi-Newton methods. It has been implemented in several scientific computing packages that offer gradient-based optimization utilities, including the GNU Scientific Library, MATLAB, R, and SciPy.

At the beginning of the BFGS algorithm, the objective function and its gradient at the initial design point are calculated and the Hessian matrix is initialized to the identity matrix (i.e. $[\mathbf{H}]_0 = [\mathbf{I}]$) so that the first search direction corresponds to the direction of steepest descent. A line search is then performed to find the local minimum in the search direction. The design point is then updated to this location, and a new gradient vector is calculated. The BFGS algorithm then determines an updated Hessian matrix $[\mathbf{H}]_{k+1}$ from the previous Hessian matrix $[\mathbf{H}]_k$ according to

$$[\mathbf{H}]_{k+1} = [\mathbf{H}]_k + \frac{\left[1 + \left(\{\boldsymbol{\gamma}\}[\mathbf{H}]_k\{\boldsymbol{\gamma}\}^T\right)\right]}{\{\boldsymbol{\delta}\mathbf{x}\}\{\boldsymbol{\gamma}\}^T} \frac{\{\boldsymbol{\delta}\mathbf{x}\}^T\{\boldsymbol{\delta}\mathbf{x}\}}{\{\boldsymbol{\delta}\mathbf{x}\}\{\boldsymbol{\gamma}\}^T} - \frac{\{\boldsymbol{\delta}\mathbf{x}\}^T\{\boldsymbol{\gamma}\}[\mathbf{H}]_k + [\mathbf{H}]_k\{\boldsymbol{\gamma}\}^T\{\boldsymbol{\delta}\mathbf{x}\}}{\{\boldsymbol{\delta}\mathbf{x}\}\{\boldsymbol{\gamma}\}^T} \tag{3.2.1}$$

where

$$\{\boldsymbol{\delta}\mathbf{x}\} = \{\mathbf{x}\}_{k+1} - \{\mathbf{x}\}_k \tag{3.2.2}$$

$$\{\boldsymbol{\gamma}\} = \{\nabla\mathbf{f}\}_{k+1} - \{\nabla\mathbf{f}\}_k \tag{3.2.3}$$

are the changes in the design point and gradient vector, respectively, between the current and previous iterations. The direction of the next line search is then given by

$$\{\mathbf{s}\}_{k+1} = -[\mathbf{H}]_{k+1}\{\nabla\mathbf{f}\} \tag{3.2.4}$$

This algorithm is repeated until either of two exit criteria are met: 1) the curvature condition proposed by Wolfe [60,61] is no longer satisfied, or 2) the change in the objective function between iterations falls below a user-specified threshold. Once one of these conditions is satisfied, the Hessian matrix is reset to the identity matrix, and the entire process begins again using the last minimum as the initial design point. The optimization process is complete when one of the two exit criteria are met on an iteration where the Hessian matrix is equal to the identity matrix. The current value of the objective function and the corresponding design point are then returned to the calling function as the final results of the optimization process.

3.2.2 Code Structure

Optix is composed of two object classes and several utility functions. The source code for these classes and functions are listed in Appendix F. The first object class handles execution of the user-specified objective function. The objective function is a Python function provided by the user that must accept as arguments the current design point and a case identifier, and it must return the value of the objective function evaluated at the current design point. A second function can be provided to evaluate gradients of the objective function, but this is not required. If specified, the gradient function must accept the same arguments as the objective function. It must return both the value of the objective function and the gradient vector at the specified design point. If this function is not specified, Optix evaluates the primary objective function multiple times with perturbed design points and uses a second-order central differencing algorithm – see Eq. (2.2.13) – to approximate the gradient of the objective function.

The second object class contained in the Optix source code is used for controlling the optimization algorithm. The number of design variables, their names, and their initial values are specified through an instance of this class. In addition, line search settings, parameters for the exit criteria, and whether to use the linear or quadratic line search algorithms (explained later in Sec. 3.2.4) are set here. A helper function is also provided through which the user can specify a JSON file from which to load these optimizer settings.

The `optimize` function orchestrates the entire optimization process. It accepts one instance each of the two object classes described above and returns the optimized objective function value and corresponding design point upon completion. The `optimize` function relies on the Python NumPy package for efficient matrix computations.

3.2.3 Parallel Execution of Independent Function Evaluations

Optix uses the `multiprocessing` module (part of the Standard Python Library) to execute independent evaluations of the objective function simultaneously. For example, each function evaluation within the line search is independent of all other line search evaluations, so the separate function evaluations can be divided among available processors and run simultaneously. In addition, if finite differencing is used to approximate the gradient vector, these function evaluations can also be executed in parallel. One gradient evaluation using the second-order central differencing formula given in Eq. (2.2.13) requires $2N+1$ function evaluations, where N is the number of active design variables in the optimization analysis. Running these evaluations in parallel can significantly reduce the total optimization time when even just a few design variables are active.

3.2.4 Linear and Quadratic Line Searching

As mentioned previously, Optix relies on line searching methods to locate the local minimum in a given search direction. Two types of line searching algorithms exist in the literature, namely exact and inexact methods. Exact methods typically require some *a priori* knowledge of the design space to be searched and are restricted in application to only specific problem types. For example, the conjugate gradient method of Hestenes and Stiefel [62] requires the design space to be composed of a system of linear equations whose matrix is symmetric and positive-definite. While exact line searching algorithms are typically quite efficient when applied to the correct types of problems, restrictions to specific problem types make them ill-suited for a general optimization framework such as Optix.

Inexact line searching algorithms use approximation methods to estimate the minimum in a given search direction, and then rely on the optimization algorithm to narrow in on the design space minimum through successive updates to the gradient vector and Hessian matrix. Two popular inexact algorithms are the backtracking algorithms of Goldstein [63] and Armijo [64], which attempt to overshoot the minimum of the objective function and then successively reduce the step size until the minimum is sufficiently bounded.

Optix provides two backtracking algorithms of relative simplicity: a linear and a quadratic algorithm. The linear algorithm evaluates the objective function at multiple design points in the search direction. Evaluations continue at increasingly larger distances from the initial design point until an increase in the objective function is obtained, at which point the algorithm fits a parabola through the last three design points evaluated. The next design point is placed at the parabola's vertex.

The quadratic algorithm evaluates the objective function at a user-specified number of equally-spaced design points along the search direction and fits a parabola to the results. If the minimum of the parabola is within a user-specified threshold of any one of the design points evaluated, or if a minimum does not exist (i.e. the parabola reduces to a straight line or is concave), the design point corresponding to the minimum objective function value is returned to the BFGS algorithm. If neither of these conditions are met, a new set of design points is selected with one of them located at the parabola's vertex, and the objective function is reevaluated at these new locations. For convex problems, the quadratic line searching algorithm can significantly reduce the total optimization time from that required for the linear algorithm.

3.2.5 Limitations

One limitation of Optix over other similar optimization frameworks is the lack of built-in methods for applying bounds and constraints. Constraints can still be enforced using penalty function methods (for example, see Smith and Coit [65]), but more efficient and robust methods exist. Some example methods include the method of Lagrange multipliers for problems having only equality constraints, the simplex method for linear programming problems, and the ellipsoid method for quadratic programming problems. Explanations of these methods can be found in any introductory textbook on gradient-based optimization (for example, see Griva et al. [66]). However, it should be noted that each method is only suited to a specific subset of constrained optimization problems, whereas the penalty function method can be applied universally to all constrained optimization problems.

Another limitation of Optix is its inability to distinguish between local minima and the global minimum of a design space. For optimization problems that involve complex design spaces with multiple local minima, results returned by Optix may depend on the initial design point provided to the optimizer. This is a common shortfall of gradient-based optimization methods, but there are some algorithms – e.g. the conservative convex separable approximation (CCSA) methods presented by Svanberg [67] – that have overcome this limitation. Locatelli and Schoen [68] present a thorough overview of globally-convergent optimization methods. The BFGS method implemented in Optix is not globally convergent. However, a simple design of experiments (DOE) approach can be used to improve the likelihood of finding the global minimum of a design space. The DOE approach systematically selects multiple design points distributed throughout the design space and uses each point to initiate a complete gradient-based optimization analysis. The collective results from this set of analyses does not guarantee discovery of the global minimum within the design space but simply improves the likelihood of finding it. The level of complexity of the design space will determine the number of simulations needed to provide adequate coverage of the design space. The computational cost of this approach can be significant since each analysis is a full optimization analysis, but the separate analyses are completely independent and can be run simultaneously given adequate computational resources. In the sample analyses that follow, the objective functions are convex so the locally-convergent BFGS method is satisfactory for this work.

3.3 Wing Shape Optimization in Inviscid Flow

Optix and MachUp have been used to solve several wing shape optimization problems for which closed-form solutions are available. Results from these simulations are presented and discussed here. Each simulation began with the same initial wing model – namely a spanwise-symmetric, untwisted, rectangular wing with an aspect ratio of $A = 8$ and a section lift slope of $a_0 = 6.8806 \text{rad}^{-1}$ – operating at a wing lift coefficient of $C_L = 0.5$. Control points (locations where the optimizer is allowed to vary geometric and aerodynamic properties of the wing) were spaced uniformly along one semispan of the wing. The other semispan was automatically updated so that the wing remained symmetric. All simulations used a grid density of 100 nodes per semispan. The main MachUp input file and Python code used for these simulations are given in Appendix G.

The aerodynamic properties of the airfoils used in these analyses are shown in Table 3.1 and correspond to the NACA 4-digit X412 family of airfoils with percent maximum cambers, $\bar{z}_{c_{max}}$, ranging from 0% to 8%. The data were generated using an inviscid panel code. The induced drag for the initial rectangular model at a lift coefficient of $C_L = 0.5$ was calculated to be $C_{D_i} = 1.0554 \times 10^{-2}$.

Table 3.1 Aerodynamic coefficients for the NACA X412 family of airfoils in inviscid flow

Airfoil	α_{L0} (rad)	a_0 (rad^{-1})
NACA 0012	0.0000	6.8806
NACA 2412	-0.0380	6.8583
NACA 4412	-0.0761	6.8369
NACA 6412	-0.1141	6.8165
NACA 8412	-0.1519	6.7973

Prandtl [10,11] showed analytically that, for a finite wing of given aspect ratio, the induced drag is minimized by an elliptic lift distribution according to the relation

$$l = \frac{4L}{\pi b}\sqrt{1-(2y/b)^2} \tag{3.3.1}$$

where L is the total lift generated by the wing, y is the spanwise coordinate measured from the root of the wing, and b is the wingspan. In nondimensional form, Eq. (3.3.1) becomes

$$c_l = \frac{4b}{\pi A}\frac{C_L}{c}\sqrt{1-(2y/b)^2} \tag{3.3.2}$$

where c is the local chord and A is the aspect ratio determined by the wingspan b and the planform area S_w according to the relationship

$$A \equiv \frac{b^2}{S_w} \tag{3.3.3}$$

The induced drag generated by a wing having this lift distribution is given by

$$C_{D_i} = \frac{C_L^2}{\pi A} \tag{3.3.4}$$

For example, a wing having an elliptic lift distribution, an aspect ratio of $A = 8$, and a wing lift coefficient of $C_L = 0.5$ will have an induced drag coefficient of $C_{D_i} = 9.9472\times10^{-3}$, which is about 6% less than the rectangular planform mentioned above. This result is used for comparison in the following inviscid optimization analyses.

3.3.1 Optimized Planform Shapes for Minimum Induced Drag

Prandtl [10,11] proposed achieving the elliptic lift distribution by using an elliptic planform, i.e. a chord distribution prescribed by

$$c = \frac{4b}{\pi A}\sqrt{1-(2y/b)^2} \tag{3.3.5}$$

and no geometric or aerodynamic twist. For this planform, the section lift coefficient specified by Eq. (3.3.2) reduces to a constant value equal to the wing lift coefficient. Here we present a numerical optimization solution to this problem. To solve for the optimized planform shape numerically, control points were spaced uniformly along one semispan of the wing and the optimizer was configured to adjust the local chord length at each control point except the wing tip. The chord length at the wing tip was constrained such that an aspect ratio of $A = 8$ was maintained, thus making the degrees of freedom for this problem one less than the number of control points *N*. A linear interpolation scheme was used to define the section chord at spanwise locations between control points. The gradient of the induced drag coefficient was calculated internally with MachUp using the DNAD module discussed in Chapter 2.

The planform shape was optimized using 2, 3, 6, and 11 control points. Results are summarized in Table 3.2. When $N = 2$, the planform corresponds to a tapered wing with a taper ratio of approximately $R_T = 0.366$ and produces 1.121% more induced drag than the elliptic planform. Adding a third control point reduces this percentage by almost a fourth. As the number of control points is increased, the induced drag for the optimized solution converges toward that of an elliptic planform at the expense of increasing planform complexity and computational cost.

Figure 3.1 compares the final planform design of each optimization simulation to the initial rectangular planform and an elliptic planform of the same aspect ratio. The optimized solutions appear to converge toward the elliptic planform as the number of degrees of freedom increases. In each case, the maximum deviation from the elliptic planform occurs at the wing tip where the curvature is highest and the section lift coefficient and contribution to induced drag are smallest.

Table 3.2 Minimum induced drag results for untwisted wings with optimized planforms

Case	**BFGS Iterations**	C_{D_i}	**Difference from Eq. (3.3.4)**
Rectangular	–	0.010554	6.099%
$N = 2$	5	0.010059	1.121%
$N = 3$	9	0.009977	0.297%
$N = 6$	18	0.009952	0.051%
$N = 11$	38	0.009949	0.014%
Elliptic	–	0.009947	–

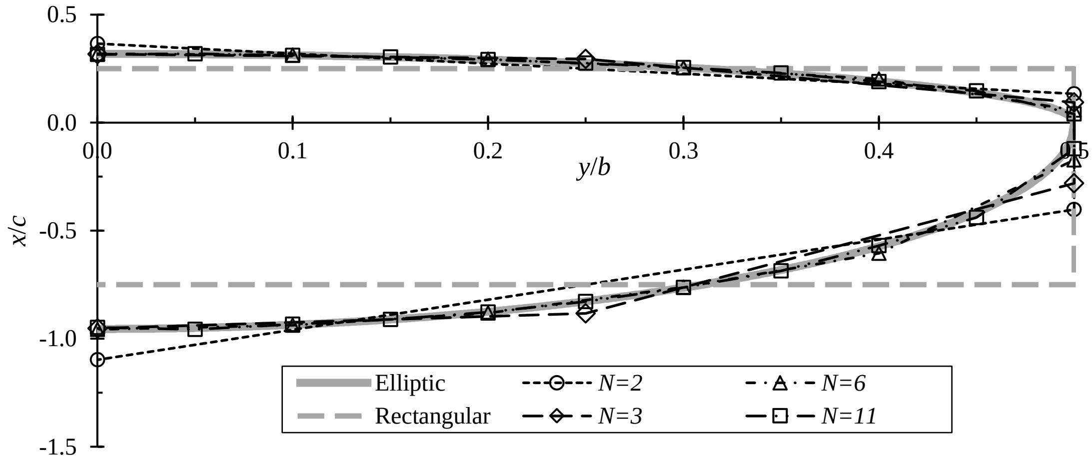


Figure 3.1 Optimized planforms for minimum induced drag.

3.3.2 Optimized Geometric Twist Distributions for Minimum Induced Drag

In 2004, Phillips [69] extended the analytical solutions of Prandtl [10,11] and showed that the minimum induced drag produced by a finite wing with an elliptic planform and no twist can also be achieved by a finite wing of arbitrary planform by prescribing the total twist as a function of spanwise location. For a generic wing with no sweep and no dihedral, Phillips [69] showed that the optimum twist distribution as a function of spanwise location is given by

$$\left(\alpha - \alpha_{L0}\right)_{\text{root}} - \left(\alpha - \alpha_{L0}\right) = \Omega_{\max}\left(1 - \sqrt{1-(2y/b)^2}\right) \tag{3.3.6}$$

where the angles α and α_{L0} represent local section values. The maximum twist $\Omega_{\max}$ is related to the lift coefficient according to

$$\Omega_{\max} = \frac{4bC_L}{\pi A a_0 c_{\text{root}}} \tag{3.3.7}$$

and occurs at the wing tips ($y = \pm b/2$). For a rectangular wing with uniform camber (i.e. no aerodynamic twist), $\alpha_{L0} = (\alpha_{L0})_{\text{root}}$ so that Eq. (3.3.6) reduces to

$$\Delta\alpha_{\text{geometric}} = \Omega_{\max}\left(1 - \sqrt{1-(2y/b)^2}\right) \tag{3.3.8}$$

where

$$\Delta\alpha_{\text{geometric}} \equiv \alpha_{\text{root}} - \alpha \tag{3.3.9}$$

Equation (3.3.8) describes an elliptic geometric twist distribution analogous to the elliptic chord distribution given by Eq. (3.3.5). Note that, by definition, the geometric twist at the root is zero. For the present optimization analyses, the root angle of attack α_{root} was determined such that a lift coefficient of $C_L = 0.5$ was maintained. The amount of twist at each control point other than the wing root was controlled by the optimizer, so that the degrees of freedom were again one less than the number of control points *N*.

The optimization problem was again solved numerically with 2, 3, 6, and 11 control points. The wing design was constrained to a rectangular planform with a uniform NACA 0012 cross section. Results are summarized in Table 3.3. We again see that the induced drag converged toward that of an elliptic lift distribution as the number of control points was increased. Comparing the results in Table 3.3 to those in Table 3.2 for equal *N*, optimizing the geometric twist distribution is slightly more efficient than optimizing the chord distribution in terms of both the number of BFGS iterations required and the aerodynamic performance of the optimized wings.

Figure 3.2 compares the geometric twist distribution determined by each optimization analysis to that prescribed by Eq. (3.3.8). We again see that the optimization solutions converge toward the expected solution as the number of control points increases, and the largest deviation from the geometric twist distribution prescribed by Eq. (3.3.8) occurs at the wing tip in each case.

Table 3.3 Minimum induced drag results for geometric-twist-optimized rectangular wings

Case	BFGS Iterations	C_{D_i}	Difference from Eq. (3.3.4)
Rectangular	–	0.010554	6.099%
N = 2	4	0.010026	0.793%
N = 3	7	0.009960	0.124%
N = 6	13	0.009948	0.009%
N = 11	24	0.009947	0.001%
Elliptic	–	0.009947	–

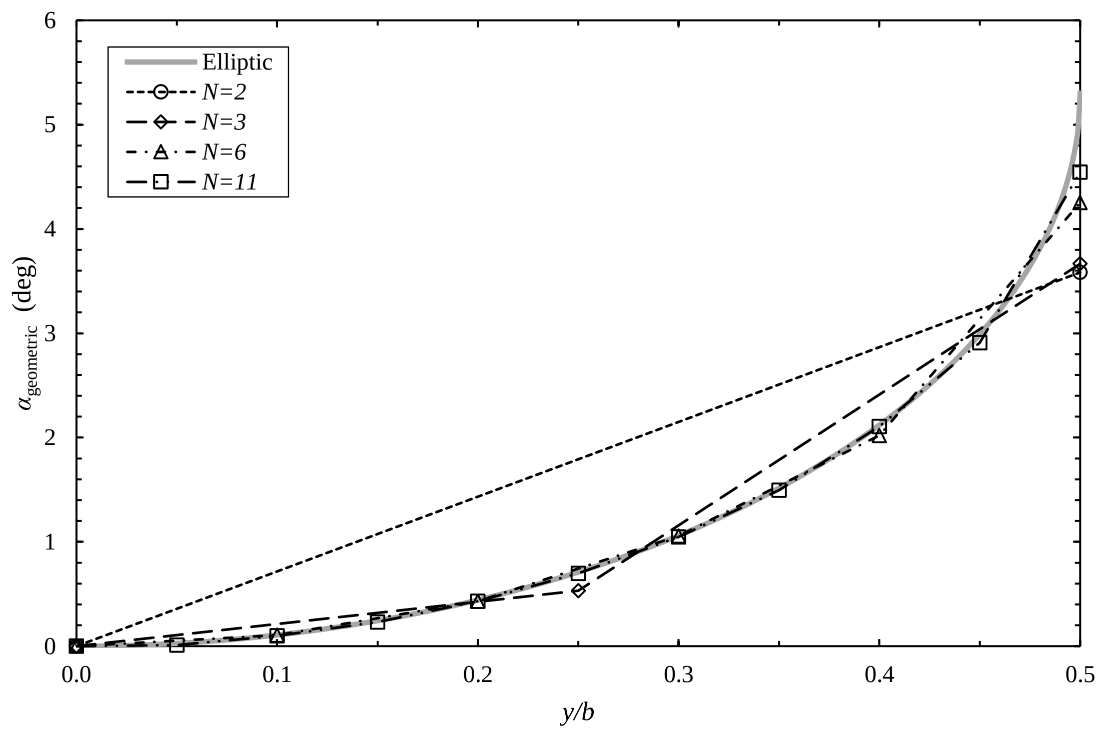


Figure 3.2 Optimized geometric twist distributions for minimum induced drag.

3.3.3 Optimized Aerodynamic Twist Distributions for Minimum Induced Drag

In the previous section, the geometric twist distribution of a finite rectangular wing with no aerodynamic twist was optimized for minimum induced drag. Similar results can be obtained by setting the geometric twist to zero and instead optimizing the spanwise aerodynamic twist of the wing. In this section, we present a method for optimizing the spanwise aerodynamic twist of a wing by allowing the optimizer to select from a family of airfoils of varying amounts of camber. For the analyses presented here, airfoil selection was restricted to the NACA X412 airfoils with maximum camber values ranging from 0% to 8% of the chord. Airfoil properties from Table 3.1 were used, and properties for airfoils in the specified camber range but not explicitly listed in Table 3.1 were determined by linear interpolation.

The section lift slopes of the NACA X412 airfoils listed in Table 3.1 vary slightly. Because of this variation, Eq. (3.3.7) cannot be applied directly without introducing some error into the analytical solution. Since the standard deviation of the section lift slopes listed in Table 3.1 is less than half a percent, the average section lift slope from this family of airfoils can be used in Eq. (3.3.7) with negligible effect on the results, and we can use the approximation $\alpha \approx \alpha_{\text{root}}$ in Eq. (3.3.6) such that

$$\Delta\alpha_{\text{aerodynamic}} \approx \Omega'_{\text{max}}\left(1-\sqrt{1-(2y/b)^2}\right) \tag{3.3.10}$$

where

$$\Delta\alpha_{\text{aerodynamic}} \equiv \alpha_{L0} - (\alpha_{L0})_{\text{root}} \tag{3.3.11}$$

$$\Omega'_{\text{max}} = \frac{4bC_L}{\pi A(a_0)_{\text{avg}} c_{\text{root}}} \tag{3.3.12}$$

and the prime indicates an approximation due to the use of the average section lift slope. Optimization analyses with families of airfoils having a large standard deviation in section lift slope will not produce an aerodynamic twist distribution consistent with Eq. (3.3.10). However, the section lift distribution of the final optimized solution should still be consistent with Eqs. (3.3.1) and (3.3.2).

Because Eq. (3.3.10) specifies only a difference between the section zero-lift angles of attack and does not specify their absolute magnitude, an additional constraint is needed to isolate a single optimal solution. In the results that follow, the airfoil section at the wing tip was set to a NACA 2412 airfoil. The optimizer was configured to control the airfoil definition at all other control points by specifying the percent maximum camber $\bar{z}_{c_{\text{max}}}$, which MachUp then used to determine the airfoil properties at each control point via linear interpolation of the data in Table 3.1.

The resulting optimal designs for 2, 3, 6, and 11 control points are summarized in Table 3.4. The corresponding aerodynamic twist distributions are shown in Figure 3.3. Induced drag results are almost identical to those obtained for optimized geometric twist. The number of BFGS iterations required to obtain the optimized solutions were also comparable between the two sets of analyses in all cases except N = 11. For N = 11, about 2.5 times fewer iterations were required to find the optimal geometric twist distribution as were required to find the optimal aerodynamic twist distribution. This is because the percent maximum camber value of the third control point from the root was almost exactly 4% in the latter case. Since linear interpolation is used to determine the airfoil properties, derivatives are discontinuous at the interpolation nodes. This led to some confusion in the optimization algorithm regarding the appropriate search direction.

Table 3.4 Minimum induced drag results for aerodynamic-twist-optimized rectangular wings

Case	BFGS Iterations	C_{D_i}	Difference from Eq. (3.3.4)
Rectangular	–	0.010554	6.099%
$N = 2$	5	0.010026	0.792%
$N = 3$	7	0.009960	0.124%
$N = 6$	15	0.009948	0.009%
$N = 11$	61	0.009947	0.001%
Elliptic	–	0.009947	–

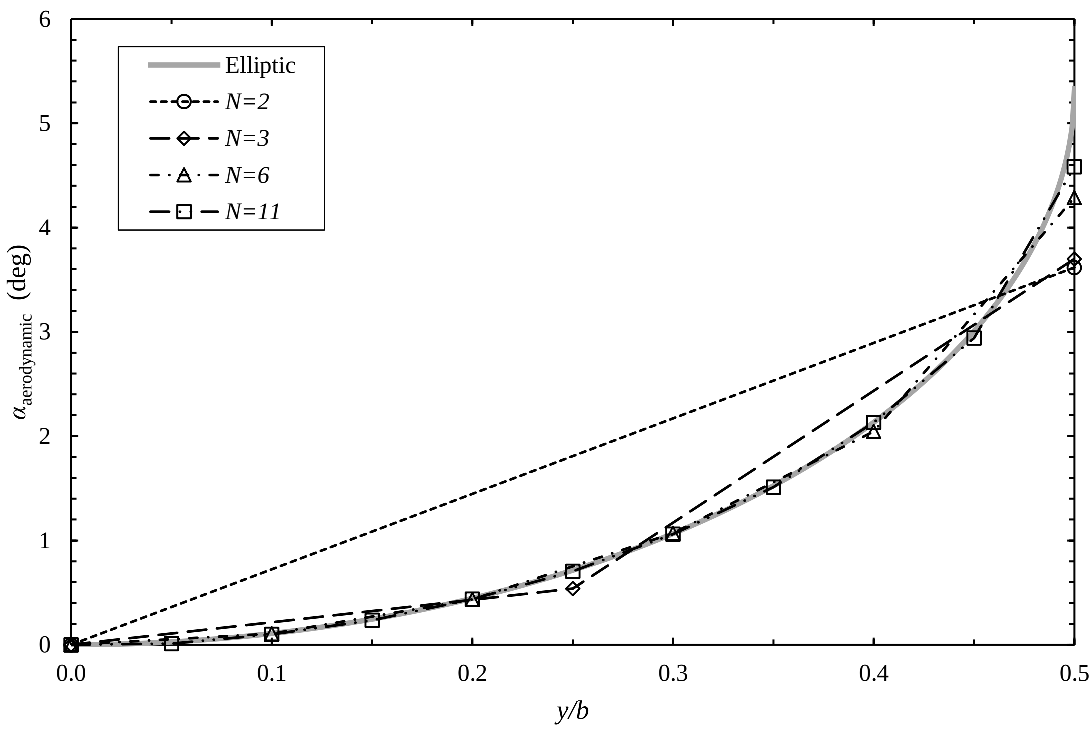


Figure 3.3 Optimized aerodynamic twist distributions for minimum induced drag.

3.4 Wing Shape Optimization for Viscous Flow

While the optimization analyses presented in Sec. 3.3 are beneficial from an academic perspective due to the existence of known closed-form solutions, they have limited practical application due to their neglect of viscous effects. As described in Sec. 2.6.1, viscous effects can be included within a MachUp analysis by using viscous airfoil data that includes estimates of the section parasitic drag coefficients c_{d_0}, c_{d_1}, and c_{d_2} from Eq. (2.6.1). Estimates of these parasitic drag coefficients, the zero-lift angle of attack, and the section lift slope are listed in Table 3.5 for the NACA X412 family of airfoils in viscous, incompressible flow at a Reynolds number of $Re = 10^6$. These coefficients were determined using XFOIL (see Refs. [51,52]).

For the inviscid optimization problems presented in Sec. 3.3, closed-form solutions were available for comparison with the optimized results produced by Optix. For the viscous optimization problems of this section, no closed-form solutions exist. An untwisted elliptic wing with an aspect ratio of $A = 8$, uniform camber, and no sweep or dihedral will be used as a baseline model for comparisons. An optimization analysis was performed to determine the optimum airfoil section (selected from the NACA X412 family of airfoils) to achieve minimum total drag for this baseline elliptic wing operating at a lift coefficient of $C_L = 0.5$. The optimum airfoil section selected by the optimizer has a maximum camber of $\bar{z}_{c_{\max}} = 3.59\%$.

The setup for the optimization cases presented in this section was similar to that of the inviscid cases of Sec. 3.3 with one important difference. The airfoil specified at the tip section had negligible effect on the induced drag results of the foregoing analyses, and therefore was selected arbitrarily in each analysis. However, this is not the case for the viscous analyses presented in this section. In each case here, an additional degree of freedom was needed to allow the optimizer to specify the airfoil at the tip section (and therefore the entire wing for wings of uniform cross-section). As a result, the degrees of freedom were equal to the number of control points for each viscous analysis. The initial rectangular wing design used to start each viscous analysis was identical to that used in the corresponding inviscid analyses except that the airfoil section at each control point was initialized to the airfoil section of the baseline elliptic wing, i.e. $\bar{z}_{c_{\max}} = 3.59\%$. The drag results for this initial rectangular wing and the baseline elliptic wing in viscous flow are summarized in Table 3.6.

Table 3.5 Aerodynamic coefficients for the NACA X412 family of airfoils in viscous, incompressible flow at a Reynolds number of $Re = 10^6$.

Airfoil	α_{L0} (rad)	a_0 (rad^{-1})	c_{d_0}	c_{d_1}	c_{d_2}
NACA 0012	0.00000	6.4194	0.00562	0.00000	0.00804
NACA 2412	-0.03720	6.2639	0.00612	-0.00362	0.00827
NACA 4412	-0.07536	6.1093	0.00780	-0.00753	0.00829
NACA 6412	-0.11176	6.0833	0.01091	-0.01067	0.00739
NACA 8412	-0.15006	5.9184	0.01556	-0.01556	0.00821

Table 3.6 Drag coefficients for initial rectangular wing and baseline elliptic wing in viscous flow.

Component	Initial Rectangular Wing	Baseline Elliptic Wing	Percent Difference
C_{D_i}	0.010643	0.009952	6.94%
C_{D_p}	0.006149	0.006085	1.06%
C_D	0.016792	0.016037	4.71%

3.4.1 Optimized Planform Shapes for Minimum Total Drag

Here we present the planform shapes that correspond to minimum total drag, including viscous effects, as determined by Optix and MachUp for 2, 3, 6, and 11 control points. Total drag results for these cases are summarized in Table 3.7. The viscous cases generally required more BFGS iterations than their inviscid counterparts listed in Table 3.2. The one exception to this is $N = 11$, where the viscous case required four fewer BFGS iterations than the inviscid case, a reduction in computational cost of about 10%. In all four cases, the airfoil section selected by the optimizer matched that selected for the baseline elliptic wing to within 0.01% maximum camber.

The planform designs produced by the optimizer are plotted in Figure 3.4 and are nearly indistinguishable from those shown in Figure 3.1. Indeed, the chord lengths at each control point differ between the two cases by less than 1% of the average chord. These slight differences became smaller as the number of control points increased, and we therefore conclude that the optimum planform design for an untwisted wing is essentially the same in both viscous and inviscid flows and has the elliptic spanwise chord distribution defined by Eq. (3.3.5).

Table 3.7 Minimum total drag results for untwisted wings with optimized planforms.

Case	BFGS Iterations	C_D	Difference from Baseline Elliptic Wing
Rectangular	–	0.016792	4.710%
$N = 2$	11	0.016182	0.907%
$N = 3$	13	0.016078	0.256%
$N = 6$	21	0.016045	0.048%
$N = 11$	34	0.016039	0.014%
Elliptic	–	0.016037	–

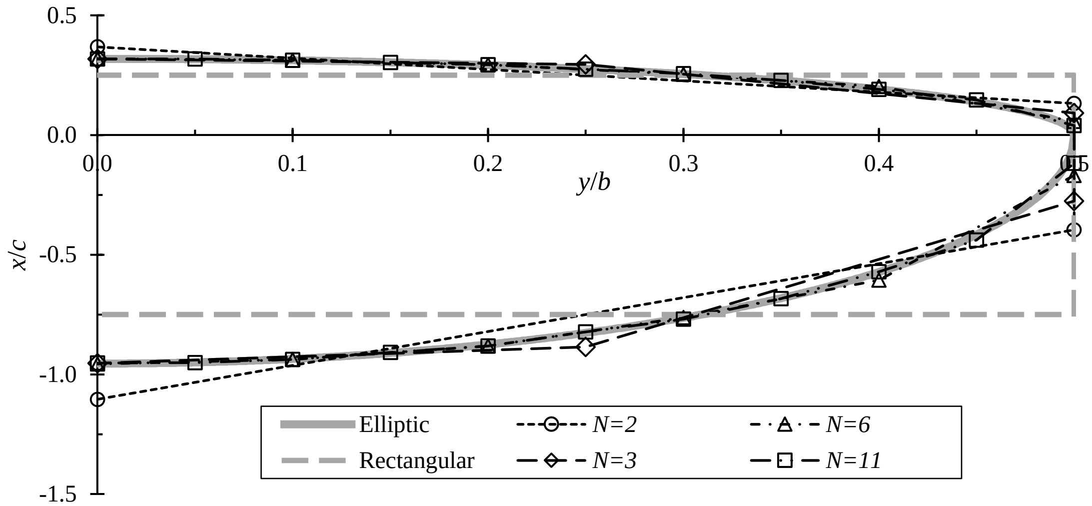


Figure 3.4 Optimized planforms for minimum total drag.

We note one issue in how viscous effects have been included in these analyses. The viscous airfoil coefficients included in Table 3.5 correspond to a Reynolds number based on chord length of $Re = 10^6$. In the optimized planform analyses, however, the chord length is a function of spanwise location and so, therefore, is the Reynolds number. In fact, the spanwise Reynolds numbers for the elliptic planform vary from 1.27×10^6 at the root to 0 at the tip. Adjustments to the airfoil coefficients due to these variations in Reynolds number were not attempted during the analyses presented here. This issue does not affect the inviscid optimization cases presented previously, nor does it affect the following optimization cases in which the chord length is held constant.

3.4.2 Optimized Geometric Twist Distributions for Minimum Total Drag

Table 3.8 summarizes the results for optimized geometric twist distributions to minimize total drag on a rectangular wing with uniform cross section. In each case the drag was more than 1% higher than the baseline elliptic solution. This increased drag came almost entirely from parasitic drag, as shown in Table 3.9 for the case of $N = 11$.

Table 3.8 Minimum total drag results for geometric-twist-optimized rectangular wings.

Case	BFGS Iterations	C_D	Difference from Baseline Elliptic Wing
Rectangular	–	0.016792	4.710%
$N = 2$	6	0.016284	1.537%
$N = 3$	10	0.016213	1.099%
$N = 6$	18	0.016200	1.014%
$N = 11$	29	0.016199	1.008%
Elliptic	–	0.016037	–

Table 3.9 Drag components for geometric-twist-optimized rectangular wing with 11 control points.

Component	Value	Difference from Baseline Elliptic Wing
C_{D_i}	0.009959	0.065%
C_{D_p}	0.006240	2.549%
C_D	0.016199	1.008%

Again, the airfoil section selected by the optimizer for each case matched that selected for the baseline elliptic wing to within 0.01% maximum camber. The optimized geometric twist distributions are plotted in Figure 3.5. For each case, the geometric twist angle selected at each control point differed from the corresponding value for minimum induced drag (see Figure 3.2) by less than 0.06 deg. The only exceptions

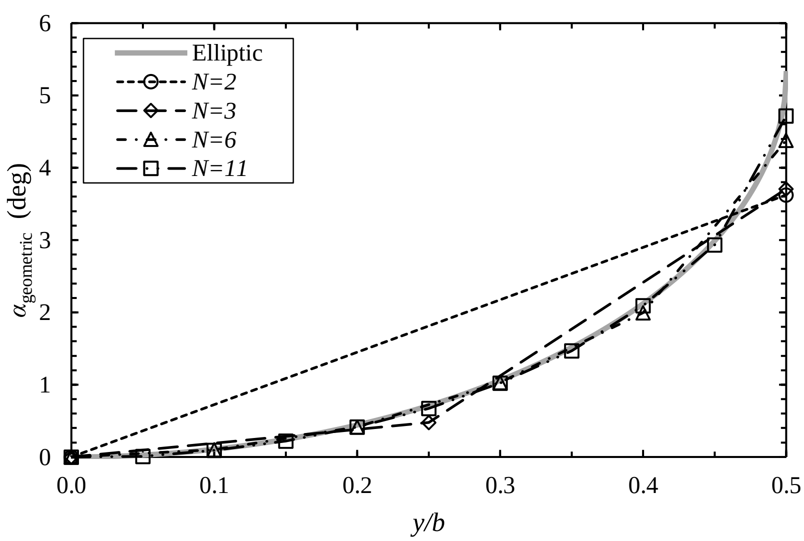


Figure 3.5 Optimized geometric twist distributions for minimum total drag.

to this were at the wing tip for $N = 6$ and $N = 11$, where the slope of the twist distribution curve is large and the influence on wing drag is small.

3.4.3 Optimized Aerodynamic Twist Distributions for Minimum Total Drag

Optimized aerodynamic twist distributions for minimizing total drag are summarized in Table 3.10. Values here are lower than those for the corresponding optimized geometric twist cases, but not as low as those for the optimized planform cases. The parasitic drag component is still the largest contributor to the increase in drag above the baseline elliptic wing result, as shown in Table 3.11 for the case of $N = 11$.

Optimized aerodynamic twist distributions are plotted in Figure 3.6. Noticeable differences between these data and the inviscid data presented in Figure 3.3, especially around $y/b = 0.25$, indicate that the optimized aerodynamic twist distribution for minimum total drag does not converge to the elliptic distribution described by Eq. (3.3.10) as the number of control points increases. This is primarily due to the differences in the parasitic drag equation coefficients of the airfoils in the NACA X412 family. Essentially, the optimizer is favoring an increase in induced drag in these profiles to achieve a more substantial decrease in parasitic drag.

Table 3.10 Minimum total drag results for aerodynamic-twist-optimized rectangular wings.

Case	BFGS Iterations	C_D	Difference from Baseline Elliptic Wing
Rectangular	–	0.016792	4.710%
$N = 2$	13	0.016203	1.033%
$N = 3$	20	0.016107	0.435%
$N = 6$	46	0.016095	0.362%
$N = 11$	122	0.016093	0.351%
Elliptic	–	0.016037	–

Table 3.11 Drag components for aerodynamic-twist-optimized rectangular wing with 11 control points.

Component	Value	Difference from Baseline Elliptic Wing
C_{D_i}	0.009962	0.103%
C_{D_p}	0.006131	0.755%
C_D	0.016093	0.351%

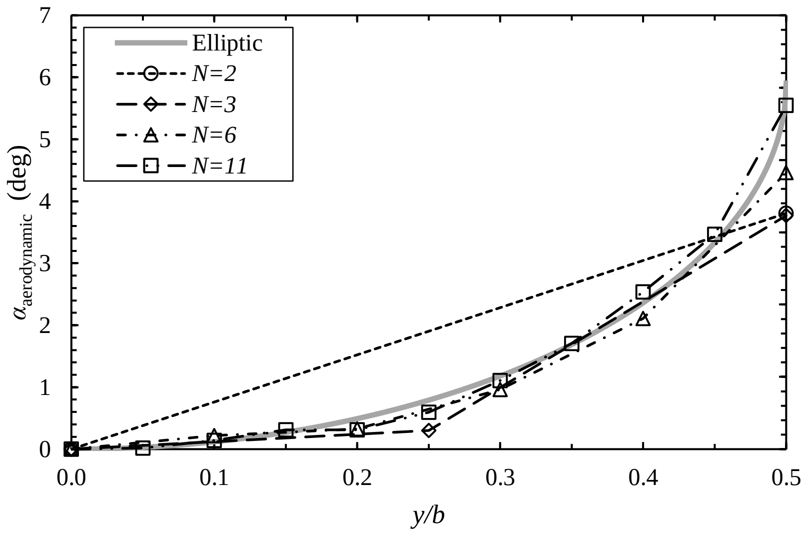


Figure 3.6 Optimized aerodynamic twist distributions for minimum total drag.

3.5 Visualization of the Design Space for Wing Shape Optimization

Let us here consider the aerodynamic-performance design space within which airfoil sections of a finite wing must operate. The total drag produced by any section of an airfoil wing is the sum of the section induced and parasitic drags. The section induced drag is a function of the section lift and induced angle of attack α_i as given by

$$c_{d_i} = c_l \sin(\alpha_i) \tag{3.5.1}$$

The section parasitic drag is a function of the section lift coefficient and parasitic drag coefficients c_{d_0}, c_{d_1}, and c_{d_2} from Eq. (2.6.1). The section lift coefficient is, in turn, a function of the section angle of attack α, the section lift slope a_0, and the section zero-lift angle of attack α_{L0} according to

$$c_l = a_0(\alpha - \alpha_{L0}) \tag{3.5.2}$$

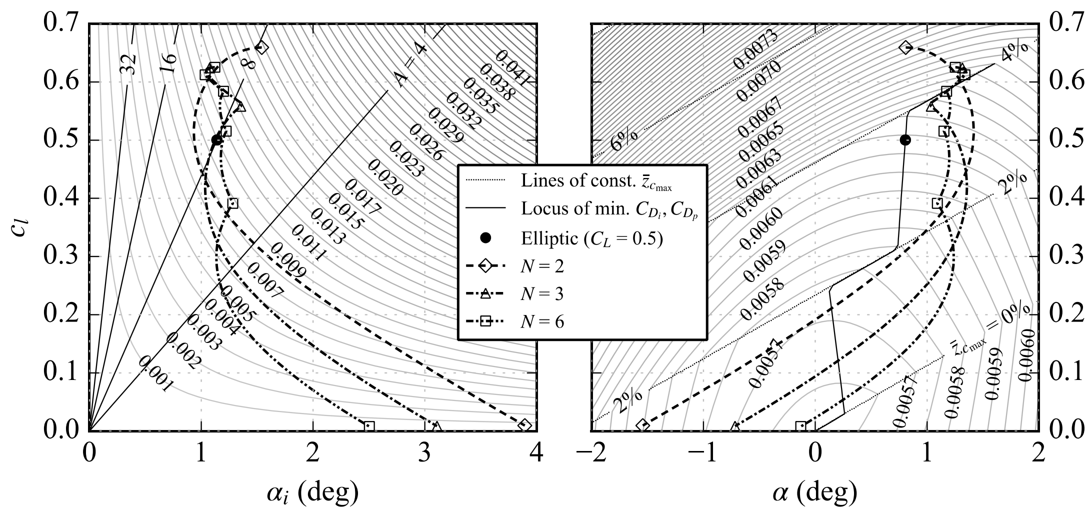
Note that the local angle of attack α is a function of the wing angle of attack α_{wing}, the local induced angle of attack α_i, and the local geometric twist $\Delta\alpha_{\text{geometric}}$ according to

$$\alpha = \alpha_{\text{wing}} + \alpha_i - \Delta\alpha_{\text{geometric}} \tag{3.5.3}$$

The relationships described above have been used to generate the contour plots shown in Figure 3.7. Although these contour plots do not follow the traditional approach to visualizing drag as a function of lift or angle of attack, great care has been taken to construct this figure in a manner that portrays the key relationships when considering the additional dimensions of variable airfoils and wing geometries. Visualizing the data in this format can provide significant insights into the intricacies of wing shape optimization.

Figure 3.7(a) presents contours of section induced drag as a function of section induced angle of attack and section lift coefficient, and it is independent of airfoil section. Figure 3.7(b) presents contours of section parasitic drag as a function of section angle of attack and section lift coefficient, and it is specific to the NACA X412 family of airfoils defined in Table 3.5. Figure 3.7(b) uses the same linear interpolation scheme implemented in MachUp for determining lift and parasitic drag coefficients for airfoils intermediate to those listed in Table 3.5. First-order discontinuities in the contour lines are artifacts of this interpolation scheme.

The locus of minimum parasitic drag values for each section lift coefficient has been plotted as a solid line in Figure 3.7(b) and indicates the airfoil section that provides the minimum parasitic drag for a given lift coefficient. The locus of minimum parasitic drag values is first-order discontinuous for the same reason as the contour lines. Using a higher-order interpolation scheme or refining the airfoil data by adding additional airfoils will lead to more realistic contour lines and a smoother locus of minimum parasitic drag values.



a) b)

Figure 3.7 Contours of (a) induced and (b) parasitic drag for finite wing sections. Results are included for rectangular wings with $A = 8$, $C_L = 0.5$, and optimized aerodynamic twist for variable *N*.

Unlike the locus of minimum parasitic drag values, the corresponding locus of minimum induced drag values is a function of aspect ratio. For an untwisted elliptic wing, downwash is uniform and all sections of the wing experience the same induced angle of attack and, therefore, the same section lift coefficient. Thus $c_l = C_L$ and $c_{d_i} = C_{D_i}$ along the entire wingspan. Using these substitutions in Eq. (3.3.4), we can then combine Eqs. (3.3.4) and (3.5.1) to yield

$$c_l = \pi A \sin(\alpha_i) \tag{3.5.4}$$

Equation (3.5.4) has been plotted in Figure 3.7(a) for a selection of aspect ratios.

Now consider again the untwisted elliptic wing. Downwash is uniform for this wing, and therefore every section along the wingspan operates at the same angle of attack, induced angle of attack, and lift coefficient. The data for every section along the wingspan therefore collapses to a single point on each of the contour plots in Figure 3.7. This point is shown for an untwisted elliptic wing with an aspect ratio of $A = 8$ and uniform camber of $\bar{z}_{c_{\max}} = 3.59\%$, operating at a lift coefficient of $C_L = 0.5$.

Prandtl [10,11] showed analytically that an untwisted elliptic wing produces the least amount of induced drag possible for a given aspect ratio and lift coefficient, and Phillips [69] extended this work to show that the same minimum in induced drag can be achieved by a rectangular wing having an elliptic twist distribution. This can be seen visually in Figure 3.7(a). Because the relationship between lift and induced drag is approximately linear for small angles (see Eq. (3.5.1)), any two wings whose sections all lie along the same line of constant α_i and integrate to the same lift coefficient will also integrate to the same induced drag coefficient.

The relationship between lift and parasitic drag is nonlinear and strongly dependent on the characteristics of the airfoils selected, so that designs which produce minimum induced drag may not necessarily produce minimum parasitic drag. It is the job of an optimizer, then, to balance these two competing objectives in order to find a design that represents the minimum total drag for a given design space and given constraints.

The patterned lines with symbols in Figure 3.7 represent data from the optimization cases of Sec. 3.4.3 – that is, optimized aerodynamic twist distributions for minimum total drag produced by a rectangular wing of aspect ratio $A = 8$ with no geometric twist and operating at a lift coefficient of $C_L = 0.5$. The symbols indicate the aerodynamic state at each control point. The connecting lines represent the distribution of aerodynamic states at spanwise locations between control points. Note that, for a given wing, changing the camber at one control point will affect the downwash along the entire wingspan, adjusting the location of all other control points for that wing and the shape of the lines connecting them. This is also true of changes in geometric twist and chord in designs where those parameters are allowed to vary. There is no mechanism whereby the optimizer can adjust the aerodynamic state of one control point without affecting the aerodynamic state of every section along the wingspan.

The data for a rectangular wing with no geometric twist and an elliptic aerodynamic twist distribution – see Eq. (3.3.10) – would form a vertical line on both contour plots in Figure 3.7. This would minimize induced drag but not parasitic drag. A rectangular wing with a twist and camber distribution such that the data for each section of the wing lie somewhere along the locus of minimum parasitic drag would minimize parasitic drag but not induced drag. In an optimization analysis, the optimizer must find a solution that represents the best possible balance between these two competing objectives, with the added restriction of a limited number of control points.

The ability to visualize this design space and understand the balancing act between the two components of drag is significant. Several rudimentary implications of the data portrayed in Figure 3.7 have been discussed here. More detailed studies of this plot and similar plots based on other families of airfoils may provide additional insights that can aid in our understanding of the design space of finite wings. This level of insight is impossible to gain from traditional approaches to aerodynamic shape optimization that rely on high-fidelity CFD simulations to evaluate the objective function (see Lyu et al. [4]). By limiting the section profiles of the wing to a family of airfoils with known aerodynamic properties, the optimization method presented here reduces the number of design variables and the complexity of the numerical solution by several orders of magnitude. Additionally, this approach allows us to generate drag contours of the design space in order to more fully understand the optimized distributions arrived at through numerical optimization. For applications that lie within the assumptions of lifting line theory, the benefits are realized without a significant reduction in accuracy. For applications that lie outside the limits of these assumptions, this method should be applied with caution but may still be useful in providing qualitative insights to the design space and optimization problem of the specific application.

4 ANALYTICAL LIFTING LINE METHOD FOR WINGS OF ARBITRARY ASPECT RATIO

4.1 Introduction

In 1918, Prandtl [10,11] published his landmark paper on what is known today as classical lifting line theory. This groundbreaking innovation was the first practical method for analyzing flow over a finite three-dimensional wing. From this theory, Prandtl was able to show that the most efficient wing design in terms of minimizing induced drag is one which produces an elliptic spanwise lift distribution. This finding led to the distinctive elliptic planform design of the famous British Supermarine Spitfire, which is considered one of the most strategically important fighter aircraft of World War II. Many other valuable insights to aircraft design and performance have been gleaned from this theory in the one hundred years since its publication.

Despite its successes, classical lifting line theory is not without limitations. For example, it was originally formulated for wings with no sweep or dihedral, and it quickly became clear that Prandtl's theory did not adequately account for the effects of aspect ratio for aspect ratios below about 4 (for example, see Birnbaum [70] and Blenk [71]). However, by building upon the solid foundation laid by Prandtl, researchers have been able to produce methods for overcoming one or more of the early limitations of classical lifting line theory. One method, published in 2000 by Phillips and Snyder [16], accounts for sweep and dihedral in the wing and provides a method for integrating viscous effects into the solution. Another method, known as lifting surface theory (see Multhopp (1938) [72]), extends the applicable range of aspect ratios to less than 1. Küchemann [73] provides an elegant formulation of lifting surface theory which recasts it in such a way that the concepts of lifting surface theory can be readily applied to classical lifting line theory. The current work seeks to combine the efforts of Phillips and Snyder [16] and Küchemann [73] into a single method that incorporates the advantages of both methods. In essence, we seek to use Küchemann's development to modify the numerical lifting line algorithm of Phillips and Snyder so that its range of validity extends to slender wings as well as high-aspect ratio wings.

In this chapter, a rigorous development of classical lifting line theory is presented, along with some important observations regarding elliptic wings that arose from direct application of classical lifting line theory. The equations for slender wing theory and lifting surface theory are also developed. Several observations are made in comparing these theories with classical lifting line theory, and a modified equation for slender wing theory is proposed. Several empirical equations are discussed and compared for modifying classical lifting line theory to bring it into closer agreement with results from lifting surface theory and a high-order panel method. A generalized formulation of lifting line theory is presented that allows us to more easily compare the various modifications that have been presented in the literature. Finally, a new equation is proposed that provides a reasonable balance between simplicity and accuracy for wings of arbitrary aspect ratio and planform shape.

4.2 Lift Generated by a Finite Wing

We begin with a statement of the most basic equation which we are trying to solve – that of lift generated by a finite wing. From classical airfoil theory, the lift generated by a 2D airfoil is related to the angle of attack by the approximation

$$c_l = a_0\left(\alpha - \alpha_{L0}\right) \tag{4.2.1}$$

where small angles have been assumed for both α and α_{L0}. This equation works well for 2D airfoils (i.e. wings of infinite span), but quickly breaks down when we move to 3D wings of finite span. At the wing tips of a finite wing, nothing separates the airflow below the wing from the airflow above the wing. The pressure difference across the wing causes some of the higher-pressure air below the wing to escape around the wingtip and into the lower-pressure region above the wing. As a result, the airflow below the wing will have a slight outward motion (from root to tip), while the airflow above the wing will have a slight inward motion (from tip to root). This effect is more exaggerated at the wingtips but is present along the entire span of a finite wing.

The circulation of air around the wingtips, as just described, generates a sheet of vorticity immediately behind the trailing edge of the wing. The energy that goes into generating this vortex sheet reduces the effectiveness of the wing in generating lift, and Eq. (4.2.1) no longer holds. Empirically, we can modify Eq. (4.2.1) by defining two new angles: the “effective” angle of attack, α_e, and the “induced” angle of attack, α_i. These angles are defined such that

$$\alpha_e = (\alpha - \alpha_{L0}) - \alpha_i \tag{4.2.2}$$

and Eq. (4.2.1) becomes

$$c_l = a_0\alpha_e \tag{4.2.3}$$

To determine the total lift generated by a finite wing, we take the area-weighted average of the lift coefficients at each spanwise location by integrating Eq. (4.2.3) over the wingspan,

$$C_L = \frac{1}{S_w} \int_{y=-b/2}^{b/2} c_l c dy = \frac{1}{S_w} \int_{y=-b/2}^{b/2} a_0 \alpha_e c dy \tag{4.2.4}$$

In doing so, we have relied upon an assumption that was first made by Prandtl [10,11] in his original development of classical lifting line theory – that is, that the spanwise sections of a finite wing behave the same as a 2D airfoil section operating at the same angle of attack. While Prandtl offered no mathematical justification for this assumption, it has proven to be quite reliable through a century of application.

The trick now, and the focus of this chapter, is to determine an expression for α_e such that Eq. (4.2.4) is accurate for finite wings ranging in aspect ratio from slender to infinite. Several researchers, beginning with Prandtl [10,11], have presented methods for determining an appropriate expression for α_e under various assumptions. Those of greatest interest to the present work are classical lifting line theory, slender wing theory, and lifting surface theory. Each, in turn, shall here be presented and discussed.

4.3 Classical Lifting Line Theory

Consider a flat or uniformly cambered wing of finite span in an infinite potential flow field. The boundary condition for this flow field requires that the flow is everywhere tangent to the wing surface such that

$$\frac{w}{V_\infty} = -\frac{dz}{dx} = \alpha - \alpha_{L0} \tag{4.3.1}$$

where we have applied the small angle approximation $\tan(\alpha - \alpha_{L0}) \approx \alpha - \alpha_{L0}$. Note that we have defined the downwash w to be positive for positive angle of attack, so that positive downwash acts in the direction of the $-z$ axis.

Prandtl proposed satisfying this boundary condition by representing the wing as a series of horseshoe vortices, each horseshoe vortex consisting of a single spanwise vortex segment aligned with the quarter-chord of the wing and two semi-infinite vortex segments extending chordwise downstream, as shown in Figure 4.1. In the limit as the number of horseshoe vortices goes to infinity, the streamwise vortex segments form a continuous vortex sheet trailing behind the wing. It is this vortex sheet that is responsible for the reduced

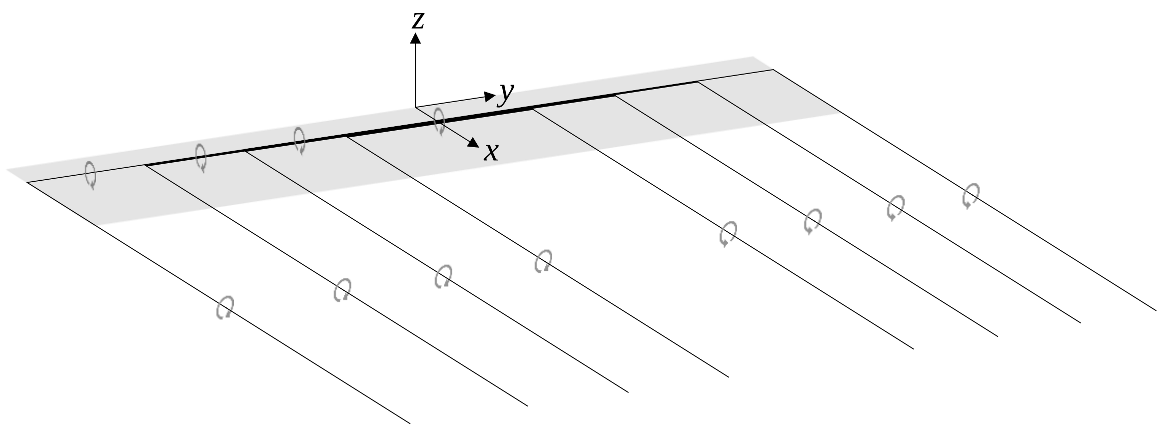


Figure 4.1 Discrete system of overlapping horseshoe vortices on a finite wing.

effectiveness of the wing as was explained earlier. The spanwise vortex segments, on the other hand, generate a force on the surrounding fluid that is equal in magnitude but opposite in direction to the lift force experienced by the wing. This is the "lifting line" from which Prandtl's theory derives its name.

It is important to note here that each horseshoe vortex is a continuous vortex filament that must abide by the vortex theorems of Helmholtz [74] – namely, that the filament cannot end in a fluid (so that the trailing vortex segments must be semi-infinite), and that the circulation strength of the vortex filament cannot vary along its length. Therefore, the strength of each spanwise vortex segment must be equal to that of the two trailing vortex segments to which it is attached, and the direction of circulation must be unchanged along the entire filament (as shown in Figure 4.1).

The problem is now to determine the continuous distribution of vortex strengths $\Gamma(y)$ such that the boundary condition of Eq. (4.3.1) is satisfied. To facilitate this, we first decompose w into two components, w_e and w_i, such that

$$w = w_e + w_i \tag{4.3.2}$$

where w_e, or the "effective downwash," is the downwash due to the spanwise vortex segments; and w_i, or the "induced downwash," is the downwash due to the trailing vortex sheet. Again, both components of downwash are considered positive in the –*z* direction. Note that by dividing Eq. (4.3.2) by V_∞, applying the small angle approximation, and rearranging, we arrive at Eq. (4.2.2), so that both are equivalent statements of the boundary condition.

Consider the semi-infinite vortex filament shown in Figure 4.2. The origin *O* and the arbitrary point *P* lie on the same plane perpendicular to the vortex filament, with the vector **h** pointing from *O* to *P*. From the Biot-Savart law (see Sec. 5.2 of Anderson [75]), the differential velocity vector **dV** induced at point *P* due to the differential vortex element of length **dl** located at point *Q* is given by

$$\mathbf{dV} = \frac{\Gamma}{4\pi}\frac{\mathbf{dl}\times\mathbf{r}}{r^3} \tag{4.3.3}$$

where **r** is the vector pointing from *Q* to *P*. The total velocity induced at point *P* by the entire vortex filament is then

$$\mathbf{V} = \int \mathbf{dV} = \int_0^\infty \frac{\Gamma}{4\pi}\frac{\mathbf{dl}\times\mathbf{r}}{r^3} \tag{4.3.4}$$

By inspection, the length from *O* to *Q* is $h/\tan(\theta)$ so that

$$dl = -\frac{h}{sin^2(\theta)}d\theta \tag{4.3.5}$$

and the magnitudes of the vectors **r** and **h** are related by

$$r = h/\sin\theta \tag{4.3.6}$$

Using Eqs. (4.3.5) and (4.3.6) in Eq. (4.3.4) and taking the magnitude, we get

$$V = \frac{\Gamma}{4\pi}\int_{\pi/2}^{\pi}\frac{\sin\theta}{h}d\theta = \frac{\Gamma}{4\pi h} \tag{4.3.7}$$

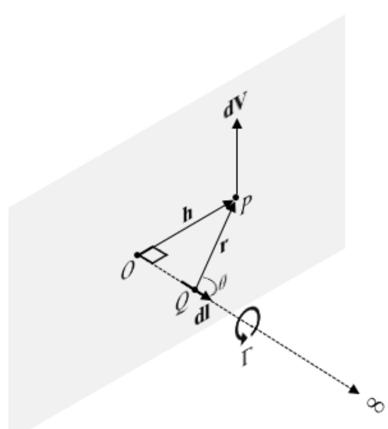


Figure 4.2 Velocity induced at point *P* by a semi-infinite vortex filament.

The direction of V is given by the cross product of the vectors $\overline{OQ}$ and $\overline{OP}$.

Now consider the trailing vortex sheet proposed by Prandtl, consisting of an infinite number of differential semi-infinite vortex filaments originating at the quarter chord line and extending in the direction of the chord. By analogy to Eq. (4.3.7), the differential induced downwash at a point y along the quarter chord by a differential slice of this vortex sheet located at $y = \eta$ is

$$dw_i(y) = \frac{d\Gamma(\eta)}{4\pi(y-\eta)} \tag{4.3.8}$$

The total induced downwash due to the trailing vortex sheet is then the integral of Eq. (4.3.8) over the wingspan

$$w_i(y) = \frac{1}{4\pi} \int_{\eta=-b/2}^{b/2} \frac{d\Gamma(\eta)/d\eta}{y-\eta} d\eta \tag{4.3.9}$$

The effective downwash $w_e(y)$ cannot be determined in the same manner, i.e. using the potential flow equations of a vortex filament, because the locations at which we are interested in determining the velocity lie along the axis of the spanwise vortex segments where the velocity is singular. Prandtl therefore turned to an alternative means for computing the effective downwash – namely, the two-dimensional Kutta-Joukowski theorem. At this point, Prandtl injects his hypothesis that each spanwise section of a finite wing could be modeled as a two-dimensional airfoil subject to the same circulation. The spanwise section lift is then, by the Kutta-Joukowski theorem,

$$l(y) = \rho_\infty V_\infty \Gamma(y) \tag{4.3.10}$$

and from classical airfoil theory we have

$$l(y) = \tfrac{1}{2} \rho_\infty V_\infty{}^2 c(y) c_l(y) = \tfrac{1}{2} \rho_\infty V_\infty{}^2 c(y) a_0 \alpha_e(y) \tag{4.3.11}$$

Equating the right-hand sides of Eqs. (4.3.10) and (4.3.11) and solving for α_e gives

$$\alpha_e(y) = \frac{2\Gamma(y)}{V_\infty a_0 c(y)} \tag{4.3.12}$$

so that the effective downwash becomes

$$w_e(y) = V_\infty \alpha_e(y) = \frac{2\Gamma(y)}{a_0 c(y)} \tag{4.3.13}$$

Substituting the right-hand sides of Eqs. (4.3.9) and (4.3.13) into Eq. (4.3.2) and substituting that result in for the total downwash in Eq. (4.3.1) gives

$$\frac{2\Gamma(y)}{V_\infty a_0 c(y)}+\frac{1}{4\pi V_\infty}\int_{\eta=-b/2}^{b/2}\frac{d\Gamma(\eta)/d\eta}{y-\eta}d\eta=\alpha-\alpha_{L0} \tag{4.3.14}$$

which is the fundamental equation of Prandtl's classical lifting line theory. The only unknown in Eq. (4.3.14) is the circulation distribution $\Gamma(y)$, all other variables being parameters of either the wing geometry or the operating conditions. Solution methods to this equation are not directly relevant to the current discussion; the reader is instead referred to Anderson [75] and Phillips [53].

One important solution to Eq. (4.3.14) is the elliptic circulation distribution given by

$$\Gamma(y)=\Gamma_0\sqrt{1-\left(\frac{2y}{b}\right)^2} \tag{4.3.15}$$

Besides being the most efficient circulation distribution in terms of minimizing induced drag, it conveniently results in uniform induced downwash across the entire wingspan. We can see this by inserting the derivative of Eq. (4.3.15) with respect to *y* into Eq. (4.3.9) and evaluating. The substitution gives

$$\left(w_i\right)_{\text{elliptic}}=-\frac{\Gamma_0}{\pi b^2}\int_{\eta=-b/2}^{b/2}\frac{\eta}{y-\eta}\left[1-\left(\frac{2\eta}{b}\right)^2\right]^{-1/2}d\eta \tag{4.3.16}$$

A convenient method for solving this integral is to use the change of variables $y=-\frac{b}{2}\cos(\theta_0)$ and $\eta=-\frac{b}{2}\cos(\theta)$, which gives

$$\left(w_i\right)_{\text{elliptic}}=\frac{\Gamma_0}{2\pi b}\int_{\theta=0}^{\pi}\frac{\cos(\theta)}{\cos(\theta)-\cos(\theta_0)}d\theta \tag{4.3.17}$$

Eq. (4.3.17) can be evaluated from (see Karamcheti [76])

$$\int_0^{\pi}\frac{\cos(n\theta)}{cos(\theta)-cos(\theta_0)}d\theta=\frac{\pi\sin(n\theta_0)}{\sin(\theta_0)} \tag{4.3.18}$$

By inspection we see that $n=1$ so that

$$\left(w_i\right)_{\text{elliptic}}=\frac{\Gamma_0}{2b} \tag{4.3.19}$$

Consider a flat or uniformly-cambered wing with the elliptic circulation distribution given by Eq. (4.3.15). Eq. (4.3.19) states that the induced downwash is uniform along the wingspan, and therefore by Eq.

(4.3.2) the effective downwash must also be uniform. Substituting the circulation distribution, Eq. (4.3.15), into Eq. (4.3.13) gives

$$(w_e)_{\text{elliptic}} = \frac{2}{a_0 c(y)}\Gamma_0\sqrt{1-\left(\frac{2y}{b}\right)^2} \tag{4.3.20}$$

which (assuming a_0 to be constant over the wingspan) can only be constant if the chord distribution is also elliptic, i.e.

$$c_{\text{elliptic}} = c_0\sqrt{1-\left(\frac{2y}{b}\right)^2} = \frac{4\bar{c}}{\pi}\sqrt{1-\left(\frac{2y}{b}\right)^2} \tag{4.3.21}$$

so that Eq. (4.3.20) becomes

$$(w_e)_{\text{elliptic}} = \frac{\pi\Gamma_0}{2a_0\bar{c}} \tag{4.3.22}$$

Substituting the right-hand sides of Eqs. (4.3.19) and (4.3.22) into Eq. (4.3.2) and solving for Γ_0, we get

$$\Gamma_0 = 2wb\left(\frac{a_0}{a_0+\pi A}\right) \tag{4.3.23}$$

where $A = b^2/S_w = b/\bar{c}$ is the aspect ratio of the wing. Using this result in Eqs. (4.3.19) and (4.3.22) gives

$$w_i = \frac{w}{1+(\pi A)/a_0} \tag{4.3.24}$$

$$w_e = \frac{w}{1+a_0/(\pi A)} \tag{4.3.25}$$

respectively; or, by Eq. (4.3.1),

$$\alpha_i = \frac{\alpha-\alpha_{L0}}{1+(\pi A)/a_0} \tag{4.3.26}$$

$$\alpha_e = \frac{\alpha-\alpha_{L0}}{1+a_0/(\pi A)} \tag{4.3.27}$$

We can now use Eq. (4.3.27) in Eq. (4.2.4) to solve for the total lift coefficient of the wing. The chord is the only variable that is a function of *y*, and the integral of the chord over the wingspan gives the planform area S_w. Then Eq. (4.2.4) evaluates to

$$C_L = a_0\alpha_e = \frac{a_0}{1+a_0/(\pi A)}(\alpha-\alpha_{L0}) \tag{4.3.28}$$

and the effective wing lift slope becomes

$$a = \frac{a_0}{1 + a_0 / (\pi A)} \tag{4.3.29}$$

Equation (4.3.28) gives the lift coefficient of a finite elliptic wing according to Prandtl's classical lifting line theory. Note that in developing this equation we have assumed a_0, α, and α_{L0} are all constant along the wingspan. This led to the requirement that the chord distribution be elliptic according to Eq. (4.3.21). In reality, the same circulation distribution specified in Eq. (4.3.15) can be achieved through an unlimited number of wing configurations in which the twist (α), camber (α_{L0}), thickness (a_0), and chord are all potential functions of y. In any case, Eq. (4.3.10) still holds so that an elliptic circulation distribution will always produce an elliptic lift distribution. The implications these alternative wing designs have on drag were considered in detail in Chapter 3.

We can now also compute the induced drag generated on the wing by the trailing vortex sheet. This drag is not due to viscous effects (recall that everything we have considered so far is based on inviscid potential flow), but instead is due to the change in local angle of attack resulting from the induced downwash w_i. By definition, lift is the component of the aerodynamic force that is normal to the freestream, and drag is the component tangent to the freestream. The section lift coefficient given by Eq. (4.2.3) is therefore not actually lift relative to the wing but the total resultant aerodynamic force coefficient, c_r, generated by the wing section and acting at an angle α_i to the direction of lift. However, applying the small angle approximation we get

$$c_l = c_r \cos(\alpha_i) \approx c_r \tag{4.3.30}$$

so that Eq. (4.2.3) still holds to a good approximation. The induced drag coefficient is given by

$$c_{d_i} = c_r \sin(\alpha_i) \approx c_r \alpha_i \tag{4.3.31}$$

and the total induced drag is then the area-weighted average of Eq. (4.3.31) similar to Eq. (4.2.4). Evaluating this integral with the definitions for the induced and effective angles of attack from Eqs. (4.3.26) and (4.3.27) gives

$$\alpha_i = \frac{C_L}{\pi A} \tag{4.3.32}$$

$$C_{D_i} = \frac{{C_L}^2}{\pi A} \tag{4.3.33}$$

which is the well-known solution first given by Prandtl [10,11].

Equation (4.3.33) was developed under the assumption that the wing has an elliptic lift distribution, which Prandtl [10,11] showed produces the minimum induced drag for a wing of given aspect ratio and lift. A span efficiency factor (e_s) can be used to adjust Eq. (4.3.33) for other lift distributions. See Phillips [53]. Equation (4.3.33) is then

$$C_{D_i} = \frac{C_L^{\ 2}}{\pi A e_s} \tag{4.3.34}$$

where $e_s = 1$ for any wing with an elliptic lift distribution and $e_s < 1$ for all other lift distributions.

4.4 Slender Wing Theory

The limitations of classical lifting line theory with respect to aspect ratio were recognized shortly after its publication. Prandtl's model was based on the assumption that the chordwise vorticity distribution at every section of a finite wing is the same as that predicted by classical airfoil theory for a two-dimensional wing. This assumption is only strictly true for wings of infinite aspect ratio. For finite wings, the chordwise vorticity distribution is a function of spanwise location. Deviations from the predictions of classical airfoil theory become more pronounced as aspect ratio is reduced, especially at spanwise sections near the wing tips.

In considering the lower limit of aspect ratio, the works of Munk [77], Bollay [78], and Jones [79] provide analytical relationships for the aerodynamic characteristics of finite wings in the limit as $A \to 0$. Their results form the basis of slender wing theory. Here we present a modified formulation of classical lifting line theory based on their results, which is only valid for small aspect ratios ($A < 1$).

In order to develop a formulation for slender wing theory similar to Eq. (4.3.14), we must first consider several important results obtained through the development of slender wing theory. Munk [77] demonstrated that flow in planes perpendicular to the x-axis of a slender body can be viewed as two-dimensional, and he introduced the concept of an "additional apparent mass," m' , that represents the mass of fluid displaced by each chordwise section of the wing as the wing moves into the flow. Jones [79] extended this concept to develop relations for the lift generated by an uncambered slender delta wing. He used Munk's relation for the additional apparent mass,

$$m' = \rho \frac{\pi b^2}{4} dx \tag{4.4.1}$$

which is equivalent to the mass of fluid within a cylindrical volume of diameter *b* and length *dx*. The momentum imparted to the wing at each chordwise section must balance the momentum imparted to the additional apparent mass of fluid at that chordwise section, so that the lift force on the wing becomes

$$l(x) = V\alpha \frac{dm'}{dt} \tag{4.4.2}$$

The time derivative of the additional apparent mass is given by

$$\frac{dm'}{dt} = \frac{dm'}{dx}\frac{dx}{dt} = \left(\rho \frac{\pi b}{2}\frac{db}{dx}dx\right)V \tag{4.4.3}$$

so that the lift becomes

$$l(x) = \frac{1}{2}\rho V^2 \pi\alpha b \frac{db}{dx}dx \tag{4.4.4}$$

and the lift coefficient becomes

$$c_l(x) = \pi\alpha \frac{db}{dx} \tag{4.4.5}$$

The total lift coefficient generated by the delta wing is then given by integration, as in Eq. (4.2.4), but this time we are integrating over the chord. We get

$$C_L = \frac{1}{S_w}\int_0^c c_l(x) b dx = \frac{\pi\alpha}{2}\frac{b_{\max}{}^2}{S_w} = \frac{\pi A}{2}\alpha \tag{4.4.6}$$

since, for any delta wing, $b = 0$ at $x = 0$ and $b = b_{\max}$ at $x = c$. This gives for the lift slope of a slender delta wing

$$a = \frac{\pi A}{2} \tag{4.4.7}$$

which is the well-known result from Jones [79].

Jones then made three observations critical to our current development. The first is that, in order to satisfy the Kutta condition, only sections of increasing span (i.e. $db/dx > 0$) can generate lift, so that the use of $b_{\max}$ in Eq. (4.4.6) is general for any planform regardless of the chordwise location at which $b_{\max}$ actually occurs. The second, which Jones demonstrated through a development of the surface potential, is that the spanwise lift distribution of a slender wing is elliptical and independent of chord distribution. The third observation, which also came about through Jones' development of the surface potential, is that the vorticity distribution of a slender wing is concentrated at the leading edge of the wing, so that the aerodynamic center is also located at the leading edge. We expect these observations to be true for any viable formulation of the slender wing problem.

We now proceed to cast Eq. (4.3.14) in terms of these findings. First we consider the system of horseshoe vortices that must be used to represent a slender wing. In Figure 4.1, the system of horseshoe vortices from classical lifting line theory is shown with the lifting line located at the quarter-chord of the wing, which corresponds to the aerodynamic center for wings of infinite span. Jones [79] demonstrated that the aerodynamic center of a slender wing is located at the leading edge of the wing. Küchemann [73] therefore proposed moving the lifting line from the quarter-chord to the leading edge when evaluating slender wings, and – to represent the infinite chord-to-span ratio of slender wings – placing the leading edge of the wing at $x = -\infty$.

With this modified system of horseshoe vortices, the only change needed to Eqs. (4.3.4) and (4.3.7) is the lower limit of integration, so that Eq. (4.3.8) becomes

$$dw_i(y) = \frac{d\Gamma(\eta)}{2\pi(y-\eta)} \tag{4.4.8}$$

and the induced downwash becomes

$$w_i(y) = \frac{1}{2\pi} \int_{\eta=-b/2}^{b/2} \frac{d\Gamma(\eta)/d\eta}{y-\eta} d\eta \tag{4.4.9}$$

Next we consider the effective downwash generated by slender wings. Küchemann [73] states – and it is obvious from Eq. (4.4.6) that he is correct – that $C_L \to 0$ as $A \to 0$, so that the effective downwash must also go to zero. Thus $w_e \ll w_i$ by Eq. (4.3.2) and we can write, to a good approximation,

$$\frac{1}{2\pi V_\infty} \int_{\eta=-b/2}^{b/2} \frac{d\Gamma(\eta)/d\eta}{y-\eta} d\eta = \alpha - \alpha_{L0} \tag{4.4.10}$$

There is only one possible solution to Eq. (4.4.10), namely

$$\Gamma(y) = wb\sqrt{1-\left(\frac{2y}{b}\right)^2} \tag{4.4.11}$$

which is both elliptic and independent of chord distribution in agreement with the observations of Jones [79] stated earlier. Note that Eq. (4.4.11) differs by a factor of 2 from the elliptic circulation distribution given by

Eqs. (4.3.15) and (4.3.23) in the limit as $A \to 0$. The lift can now be found by integrating Eq. (4.3.10) over the span. In nondimensional form we get

$$C_L = \frac{2}{V_\infty S_w}\int_{-b/2}^{b/2} \rho V_\infty \Gamma(y)dy = \frac{\pi A}{2}\left(\alpha - \alpha_{L0}\right) \tag{4.4.12}$$

which agrees exactly with the lift slope predicted by Jones [79], Eq. (4.4.7).

Previous works on slender wing theory have ended at this point, and we are left with Eq. (4.4.10) as our lifting line theory equivalent of slender wing theory. However, there is one additional step we can take due to the knowledge that our circulation distribution is elliptic according to Eq. (4.4.11). Recall that the induced and effective downwashes produced by a high-aspect-ratio wing with elliptic circulation, Eqs. (4.3.24) and (4.3.25) respectively, are constant over the span. Since our slender wing solution is forced to an elliptic circulation distribution, we conjecture that the two components of downwash are constant in this case as well. Then from Eq. (4.2.4) and (4.4.12) we have

$$C_L = a_0\alpha_e = \frac{\pi A}{2}\left(\alpha - \alpha_{L0}\right) \tag{4.4.13}$$

Solving for α_e and multiplying by V_∞, we get

$$w_e = \frac{\pi A}{2a_0} w \tag{4.4.14}$$

Solving Eq. (4.4.11) for *w* and substituting, we get

$$w_e = \frac{2\Gamma}{a_0\left[\dfrac{4\bar{c}}{\pi}\sqrt{1-\left(\dfrac{2y}{b}\right)^2}\right]} \tag{4.4.15}$$

which is identical to Eq. (4.3.13) except that the actual spanwise chord distribution has been replaced by the elliptic chord distribution of Eq. (4.3.21). Thus for elliptic wings, Eqs. (4.3.13) and (4.4.15) are equivalent. This again agrees with the observations of Jones [79] stated earlier, specifically that the results for a slender wing are independent of chord distribution.

By Eqs. (4.4.9) and (4.4.15) we now have for our slender wing lifting line equation

$$\frac{2\Gamma}{V_\infty a_0\left[\dfrac{4\bar{c}}{\pi}\sqrt{1-\left(\dfrac{2y}{b}\right)^2}\right]} + \frac{1}{2\pi V_\infty}\int_{\eta=-b/2}^{b/2} \frac{d\Gamma(\eta)/d\eta}{y-\eta}d\eta = \alpha - \alpha_{L0} \tag{4.4.16}$$

Eq. (4.4.16) is a more complete statement of slender wing theory than has been presented previously. Results computed using Eq. (4.4.16) are nearly identical to those computed using the slender wing theory of Jones [79] for aspect ratios less than about 1, which is typically considered the upper limit for slender wing theory. Inclusion of the effective downwash term, however, offers important insights as we look to combine slender wing theory and classical lifting line theory into a single theory applicable for all aspect ratios. We shall refer to this formulation hereinafter as the “modified slender wing equation.”

4.5 Lifting Surface Theory

Early efforts to extend the concepts of lifting line theory to wings of low aspect ratio began with the works of Birnbaum [70] and Blenk [71] and developed into what is known today as lifting surface theory. Blenk [71] was able to show good results for aspect ratios down to about 1 by replacing the single lifting line in Prandtl’s formulation with a chordwise distribution of lifting lines, thus forming a lifting surface. His overall formulation, however, was limited to a single finite wing of rectangular planform in inviscid, incompressible flow. He attributed the error for aspect ratios less than one to nonlinear effects that could not be accounted for in his linear formulation.

Zimmerman [80,81] and Winter [82] performed experimental studies on low-aspect-ratio wings. They attributed the divergence between their results and lifting surface theory to stalling phenomena that could not be treated with inviscid theoretical methods. Bollay [78] later developed analytical solutions for a flat rectangular plate in which he attributed the phenomena seen by the aforementioned experimenters to the finite angle at which the trailing vortices leave the plate.

The only purely analytical solution of lifting surface theory currently available in the literature is that of Hauptman and Miloh [83]. By assuming an elliptic planform with a straight midchord, they derived the acceleration potential (see Prandtl [84]) in terms of ellipsoidal harmonics and applied this to a linearized formulation of lifting surface theory. The result was a set of closed-form equations for spanwise lift as a function of aspect ratio. While Hauptman and Miloh [83] state that these equations “are exact within the realm of the linear theory,” the equations are too cumbersome to apply practically as they involve evaluation of the complete elliptic integral of the second kind (see Hauptman and Miloh [85], Smith [86], and Laitone [87]) and, for spanwise distributions, evaluation of infinite summations that do not readily converge.

Using numerical calculations from lifting surface theory, several authors have proposed empirical and semi-empirical expressions for the lift slope of low-aspect-ratio wings. Some of the more interesting proposals come from Jones [88], Helmbold [89], Van Dyke [90], Germain [91], Kida and Miyai [92], and Laitone [87]; but there are many others. Unfortunately, none of these provide spanwise lift distributions in their developments, but only provide equations for the total lift coefficient produced by a wing as a function of aspect ratio. Küchemann [73], on the other hand, presents a very elegant method for correcting the two components of downwash in classical lifting line theory, Eq. (4.3.14), which does allow for the calculation of spanwise lift distributions. He facilitates this by assuming that the effects of aspect ratio are uniform over the span, which is obviously not true near the wing tips, but Küchemann [73] observes that "the error cannot be serious, however, as the lift falls to zero there."

4.6 A Unifying Formulation of Lifting Line, Slender Wing, and Lifting Surface Methods

Without considering spanwise lift distributions (since most of the methods mentioned above lack sufficient development to facilitate consideration of such), we shall here present a unifying formulation of the three theories described above for calculating the wing lift coefficients. All the methods presented here consider only an elliptic planform in their development. Most consider a straight quarter-chord line while some – namely Hauptman and Miloh [83] and Küchemann [73] – consider a straight mid-chord. Panair results given in Figure H.6 of Appendix H show little difference between the two, so we shall not attempt any correction to the methods based on where the straight chord line is located.

In general, the lift coefficient of a finite wing has already been given as

$$C_L = \frac{1}{S_w}\int_{y=-b/2}^{b/2} c_l c(y)dy = \frac{1}{S_w}\int_{y=-b/2}^{b/2} a_0\alpha_e c(y)dy \tag{4.2.4}$$

where

$$\alpha_e = (\alpha - \alpha_{L0}) - \alpha_i \tag{4.2.2}$$

For an elliptic wing, the effective and induced angles of attack predicted by classical lifting line theory are constant over the wingspan and Eq. (4.2.4) becomes

$$C_L = a_0\alpha_e = a_0\left(\alpha - \alpha_{L0} - \alpha_i\right) \tag{4.6.1}$$

The induced angle of attack is related to the lift coefficient according to Eq. (4.3.32),

$$\alpha_i = \frac{C_L}{\pi A} \tag{4.3.32}$$

and by definition the effective wing lift slope is related to the lift coefficient according to

$$C_L = a\left(\alpha - \alpha_{L0}\right) \tag{4.6.2}$$

Using Eqs. (4.3.32) and (4.6.2) in Eq. (4.6.1) and solving for *a* we get

$$a_{\text{classical}} = \left(\frac{1}{a_0} + \frac{1}{\pi A}\right)^{-1} \tag{4.6.3}$$

where we have designated this coefficient with the subscript "classical" to indicate it is the effective lift slope predicted by classical lifting line theory. This equation is strikingly similar to that commonly used to determine the total resistance in a circuit of two parallel resistors, namely

$$R_{\text{tot}} = \left(\frac{1}{R_1} + \frac{1}{R_2}\right)^{-1} \tag{4.6.4}$$

where, by analogy, a_0 is the resistance of the first resistor and $C_L/\alpha_i = \pi A$ is the resistance of the second. We shall use this analogy in comparing the equations of Refs. [10,73,79,83,88–91] by casting the wing lift slope given by each method in the form of Eq. (4.6.4) and comparing the values of R_1 and R_2.

Additionally, we also wish to consider the limits of *a* for the two bounding conditions. From Eq. (4.6.3) we get

$$\lim_{A\to\infty} a_{\text{classical}} = a_0 \tag{4.6.5}$$

which is the exact value predicted by classical airfoil theory, and

$$\lim_{A\to 0} a_{\text{classical}} = \pi A \tag{4.6.6}$$

which is twice the value predicted by slender wing theory, Eq. (4.4.7).

From slender wing theory, Jones [79] gives

$$a_{\text{slender}} = \frac{\pi A}{2} \tag{4.6.7}$$

By comparison to Eq. (4.6.4), $R_1 = \infty$ and $R_2 = \pi A/2$, and the trivial bounding limits are

$$\lim_{A\to\infty} a_{\text{slender}} = \lim_{A\to 0} a_{\text{slender}} = \frac{\pi A}{2} \tag{4.6.8}$$

The modified slender wing equation – Eq. (4.4.16) – gives for the wing lift slope

$$a_{\text{modified\_slender}} = \left(\frac{1}{a_0} + \frac{2}{\pi A} \right)^{-1} \tag{4.6.9}$$

which has the limits

$$\lim_{A \to \infty} a_{\text{modified\_slender}} = a_0 \tag{4.6.10}$$

$$\lim_{A \to 0} a_{\text{modified\_slender}} = \frac{\pi A}{2} \tag{4.6.11}$$

Now let us consider some of the other proposals that attempt to bridge the gap between the two bounding theories. Helmbold [89] proposed

$$a_{\text{Helmbold}} = \frac{a_0}{\sqrt{1 + \frac{a_0^{\,2}}{(\pi A)^2}} + \frac{a_0}{\pi A}} \tag{4.6.12}$$

which, when recast in the form of Eq. (4.6.4), gives

$$a_{\text{Helmbold}} = \left(\sqrt{\frac{1}{a_0^{\,2}} + \frac{1}{(\pi A)^2}} + \frac{1}{\pi A} \right)^{-1} \tag{4.6.13}$$

Helmbold's equation is especially remarkable because it is quite elegant compared to the other proposals and gives the correct limits for both the upper and lower bounds of aspect ratio, namely

$$\lim_{A \to \infty} a_{\text{Helmbold}} = a_0 \tag{4.6.14}$$

$$\lim_{A \to 0} a_{\text{Helmbold}} = \frac{\pi A}{2} \tag{4.6.15}$$

It was also proposed much earlier than most of the other methods considered here, including Jones' slender wing theory [79].

Before publishing his paper on slender wing theory, Jones [88] used potential flow theory to show that the ratio of velocity around the edge of an infinite elliptic wing to that around the edge of a finite wing is equal to the ratio of parameters E for the wings, where E is itself a ratio of the semiperimeter to the span. The parameter E for an infinite wing is 1 so that the ratio of velocities becomes $1/E$. Jones then proposed adjusting the wing lift coefficient according to the relation

$$a_{\text{Jones}} = a_0 \frac{A}{EA + a_0/\pi} \tag{4.6.16}$$

or, in the form of Eq. (4.6.4)

$$a_{\text{Jones}} = \left[\frac{E}{a_0} + \frac{1}{\pi A} \right]^{-1} \tag{4.6.17}$$

This equation has the upper and lower limits of

$$\lim_{A \to \infty} a_{\text{Jones}} = a_0 \tag{4.6.18}$$

$$\lim_{A \to 0} a_{\text{Jones}} = \frac{\pi a_0 A}{\pi + a_0} \tag{4.6.19}$$

Van Dyke [90] approached this problem as a singular perturbation problem in which the span and chord are primary and secondary characteristic dimensions, respectively. He presented a third-order approximation to the lift slope equation as

$$a_{\text{VanDyke}} = \frac{a_0}{1 + \frac{a_0}{\pi A} + \left(\frac{2a_0}{\pi^2 A} \right)^2 \left[\ln(\pi A) - \frac{9}{8} \right]} \tag{4.6.20}$$

Recast in the form of Eq. (4.6.4), Eq. (4.6.20) becomes

$$a_{\text{VanDyke}} = \left\{ \frac{1}{a_0} + \frac{1 + \frac{4a_0}{\pi^3 A}\left[\ln(\pi A) - \frac{9}{8} \right]}{\pi A} \right\}^{-1} \tag{4.6.21}$$

with upper and lower limits of

$$\lim_{A \to \infty} a_{\text{VanDyke}} = a_0 \tag{4.6.22}$$

$$\lim_{A \to 0} a_{\text{VanDyke}} = \frac{\pi^4 A^2}{4a_0 \ln(\pi A)} \tag{4.6.23}$$

Van Dyke’s equation provides reasonable accuracy for aspect ratios above about 2, but fails to converge to the correct limit as $A \to 0$. Even worse, it has an asymptote at about $A = 0.4749$ and approaches the lower limit from the wrong direction. Germain [91] presented a modified version of Van Dyke’s equation which somewhat alleviated these concerns. Germain’s original equation perpetuated an error introduced by Van Dyke, which Van Dyke later corrected [93] to give

$$a_{\text{Germain}} = \frac{a_0}{1 + \frac{a_0}{\pi A} + \frac{4a_0^{\,2}}{(\pi A)^2}\left[\ln\left(1 + \pi e^{-9/8} A\right) \right]} \tag{4.6.24}$$

which can be rearranged as

$$a_{\text{Germain}} = \left(\frac{1}{a_0} + \frac{1 + \frac{4a_0}{\pi^3 A}\left[\ln\left(1 + \pi e^{-9/8} A\right)\right]}{\pi A} \right)^{-1} \tag{4.6.25}$$

This equation eliminates the asymptote found in Van Dyke's equation, and the lower limit

$$\lim_{A \to 0} a_{\text{Germain}} = \frac{\pi^2 A}{\pi + 8e^{-9/8}} \approx 1.72A \tag{4.6.26}$$

comes within about 10% of that predicted by slender wing theory.

Hauptman and Miloh [83] have provided closed-form expressions for the wing lift slope based on the acceleration potential of an ellipse (see Prandtl [84]) and a linearized formulation of lifting surface theory. Their derivation resulted in a piecewise formulation, namely

$$a_{\text{Hauptman\&Miloh}} = \begin{cases} \dfrac{4k}{1 + \dfrac{E^2(h)}{1 + \left(k^2/h\right)\ln\left(1/k + h/k\right)}}, & k = \dfrac{\pi A}{4} \le 1 \\[2ex] \dfrac{4}{k + \dfrac{E^2(h)}{k + \sin^{-1}\left(h\right)/h}}, & k = \dfrac{4}{\pi A} \le 1 \end{cases} \tag{4.6.27}$$

where h is the eccentricity of the ellipse, $k = \sqrt{1-h^2}$, and $E(h)$ is the complete elliptic integral of the second kind. Note that, by definition, the eccentricity of an ellipse is always real and between the limits of 0 and 1, i.e. $0 \le h \le 1$. Also note that Eq. (4.6.27) was taken from Laitone [87], as it corrects a typo in the original paper of Hauptman and Miloh [83]. Rearranging Eq. (4.6.27) gives

$$a = \begin{cases} \left[\dfrac{E^2(h)}{\pi A + (4k^3/h)\ln(1/k + h/k)} + \dfrac{1}{\pi A} \right]^{-1}, & k = \dfrac{\pi A}{4} \le 1 \\[2ex] \left[\dfrac{E^2(h)}{4(k + \sin^{-1}(h)/h)} + \dfrac{1}{\pi A} \right]^{-1}, & k = \dfrac{4}{\pi A} \le 1 \end{cases} \tag{4.6.28}$$

This function approaches the correct limits for both slender and infinite wings, namely

$$\lim_{A \to \infty} a_{\text{Hauptman\&Miloh}} = 2\pi \tag{4.6.29}$$

$$\lim_{A \to 0} a_{\text{Hauptman\&Miloh}} = \frac{\pi A}{2} \tag{4.6.30}$$

However, the section lift slope a_0 does not appear explicitly in Eq. (4.6.27) due to the method by which it was derived, and it is unclear how to correct this equation for section lift slopes other than 2π.

The last equation that we shall consider from the literature is that of Küchemann [73]. Küchemann uses the flat plate distribution of Birnbaum [70] to demonstrate that classical lifting line theory can actually be considered a lifting surface theory. Birnbaum [70] showed that the pressure distribution over a flat infinite plate at angle of attack is given by

$$\Delta C_p(x) = -\frac{2C}{V_\infty}\left(\frac{1 - x/c}{x/c}\right)^{1/2} \tag{4.6.31}$$

which can similarly be expressed as a continuous sheet of vorticity $\gamma_x(x)$ through the Kutta-Joukowski law, where the coefficient C must be determined. The section lift is given by

$$c_l = -\frac{1}{c}\int_0^c \Delta C_p dx \tag{4.6.32}$$

Inserting Eq. (4.6.31) into Eq. (4.6.32) and performing the integration gives

$$c_l = \frac{\pi}{V_\infty} C \tag{4.6.33}$$

or, rearranging,

$$C = \frac{V_\infty}{\pi} c_l \tag{4.6.34}$$

The effective downwash can now be determined by applying the Biot-Savart Law,

$$w_e = -\frac{1}{4\pi c}\int_0^c \int_0^c \Delta C_p(x') \frac{dx'}{x - x'} dx = \frac{C}{2} \tag{4.6.35}$$

Combining Eqs. (4.6.34) and (4.6.35), we get

$$c_l = 2\pi\alpha_e \tag{4.6.36}$$

which gives the section lift slope of a thin airfoil. This demonstrates that using a section lift slope of $a_0 = 2\pi$ in classical lifting line theory is equivalent to a lifting surface theory with the chordwise pressure distribution of Eq. (4.6.31). Other pressure distributions will result in a different section lift slope, so that the chordwise pressure distribution of any section can be incorporated into classical lifting line theory through the use of an appropriate section lift slope.

Küchemann [73] then showed that a slightly modified pressure distribution,

$$\Delta C_p(x) = -\frac{2C}{V_\infty}\left(\frac{1-x/c}{x/c}\right) \tag{4.6.37}$$

can be used to obtain the slender wing results of Jones [79] in a similar manner. The obvious parallels between Eqs. (4.6.31) and (4.6.37) led Küchemann [73] to propose a more general distribution, namely

$$\Delta C_p(x) = -\frac{2C}{V_\infty}\left(\frac{1-x/c}{x/c}\right)^n \tag{4.6.38}$$

where the parameter n can be defined such that a smooth transition occurs between the chordwise distributions of Eqs. (4.6.31) and (4.6.37). Using Eq. (4.6.38) in Eq. (4.6.32) and performing the integration gives

$$C = \frac{V_\infty}{2}\frac{\sin \pi n}{\pi n}c_l \tag{4.6.39}$$

Applying the Biot-Savart Law, Eq. (4.6.35), gives

$$c_l = \frac{4\pi n}{1-\pi n \cot \pi n}\alpha_e \tag{4.6.40}$$

The coefficient in front of α_e can be interpreted as the "effective" section lift slope. Note that, for added generality, 2π in the numerator can be replaced with a_0.

The only known requirements on n are that it must go to 1 in the limit $A \to 0$ and it must go to $1/2$ in the limit $A \to \infty$. On this basis Küchemann [73] proposed, somewhat arbitrarily, to use

$$n = 1 - \frac{1}{2}\left[1 + \left(\frac{a_0}{\pi A}\right)^2\right]^{-1/4} \tag{4.6.41}$$

Noting the similarities between Eqs. (4.3.9) and (4.4.9), Küchemann [73] then proposed multiplying the induced downwash term by the same parameter n. The result was an expression similar to Eqs. (4.3.14) and (4.4.16), from classical lifting line theory and slender wing theory respectively, but general enough to be applied over the whole range of aspect ratio,

$$\frac{2\Gamma(y)}{V_\infty a_0 c(y)}\left(\frac{1-\pi n \cot(\pi n)}{2n}\right) + \frac{n}{2\pi V_\infty}\int_{\eta=-b/2}^{b/2}\frac{d\Gamma(\eta)/d\eta}{y-\eta}d\eta = \alpha - \alpha_{L0} \tag{4.6.42}$$

By assuming an elliptic lift distribution and following the same methodology as was used to arrive at Eq. (4.3.29), the wing lift slope resulting from this equation can be determined. It is

$$a_{\text{Küchemann}} = \frac{2na_0}{1-\pi n\cot(\pi n)+\dfrac{4n^2a_0}{\pi A}} \tag{4.6.43}$$

or, recast in the form of Eq. (4.6.4),

$$a_{\text{Küchemann}} = \left[\frac{1-\pi n\cot(\pi n)}{2na_0}+\frac{2n}{\pi A}\right]^{-1} \tag{4.6.44}$$

The limits of this equation are

$$\lim_{A\to\infty} a_{\text{Küchemann}} = a_0 \tag{4.6.45}$$

$$\lim_{A\to 0} a_{\text{Küchemann}} = \frac{\pi A}{2} \tag{4.6.46}$$

What is significant here is that Küchemann [73] has provided us with a mechanism whereby any formulation cast in the form of Eq. (4.6.4), with definitions for R_1 and R_2, can be directly implemented in the more general lifting line equation

$$\frac{2\Gamma(y)}{V_\infty a_0 c(y)}\left(\frac{a_0}{R_1}\right)+\frac{1}{4\pi V_\infty}\left(\frac{\pi A}{R_2}\right)\int_{\eta=-b/2}^{b/2}\frac{d\Gamma(\eta)/d\eta}{y-\eta}\,d\eta = \alpha-\alpha_{L0} \tag{4.6.47}$$

This equation can be solved in the same manner as Eq. (4.3.14).

Let us consider again the elliptic circulation distribution given by Eq. (4.3.15), namely

$$\Gamma(y) = \Gamma_0\sqrt{1-\left(\frac{2y}{b}\right)^2} \tag{4.3.15}$$

but this time applied to the general lifting line equation, Eq. (4.6.47). The induced downwash is given by

$$\left(w_i\right)_{\text{elliptic}} = \frac{1}{4\pi}\left(\frac{\pi A}{R_2}\right)\int_{\eta=-b/2}^{b/2}\frac{d\Gamma(\eta)/d\eta}{y-\eta}\,d\eta = -\frac{\Gamma_0}{\pi b^2}\left(\frac{\pi A}{R_2}\right)\int_{\eta=-b/2}^{b/2}\frac{\eta}{y-\eta}\left[1-\left(\frac{2\eta}{b}\right)^2\right]^{-1/2}d\eta \tag{4.6.48}$$

which, following the same solution procedure as before (see Eqs. (4.3.17) and (4.3.18)), reduces to

$$\left(w_i\right)_{\text{elliptic}} = \frac{\Gamma_0}{2b}\left(\frac{\pi A}{R_2}\right) \tag{4.6.49}$$

The effective downwash, following the same reasoning as was given with Eqs. (4.3.20)-(4.3.22), is then

$$\left(w_e\right)_{\text{elliptic}} = \frac{\pi\Gamma_0}{2a_0\bar{c}}\left(\frac{a_0}{R_1}\right) \tag{4.6.50}$$

In applying the elliptic chord distribution from Eq. (4.3.21) here, we have relied on the assumptions that both a_0 and R_1 are constant over the wingspan. Substituting the right hand sides of Eqs. (4.6.49) and (4.6.50) into Eq. (4.3.2) and solving for Γ_0, we get

$$\Gamma_0 = \frac{2wb}{\pi A}\left(\frac{1}{R_1} + \frac{1}{R_2}\right) \tag{4.6.51}$$

Substitution of Eq. (4.6.51) into Eqs. (4.6.49) and (4.6.50) now gives

$$w_i = \frac{w}{1 + R_2/R_1} \tag{4.6.52}$$

$$w_e = \frac{w}{1 + R_1/R_2} \tag{4.6.53}$$

respectively; or, by Eq. (4.3.1),

$$\alpha_i = \frac{\alpha - \alpha_{L0}}{1 + R_2/R_1} \tag{4.6.54}$$

$$\alpha_e = \frac{\alpha - \alpha_{L0}}{1 + R_1/R_2} \tag{4.6.55}$$

The total lift coefficient of the wing is found by integrating Eq. (4.2.4) over the wingspan with the effective angle of attack given by Eq. (4.6.55). This gives

$$C_L = a_0\alpha_e = \frac{a_0}{1 + R_1/R_2}\left(\alpha - \alpha_{L0}\right) \tag{4.6.56}$$

and the effective wing lift slope becomes

$$a = \frac{a_0}{1 + R_1/R_2} \tag{4.6.57}$$

Equation (4.6.56) can be rewritten in terms of induced angle of attack as

$$C_L = a_0\alpha_i\left(\frac{R_2}{R_1}\right) \tag{4.6.58}$$

so that the drag coefficient is now

$$C_{D_i} = \frac{C_L^2}{a_0}\left(\frac{R_1}{R_2}\right) \tag{4.6.59}$$

in contrast to Eq. (4.3.33). Note that Eqs. (4.6.58) and (4.6.59) reduce to Eqs. (4.3.32) and (4.3.33) when the resistance values from classical lifting line theory – see Eq. (4.6.3) – are used. Similar relations can be obtained for the lift and drag coefficients based on the other low-aspect-ratio formulations discussed above.

We can derive an aspect ratio efficiency factor similar to the span efficiency factor discussed in Sec. 4.3 by direct comparison of Eqs. (4.6.59) and (4.3.34). The aspect ratio efficiency factor, e_A, is then

$$e_A = \frac{a_0}{\pi A}\frac{R_2}{R_1} \tag{4.6.60}$$

Note, however, that this efficiency factor only considers the effects of aspect ratio and does not account for deviations from the elliptic lift distribution. It is therefore not a replacement of the span efficiency factor so that our generalized induced drag equation becomes

$$C_{D_i} = \frac{C_L^2}{\pi A e_s e_A} \tag{4.6.61}$$

where $e_A \to 1$ as $A \to \infty$ (from classical lifting line theory) and $e_A \to 1/2$ as $A \to 0$ (from the modified slender wing equation). The span efficiency factor e_s has the same properties as in Eq. (4.3.34).

While all of the formulations discussed above were developed specifically for untwisted elliptic wings, Eq. (4.6.47) allows us to consider these same relations for other planforms and twist distributions, under the assumption that the effects of aspect ratio are unchanged for these other configurations. We have also assumed in the development of Eq. (4.6.47) that the effects of aspect ratio are felt equally along the wingspan. This last assumption could be removed, however, by allowing R_1 and R_2 to be functions of spanwise location. Doing so, however, would affect the result of the integration so that Eq. (4.6.57) could no longer be used to find the effective wing lift slope a. In this case a comprehensive solution of Eq. (4.6.47), either analytical or numerical, would be required.

Comparing Eq. (4.6.47) with Eqs. (4.3.14) and (4.4.16) gives the exact values for R_1 and R_2 in the limits as $A \to \infty$ and $A \to 0$. These values are summarized in Table 4.1. Note that by Eq. (4.3.21), $R_1 = \text{const} = a_0$ for an elliptic wing. While the modified slender wing equation and the equations of Helmbold [89], Hauptman and Miloh [83] (assuming a section lift slope of $a_0 = 2\pi$), and Küchemann [73] have the correct limits for wing lift slope, none of them match all of the limits given in Table 4.1 for R_1 and R_2.

Table 4.1 Summary of limits for R_1 and R_2

Parameter	$\lim\limits_{A\to\infty}$	$\lim\limits_{A\to 0}$
R_1	a_0	$a_0 \dfrac{4\bar{c}}{\pi c(y)}\sqrt{1-\left(\dfrac{2y}{b}\right)^2}$
R_2	πA	$\dfrac{\pi A}{2}$

It is a simple matter to conceive of other possible forms for R_1 and R_2 than have already been presented. An alternative equation is here proposed by the author which is both simple and accurate, though developed empirically. It is given in the form of Eq. (4.6.4) as

$$a_{Hodson} = \left\{ \frac{1}{a_0} + \frac{1}{A\left[\pi - \tan^{-1}\left(\dfrac{2a_0}{\pi A}\right)\right]} \right\}^{-1} \tag{4.6.62}$$

which gives

$$R_1 = a_0 \tag{4.6.63}$$

$$R_2 = A\left[\pi - \tan^{-1}\left(\frac{2a_0}{\pi A}\right)\right] \tag{4.6.64}$$

For an elliptic wing, this equation meets all four limit requirements from Table 4.1.

4.7 Results and Discussion

Table 4.2 provides a summary of the equations discussed in the previous section, which includes the values for wing lift slope predicted by these equations in the upper and lower limits of aspect ratio. Tables 4.3 and 4.4 summarize the resistance values to be used in Eqs. (4.6.4) and (4.6.47) for each of the methods considered, including their respective limiting values. While the modified slender wing equation and the equations of Helmbold [89], Hauptman and Miloh [83] (assuming a section lift slope of $a_0 = 2\pi$), and Küchemann [73] have the correct limits for wing lift slope, none of them match all of the limits given in Table 4.1 for R_1 and R_2. Only Eq. (4.6.62) achieves all the correct limits from Table 4.1. We note, however, that Küchemann [73] does match all the correct limits if the lower limits are taken from Jones' [79] slender wing theory given by Eq. (4.4.12) as opposed to the modified slender wing equation given by Eq. (4.4.16).

Table 4.2 Comparison of methods for calculating the wing lift slope of a finite elliptic wing

Method	**Equation**	**Wing Lift Slope (a)**	$\lim_{A\to\infty} a$	$\lim_{A\to 0} a$
Classical lifting line theory [10,11]	Eq. (4.6.3)	$a_{\text{classical}} = \left[1/a_0 + 1/(\pi A)\right]^{-1}$	a_0	πA
Slender wing theory [79]	Eq. (4.6.7)	$a_{\text{slender}} = \pi A/2$	$\pi A/2$	$\pi A/2$
Modified slender wing equation	Eq. (4.6.9)	$a_{\text{modified\_slender}} = \left[1/a_0 + 2/(\pi A)\right]^{-1}$	a_0	$\pi A/2$
Helmbold [89]	Eq. (4.6.12)	$a_{\text{Helmbold}} = \dfrac{a_0}{\sqrt{1 + a_0^{\,2}/(\pi A)^2} + a_0/\pi A}$	a_0	$\pi A/2$
Jones [88]	Eq. (4.6.16)	$a_{\text{Jones}} = a_0 \dfrac{A}{EA + a_0/\pi}$	a_0	$\dfrac{\pi a_0 A}{\pi + a_0}$
Van Dyke [90]	Eq. (4.6.20)	$a_{\text{VanDyke}} = \dfrac{a_0}{1 + a_0/\pi A + \left[2a_0/(\pi^2 A)\right]^2 \left[\ln(\pi A) - \frac{9}{8}\right]}$	a_0	$\dfrac{\pi^4 A^2}{4a_0 \ln(\pi A)}$
Germain [91]	Eq. (4.6.24)	$a_{\text{Germain}} = \dfrac{a_0}{1 + \dfrac{a_0}{\pi A} + \dfrac{4a_0^{\,2}}{(\pi A)^2}\left[\ln\left(1 + \pi e^{-9/8} A\right)\right]}$	a_0	$\dfrac{\pi^2 A}{\pi + 8e^{-9/8}}$
Hauptman & Miloh [83]	Eq. (4.6.27)	$a_{\text{Hauptman\&Miloh}} = \begin{cases} \dfrac{4k}{1 + \frac{E^2(h)}{1 + \left(k^2/h\right)\ln(1/k + h/k)}}, & k = \dfrac{\pi A}{4} \le 1 \\ \dfrac{4}{k + \frac{E^2(h)}{k + \sin^{-1}(h)/h}}, & k = \dfrac{4}{\pi A} \le 1 \end{cases}$	2π	$\pi A/2$
Küchemann [73]	Eq. (4.6.43)	$a_{\text{Küchemann}} = \dfrac{2na_0}{1 - \pi n \cot(\pi n) + 4n^2 a_0/(\pi A)}, \quad n = 1 - \dfrac{1}{2}\left[1 + \left(\dfrac{a_0}{\pi A}\right)^2\right]^{-1/4}$	a_0	$\pi A/2$
Hodson	Eq. (4.6.62)	$a_{Hodson} = \left\{1/a_0 + 1\Big/ A\left[\pi - \tan^{-1}\left(\dfrac{2a_0}{\pi A}\right)\right]\right\}^{-1}$	a_0	$\pi A/2$

Table 4.3 Comparison of resistance values R_1 from methods for calculating the wing lift slope of a finite elliptic wing

Method	Equation	R_1	$\lim\limits_{A\to\infty} R_1$	$\lim\limits_{A\to 0} R_1$
Classical lifting line theory [10,11]	Eq. (4.6.3)	a_0	a_0	a_0
Slender wing theory [79]	Eq. (4.6.7)	∞	∞	∞
Modified slender wing equation	Eq. (4.6.9)	a_0	a_0	a_0
Helmbold [89]	Eq. (4.6.13)	$\left[\dfrac{1}{a_0^{\,2}}+\dfrac{1}{(\pi A)^2}\right]^{-1/2}$	a_0	πA
Jones [88]	Eq. (4.6.17)	a_0/E	a_0	$a_0 A$
Van Dyke [90]	Eq. (4.6.21)	a_0	a_0	a_0
Germain [91]	Eq. (4.6.25)	a_0	a_0	a_0
Hauptman & Miloh [83]	Eq. (4.6.28)	$\begin{cases} \dfrac{\pi A+\left(4k^3/h\right)\ln\left(1/k+h/k\right)}{E^2(h)}, & k=\dfrac{\pi A}{4}\le 1 \\ \dfrac{4\left(k+\sin^{-1}(h)/h\right)}{E^2(h)}, & k=\dfrac{4}{\pi A}\le 1 \end{cases}$	2π	πA
Küchemann [73]	Eq. (4.6.44)	$\dfrac{2na_0}{1-\pi n\cot(\pi n)}\ ,\ n=1-\dfrac{1}{2}\left[1+\left(\dfrac{a_0}{\pi A}\right)^2\right]^{-1/4}$	a_0	0
Hodson	Eq. (4.6.62)	a_0	a_0	a_0

Table 4.4 Comparison of resistance values R_2 from methods for calculating the wing lift slope of a finite elliptic wing

Method	**Equation**	R_2	$\lim_{A\to\infty} R_2$	$\lim_{A\to 0} R_2$
Classical lifting line theory [10,11]	Eq. (4.6.3)	πA	πA	πA
Slender wing theory [79]	Eq. (4.6.7)	$\pi A/2$	$\pi A/2$	$\pi A/2$
Modified slender wing equation	Eq. (4.6.9)	$\pi A/2$	$\pi A/2$	$\pi A/2$
Helmbold [89]	Eq. (4.6.13)	πA	πA	πA
Jones [88]	Eq. (4.6.17)	πA	πA	πA
Van Dyke [90]	Eq. (4.6.21)	$\dfrac{\pi A}{1+\dfrac{4a_0}{\pi^3 A}\left[\ln(\pi A)-\dfrac{9}{8}\right]}$	πA	$\dfrac{\pi^4 A^2}{4a_0 \ln(\pi A)}$
Germain [91]	Eq. (4.6.25)	$\dfrac{\pi A}{1+\dfrac{4a_0}{\pi^3 A}\ln\left(1+\pi e^{-9/8}A\right)}$	πA	$\dfrac{\pi^2 A}{\pi+8e^{-9/8}}$
Hauptman & Miloh [83]	Eq. (4.6.28)	πA	πA	πA
Küchemann [73]	Eq. (4.6.44)	$\dfrac{\pi A}{2n}$	πA	$\pi A/2$
Hodson	Eq. (4.6.62)	$A\left[\pi-\tan^{-1}\left(\dfrac{2a_0}{\pi A}\right)\right]$	πA	$\pi A/2$

Figure 4.3 compares the wing lift slope calculations from classical lifting line theory, slender wing theory, the modified slender wing equation, Helmbold [89], Küchemann [73], Hauptman and Miloh [83], and the current author with numerical lifting surface results from Kinner [94], Krienes [95], Jordan [96], and Medan [97]. Also included are high-order panel method results computed by the author using Panair [98–100] and described in Appendix H. In these calculations we have modeled an elliptic wing with a uniform, symmetric cross section having a section lift slope of $a_0 = 2\pi$.

While the equations of Helmbold [89] and Küchemann [73] show reasonable agreement with the numerical results presented in Figure 4.3, the equations of Hauptman and Miloh [83] and the present author perform better. However, Eq. (4.6.27) from Hauptman and Miloh [83] is a piecewise formulation that requires a solution of the complete elliptic integral of the second kind. These additional complexities in Eq. (4.6.27) make Eq. (4.6.62) a more reasonable choice for any practical applications of the solution method discussed herein.

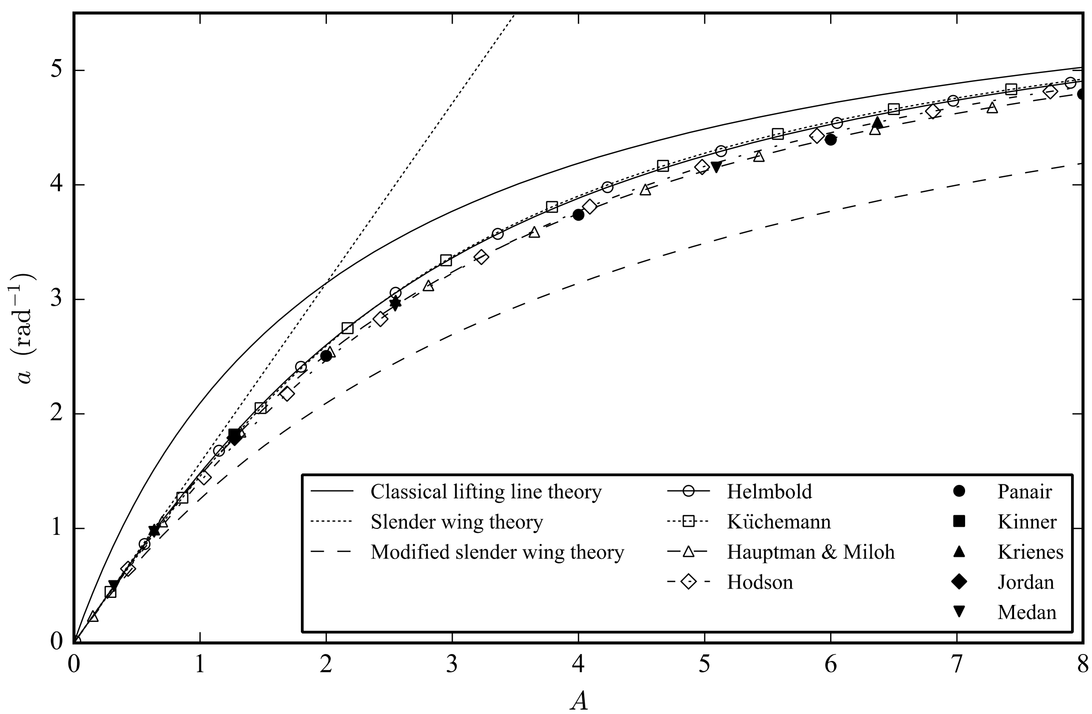


Figure 4.3 Comparison of lift slope calculation methods applied to a finite elliptic wing

It is necessary to remember that the equations presented above and summarized in Table 4.2 have been developed specifically for wings of elliptic planform according to Eq. (4.3.21). For arbitrary planforms, the correct value of R_1 must be dependent on spanwise location at low aspect ratio, and the rate at which R_1 transitions between the upper and lower limits cannot be determined from a study of elliptic planforms. Since most wing designs are not elliptic, it is of interest to the present study to consider the validity of the proposed equations to other planform designs. Figures 4.4 and 4.5 provide comparisons of solutions to Eq. (4.6.47) for rectangular and tapered planforms using the resistance values from classical lifting line theory, the modified slender wing equation, and Eqs. (4.6.63)-(4.6.64). Results computed using Panair are also included. Solutions

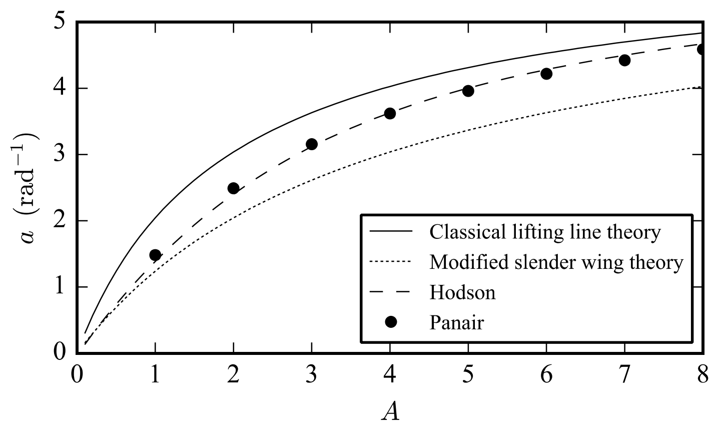


Figure 4.4 Comparison of lift slope calculation methods applied to a finite rectangular wing

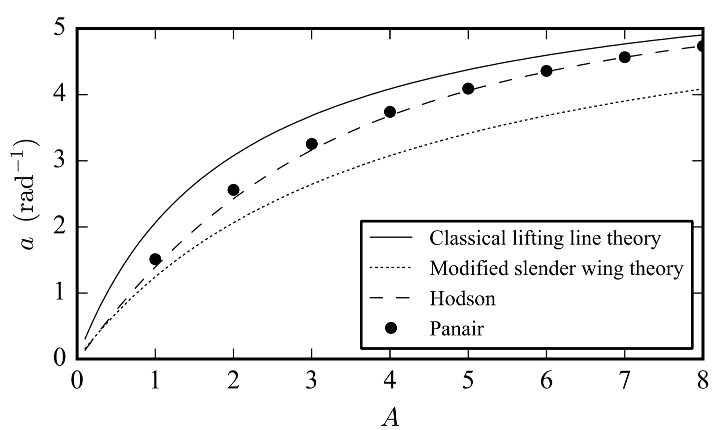


Figure 4.5 Comparison of lift slope calculation methods applied to a tapered wing with $R_T = 0.75$

to Eq. (4.6.47) were obtained using Pralines,* an open-source Fortran code developed by the author. This code is based on the solution procedure presented by Phillips [53], in which Eq. (4.6.47) is approximated as a truncated Fourier sine series. The source code is given in Appendix K.

Despite having been developed specifically for elliptic planforms, the proposed equation shows excellent agreement with the Panair results for the rectangular and tapered planforms considered. It outperforms classical lifting line theory over the full range of aspect ratios for both rectangular and tapered cases.

At this point we have only considered predictions of the total wing lift slope of a finite wing without regard to the spanwise lift distribution. In fact the development presented here has assumed that the influence of aspect ratio beyond that already captured in classical lifting line theory is felt uniformly along the wingspan. Pralines can be used to scrutinize this assumption. For an elliptic wing, Eq. (4.6.47) predicts an elliptic lift distribution (i.e. a constant section lift coefficient), regardless of the resistance values used. This is shown and compared to Panair results in Figure 4.6 for a selection of aspect ratios. The results computed using Eqs. (4.6.63) and (4.6.64) in Eq. (4.6.47) show good agreement with Panair results for $y/b < 0.4$ regardless of aspect ratio. As expected, larger discrepancies are seen at the wing tips. However, the magnitudes of the discrepancies shown in Figure 4.6 can be misleading since they are scaled by the inverse of the chord length and the chord length goes to zero at the wing tip. Figure 4.7 shows the same data but with the spanwise lift coefficients multiplied by the local chord ratio $c/\bar{c}$, which more accurately reflects the relative influence of each spanwise value on the overall wing lift coefficient. From Figure 4.7 we see that the results computed using Eqs. (4.6.63) and (4.6.64) provide excellent agreement with the Panair results over the entire wingspan. There is a slight outward shift of lift when compared to the Panair results, but overall this shift is small – less than half a percent of the total lift generated by the wing.

Figures 4.8 and 4.9 compare lifting line theory and Panair results for rectangular and tapered planforms. The lift coefficients have again been scaled by the local chord ratio $c/\bar{c}$, as in Figure 4.7, to better visualize the influence of the spanwise values on the overall lift generated by the wing. The lifting line results based on the resistance values given by Eqs. (4.6.63) and (4.6.64) again do an excellent job of matching the Panair results, though the discrepancies are slightly larger for rectangular and tapered wings than for elliptic wings.

---

* https://github.com/joddlehod/pralines

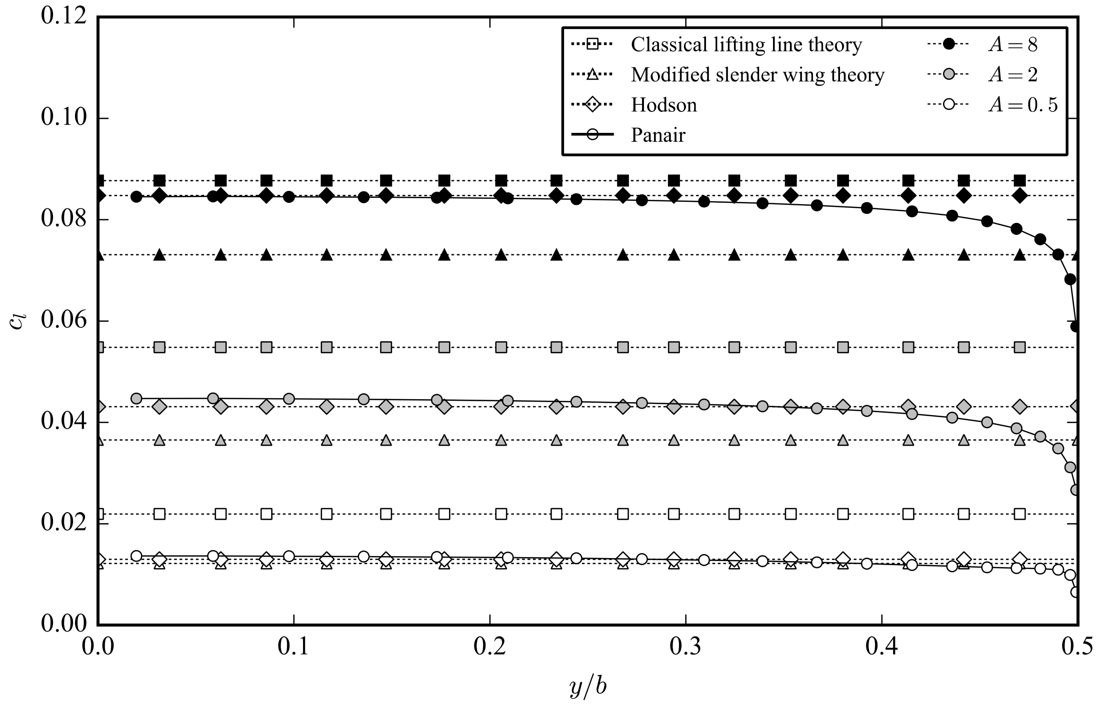


Figure 4.6 Comparison of spanwise lift coefficient distributions for elliptic wings.

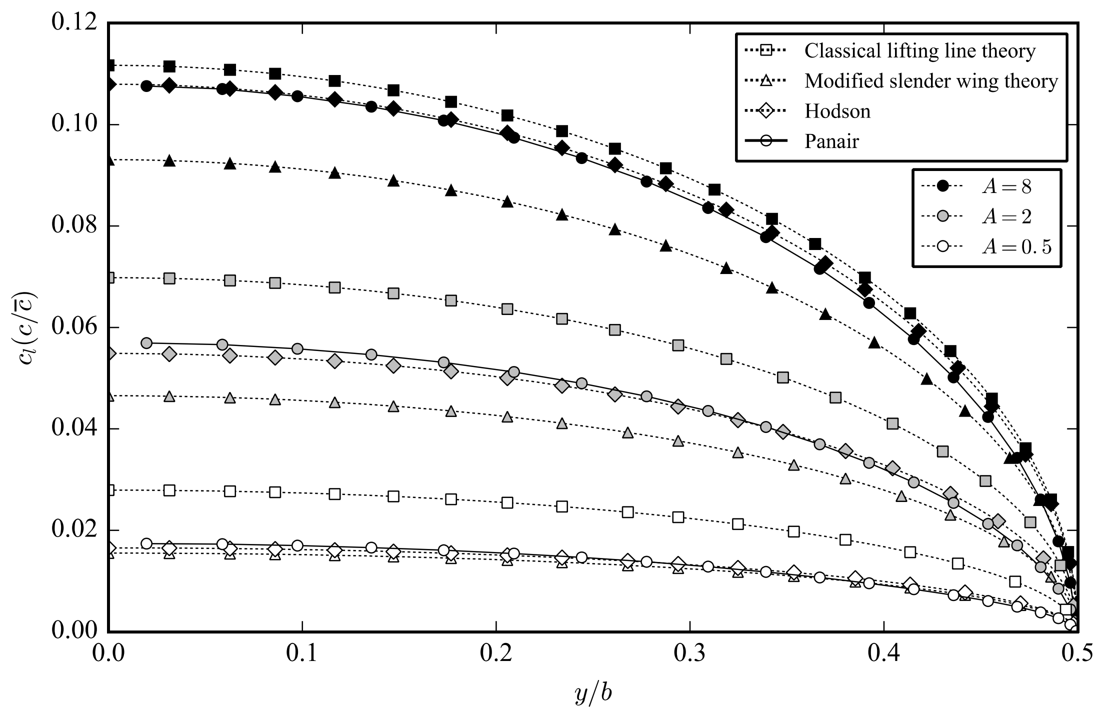


Figure 4.7 Comparison of spanwise lift distributions for elliptic wings.

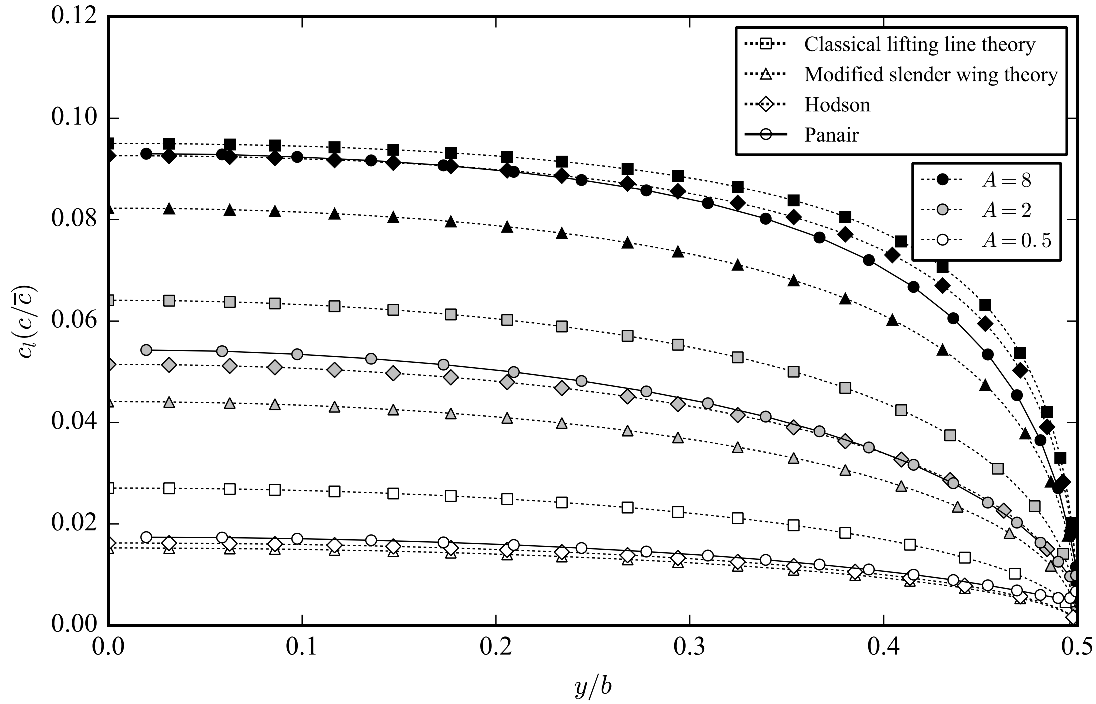


Figure 4.8 Comparison of spanwise lift distributions for rectangular wings.

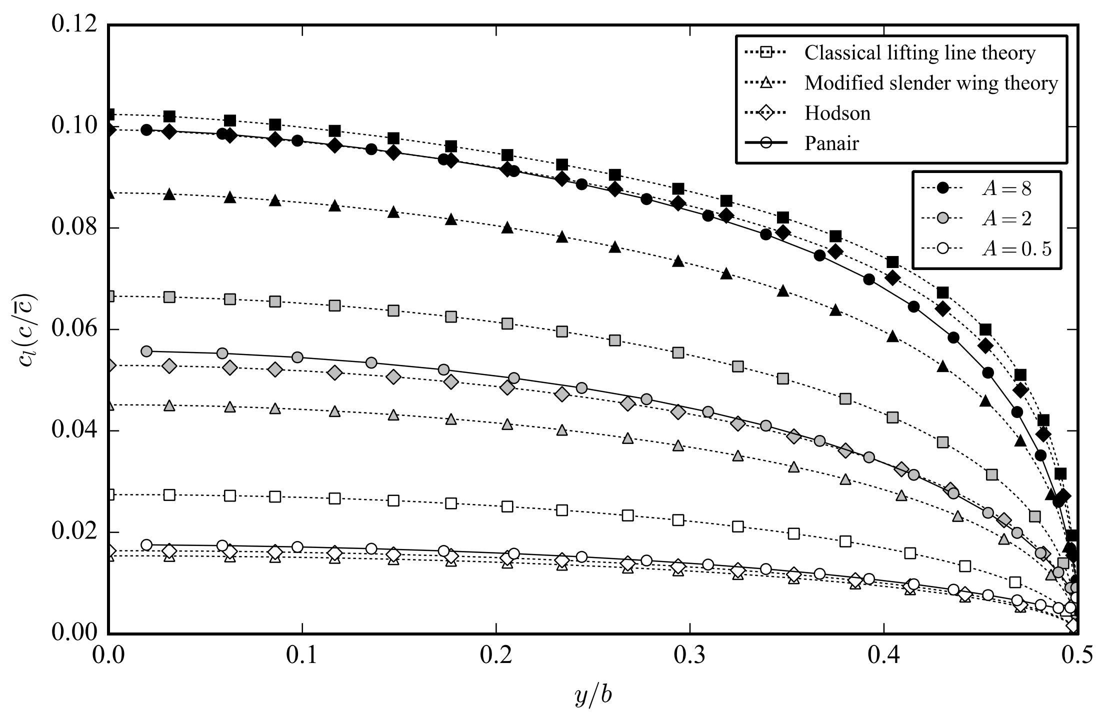


Figure 4.9 Comparison of spanwise lift distributions for tapered wings with $R_T = 0.75$.

Finally, we wish to consider how the proposed lifting line formulation affects induced drag calculations. Here we used the same codes, namely Pralines and Panair, to evaluate the lift and drag coefficients of several wings of varying aspect ratios, and then apply Eq. (4.6.61) to compute the aspect ratio efficiency factor for each analysis. These results, along with direct calculations of the aspect ratio efficiency factor using Eq. (4.6.60), are presented in Figure 4.10.

Some difficulty was encountered in obtaining accurate drag results from Panair. In computing lift results, we computed the solution on three different grid sizes and applied Richardson Extrapolation to improve the accuracy of the results. This method could not be applied to the drag results, however, because the results from successively refined grid sizes did not exhibit asymptotic convergence. Instead, each Panair data point presented in Figure 4.10 was computed from a single Panair analysis using a 4%-thick symmetric Joukowski

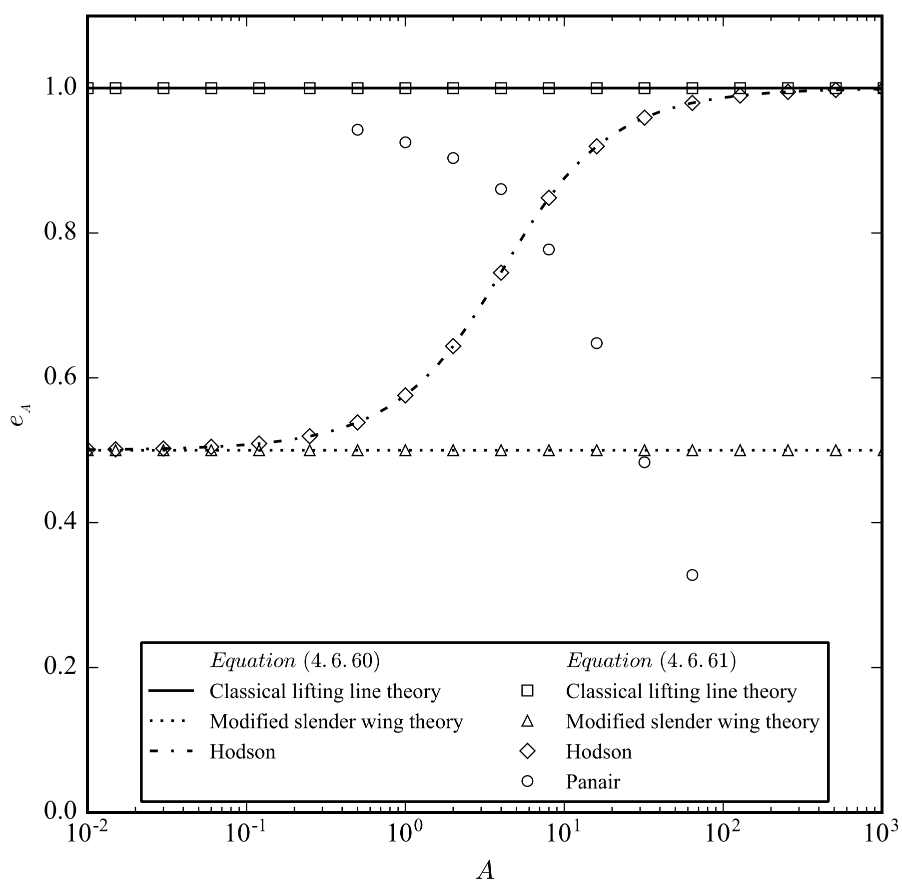


Figure 4.10 Sample calculations of the aspect ratio efficiency factor using Pralines and Panair.

airfoil for the cross section and an 80×80 grid size. This, however, still did not yield satisfactory results. The Panair data show an inverse trend in e_A to what is expected, indicating that as aspect ratio increases, both α_e and α_i increase. This would seem to violate the boundary condition given by Eq. (4.3.2), bringing into question the validity of the drag calculations produced by Panair.

In contrast, the lifting line results shown in Figure 4.10 agree well with the theory presented in this chapter. The results using classical lifting line theory give an aspect ratio efficiency factor of $e_A = 1$ regardless of aspect ratio, while the modified slender wing equation gives an aspect ratio efficiency factor of $e_A = 1/2$. These represent the theoretical upper and lower limits, respectively, of the aspect ratio efficiency factor. The equation proposed by the present author – see Eqs. (4.6.62)-(4.6.64) – shows a smooth transition between the two limits. While the shape of this transition cannot be confirmed due to the unreliability of the Panair drag results, it can be assumed that the shape is reasonable as it satisfies the boundary condition given by Eq. (4.3.2) and the corresponding lift values have already been shown to be satisfactory. Any alternative would require the influence of an aerodynamic phenomenon that has not been considered here or in the other works reviewed.

5 NUMERICAL LIFTING LINE METHOD FOR WINGS OF ARBITRARY ASPECT RATIO

5.1 Introduction

In the previous chapter, several equations for the wing lift slope of finite wings have been considered, and a method for modifying classical lifting line theory, Eq. (4.3.14), to match these equations has been presented. This formulation, however, is inapplicable to wings with sweep or dihedral, and is also limited to single isolated lifting surfaces. The numerical lifting line method of Phillips and Snyder [16] is based heavily on classical lifting line theory, but presents a formulation that allows for consideration of sweep, dihedral, and interactions between multiple lifting surfaces. In their original presentation of this method, Phillips and Snyder [16] showed the accuracy to be comparable to numerical panel methods and inviscid computational fluid dynamics solutions, but at a small fraction of the computational cost. It would be advantageous, therefore, to be able to extend the applicability of this numerical formulation to lifting surfaces of low aspect ratio, as was done for classical lifting line theory in the previous section. To do so, we begin with a comprehensive derivation of the method as originally presented by Phillips and Snyder [16]. We will then discuss the necessary modifications to this method to achieve our purpose.

As with classical lifting line theory, a lifting surface in numerical lifting line theory is represented as a series of horseshoe vortices, each horseshoe vortex consisting of a single spanwise vortex segment coincident with the quarter-chord of the wing and two semi-infinite vortex segments extending chordwise downstream. However, in contrast to the overlapping horseshoe vortices shown in Figure 4.1, the lifting surface is composed of non-overlapping, side-by-side horseshoe vortices as shown in Figure 5.1. While a gap is shown between adjacent horseshoe vortices in the figure, this is simply for illustration purposes. In reality, the incoming and outgoing vortex segments of adjacent horseshoe vortices are coincident, and vorticity is shed from the wing at each of these interfaces in an amount equal to the difference in vortex strengths between the two adjacent horseshoe vortices that comprise the interface. While this simple change in arrangement of the horseshoe vortices seems trivial, it affords some significant advantages over the arrangement used in classical lifting line theory. Each horseshoe vortex is now tied to a specific section of the finite wing being modeled, so that the properties of each section are now decoupled from one another. For example, the sweep, dihedral,

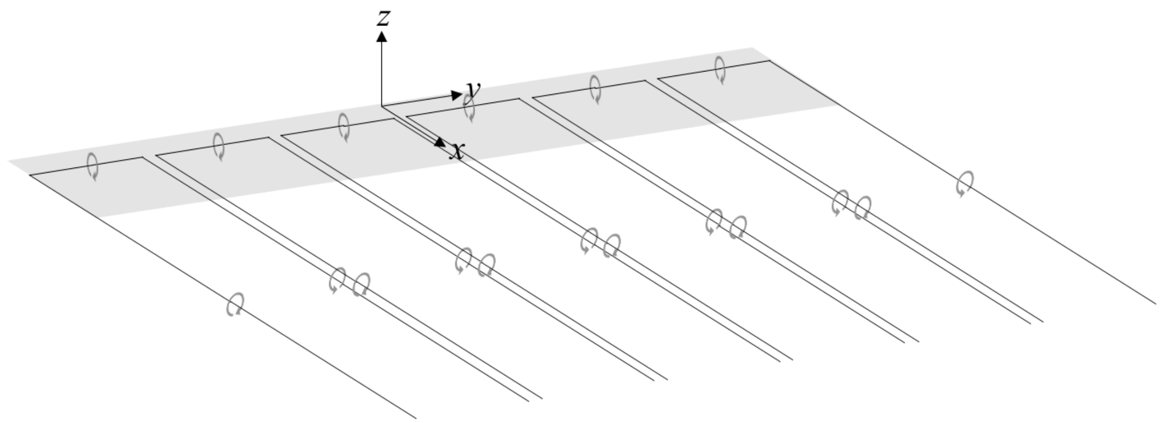


Figure 5.1 Discrete system of side-by-side horseshoe vortices on a finite wing.

and section lift slope (a_0) can all be specified individually for each section of the wing. In classical lifting line theory, where the largest horseshoe vortex spans from wing tip to wing tip, each of these values is required to be constant over the entire wing.

The fundamental problem we must solve with this new arrangement is the same as that of classical lifting line theory – namely, to determine the vortex strengths $\Gamma(y)$ such that the boundary condition of Eq. (4.3.1) is satisfied. We shall follow the same general development as was done for classical lifting line theory, but using a more general three-dimensional formulation due to the new arrangement of horseshoe vortices. This complete development can be found in its original form in Phillips and Snyder [16]. We shall then consider how the low aspect ratio modifications presented in the previous chapter can be applied to this numerical method. Finally, we shall compare results from the modified method with those of other approaches.

5.2 The Phillips and Snyder Numerical Lifting Line Method

Consider the arbitrary vortex segment $\overline{OR}$ shown in Figure 5.2. Let $\mathbf{l}$ be the vector from point O to point R, and consider the vortex element $\mathbf{dl}$ located at point Q. The velocity induced at point P by this differential vortex element is given by the Biot-Savart law,

$$\mathbf{dV} = \frac{\Gamma}{4\pi}\frac{\mathbf{dl}\times\mathbf{r}}{r^3} \tag{5.2.1}$$

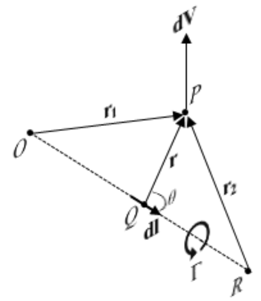


Figure 5.2 Velocity induced at point *P* by an arbitrary vortex segment $\overline{OR}$.

Let ζ be the ratio of the lengths of the line segments $\overline{OQ}$ and $\overline{OR}$. The vector **r** can be expressed as the difference of the two vectors $\mathbf{r}_1$ and $\zeta\mathbf{l}$, so that the magnitude is given by

$$r = \sqrt{r_1^2 - 2\zeta\mathbf{r}_1 \cdot \mathbf{l} + \zeta^2 l^2} \tag{5.2.2}$$

and the cross product in Eq. (5.2.1) can be rewritten as

$$\mathbf{dl} \times \mathbf{r} = (\mathbf{l}d\zeta) \times (\mathbf{r}_1 - \zeta\mathbf{l}) = (\mathbf{l} \times \mathbf{r}_1 - \zeta\mathbf{l} \times \mathbf{l})\, d\zeta = \mathbf{l} \times \mathbf{r}_1 d\zeta \tag{5.2.3}$$

Using Eqs. (5.2.2) and (5.2.3) in Eq. (5.2.1), we get

$$\mathbf{dV} = \frac{\Gamma}{4\pi} \frac{\mathbf{l} \times \mathbf{r}_1 d\zeta}{\left(r_1^2 - 2\zeta\mathbf{r}_1 \cdot \mathbf{l} + \zeta^2 l^2\right)^{3/2}} \tag{5.2.4}$$

The total velocity induced at point *P* by the vortex segment $\overline{OR}$ is then found by integrating Eq. (5.2.4) over the length of the vortex segment,

$$\mathbf{V} = \frac{\Gamma(\mathbf{l} \times \mathbf{r}_1)}{4\pi} \int_0^1 \frac{d\zeta}{\left(r_1^2 - 2\zeta\mathbf{r}_1 \cdot \mathbf{l} + \zeta^2 l^2\right)^{3/2}} = \frac{\Gamma(\mathbf{r}_1 \times \mathbf{r}_2)(r_1 + r_2)}{4\pi r_1 r_2 (r_1 r_2 + \mathbf{r}_1 \cdot \mathbf{r}_2)} \tag{5.2.5}$$

A complete proof of the integral in Eq. (5.2.5) is provided in Appendix L. For a semi-infinite vortex with $r_2 \to \infty$ in the direction of the freestream, Eq. (5.2.5) becomes

$$\mathbf{V} = \frac{\Gamma(\mathbf{u}_\infty \times \mathbf{r}_1)}{4\pi r_1 (r_1 - \mathbf{u}_\infty \cdot \mathbf{r}_1)} \tag{5.2.6}$$

where $\mathbf{u}_\infty$ is the unit vector in the direction of the freestream. The velocity induced at an arbitrary point by a complete horseshoe vortex is then

$$\mathbf{V} = \frac{\Gamma}{4\pi}\left[\frac{\left(\mathbf{u}_\infty \times \mathbf{r}_2\right)}{r_2\left(r_2 - \mathbf{u}_\infty \cdot \mathbf{r}_2\right)} + \frac{\left(r_1 + r_2\right)\left(\mathbf{r}_1 \times \mathbf{r}_2\right)}{r_1 r_2\left(r_1 r_2 + \mathbf{r}_1 \cdot \mathbf{r}_2\right)} - \frac{\left(\mathbf{u}_\infty \times \mathbf{r}_1\right)}{r_1\left(r_1 - \mathbf{u}_\infty \cdot \mathbf{r}_1\right)}\right] \tag{5.2.7}$$

where $\mathbf{r}_1$ and $\mathbf{r}_2$ are now the vectors from the two corner points of the horseshoe vortex to the arbitrary point.

Now consider the discrete system of horseshoe vortices used to represent a finite wing as shown in Figure 5.1. A control point placed anywhere along the quarter-chord of the wing (i.e. the lifting line) will experience an induced velocity due to each horseshoe vortex according to Eq. (5.2.7). The induced downwash vector at the control point by a system of *n* horseshoe vortices is then given by

$$\mathbf{w}_i = \sum_{j=1}^{n}\frac{\Gamma_j}{4\pi}\left[\frac{\left(\mathbf{u}_\infty \times \mathbf{r}_{j2}\right)}{r_{j2}\left(r_{j2} - \mathbf{u}_\infty \cdot \mathbf{r}_{j2}\right)} + \frac{\left(r_{j1} + r_{j2}\right)\left(\mathbf{r}_{j1} \times \mathbf{r}_{j2}\right)}{r_{j1} r_{j2}\left(r_{j1} r_{j2} + \mathbf{r}_{j1} \cdot \mathbf{r}_{j2}\right)} - \frac{\left(\mathbf{u}_\infty \times \mathbf{r}_{j1}\right)}{r_{j1}\left(r_{j1} - \mathbf{u}_\infty \cdot \mathbf{r}_{j1}\right)}\right] \tag{5.2.8}$$

where $\mathbf{r}_{j1}$ and $\mathbf{r}_{j2}$ are the vectors from the two corner points of the *j*th horseshoe vortex to the control point. The velocity vector at the control point can now be expressed as the sum of the freestream velocity and the induced downwash, namely

$$\mathbf{V} = \mathbf{V}_\infty + \mathbf{w}_i \tag{5.2.9}$$

Note that the middle term within the square brackets of Eq. (5.2.8) is indeterminate when the angle between $\mathbf{r}_{j1}$ and $\mathbf{r}_{j2}$ is $\pm 180\,\mathrm{deg}$. However, this condition is only satisfied when the control point lies exactly on the axis of the vortex. Since the velocity induced by a straight vortex segment along its axis is zero, the appropriate value for this term is zero anytime $\mathbf{r}_{j1}$ and $\mathbf{r}_{j2}$ are coaxial.

In the development of classical lifting line theory, Prandtl [10,11] used the Kutta-Joukowski theorem given by Eq. (4.3.10) to relate the lift force to the circulation for any spanwise section of the wing. Phillips and Snyder [16] take a similar approach, but use the more general three-dimensional vortex lifting law (see Saffman [101]). This can be written for a differential segment $\mathbf{dl}$ of the lifting line as

$$\mathbf{dF} = \rho_\infty \Gamma \mathbf{V} \times \mathbf{dl} \tag{5.2.10}$$

Following Prandtl's development of classical lifting line theory, we can equate the magnitude of this spanwise differential force to the lift predicted by 2D airfoil theory, namely

$$\left|\mathbf{dF}\right| = \frac{1}{2}\rho_\infty V_\infty{}^2 a_0 \alpha_e dS \tag{5.2.11}$$

where

$$\alpha_e = \tan^{-1}\left(\frac{\mathbf{V}\cdot\mathbf{u}_n}{\mathbf{V}\cdot\mathbf{u}_a}\right) \tag{5.2.12}$$

is the effective angle of attack, $\mathbf{u}_n$ is the unit vector normal to the local airfoil chord, and $\mathbf{u}_a$ is the unit vector aligned axially with the local airfoil chord. Both vectors $\mathbf{u}_n$ and $\mathbf{u}_a$ lie within the section plane. Equating the right hand sides of Eqs. (5.2.10) and (5.2.11) gives

$$\rho_\infty \Gamma \left|\mathbf{V}\times\mathbf{dl}\right| = \frac{1}{2}\rho_\infty V_\infty{}^2 a_0 \alpha_e dS \tag{5.2.13}$$

Expressed in this form, Eq. (5.2.13) is a single equation with n unknowns, namely the vorticities $\Gamma_{1,2,\ldots,n}$ for each horseshoe vortex. The control point, since it has been constrained to lie along the lifting line, also lies along the axis of one of the n horseshoe vortices. The values of the parameters Γ, $\mathbf{V}$, $\mathbf{dl}$, α_e, and dS in Eq. (5.2.13) are associated with the wing section and corresponding horseshoe vortex along which the control point lies. By placing a control point on each of the n horseshoe vortices and writing Eq. (5.2.13) for each control point, we generate a system of n nonlinear equations in n unknowns that can be solved using an iterative root-finding algorithm such as the Newton-Raphson method.

In Secs. 4.3 and 4.6, we were able to develop closed-form relations for the lift and drag coefficients of wings with elliptic lift distributions. Development of similar equations using the numerical formulation of Phillips and Snyder [16] is not straightforward due to the discrete summation and the vector notation used in the formulation. Instead, a solver that implements this method can compute the resultant force vector at each control point and then compute a vector sum the forces over the wing. Lift and drag components relative to the freestream can then be computed using the vector dot product.

In the original paper by Phillips and Snyder [16], results using this method are shown to be equivalent to those of classical lifting line theory for straight elliptic wings of various aspect ratios. Additionally, results are shown to be in good agreement with experimental and high-order numerical results for straight wings, wings with sweep, and wings with dihedral. However, in all comparisons the aspect ratios are greater than 4. Figure 5.3 compares wing lift slope results computed using the numerical lifting line method of Phillips and Snyder with results from classical lifting line theory and a vortex panel method. Solutions to the numerical lifting line method were computed using MachUp (see Sec. 2.6.1) with 100 nodes per semispan and a section

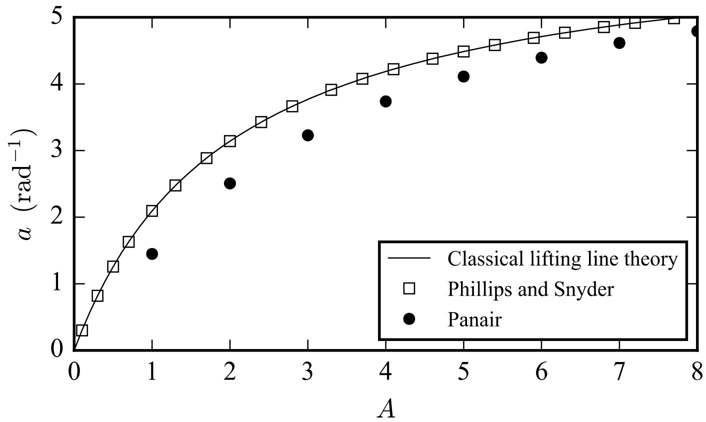


Figure 5.3 Comparison of wing lift slope calculations for finite elliptic wings.

lift slope of $a_0 = 2\pi$. Solutions to classical lifting line theory were computed using Eq. (4.6.3) with a section lift slope of $a_0 = 2\pi$. Vortex panel method results were computed using Panair [98–100]. From these data it is clear that the numerical lifting line method of Phillips and Snyder is subject to the same aspect ratio limitations as classical lifting line theory.

5.3 Modifications to the Numerical Lifting Line Method for Wings of Arbitrary Aspect Ratio

As stated in the previous section and illustrated in Figure 5.3, computations performed using the numerical lifting line method of Phillips and Snyder [16] are subject to the same aspect ratio limitations as classical lifting line theory. However, we have already developed a method for modifying classical lifting line theory to accurately account for aspect ratio (see Sec. 4.6). We shall now present a method for applying this same modification to Eq. (5.2.13).

First, we note that the effective angle of attack appears explicitly in Eq. (5.2.13). Solving for α_e gives

$$\alpha_e = \frac{2\Gamma|\mathbf{V}\times\mathbf{dl}|}{a_0 V_\infty dS} \tag{5.3.1}$$

which is quite similar to Eq. (4.3.13). In Eq. (4.6.47) a correction factor of (a_0/R_1) has been applied to the effective angle of attack. We therefore, by analogy, apply this same factor here so that Eq. (5.3.1) becomes

$$\alpha_e = \frac{2\Gamma|\mathbf{V}\times\mathbf{dl}|}{a_0 V_\infty dS}\left(\frac{a_0}{R_1}\right) \tag{5.3.2}$$

for arbitrary aspect ratios.

The induced angle of attack does not appear explicitly in Eq. (5.2.13), but the induced downwash given by Eq. (5.2.8) is used in the calculation of $\mathbf{V}$, which appears in the left hand side of Eq. (5.2.13) and in the definition for α_i given by Eq. (5.2.12). Since, by the small angle approximation,

$$\alpha_i = w_i / V_\infty \tag{5.3.3}$$

we conjecture that the induced downwash given by Eq. (5.2.8) must be scaled by the same factor as the induced angle of attack in Eq. (4.6.47), namely $(\pi A / R_2)$. This gives

$$\mathbf{w}_i = \left(\frac{\pi A}{R_2}\right) \sum_{j=1}^{N} \frac{\Gamma_j}{4\pi} \left[\frac{(\mathbf{u}_\infty \times \mathbf{r}_{j2})}{r_{j2}\left(r_{j2} - \mathbf{u}_\infty \cdot \mathbf{r}_{j2}\right)} + \frac{(r_{j1} + r_{j2})(\mathbf{r}_{j1} \times \mathbf{r}_{j2})}{r_{j1} r_{j2}\left(r_{j1} r_{j2} + \mathbf{r}_{j1} \cdot \mathbf{r}_{j2}\right)} - \frac{(\mathbf{u}_\infty \times \mathbf{r}_{j1})}{r_{j1}\left(r_{j1} - \mathbf{u}_\infty \cdot \mathbf{r}_{j1}\right)} \right] \tag{5.3.4}$$

These two simple changes are all that is required to allow the application of any one of the correction methods described in Sec. 4.6 to the numerical lifting line method of Phillips and Snyder [16]. The nature and complexity of the equations to be solved has not been changed, so that the same algorithms used to solve the unmodified equations can be used to solve the modified ones. For completeness, the equations to be solved for the modified numerical lifting line algorithm are summarized below.

$$\rho_\infty \Gamma \left| \mathbf{V} \times \mathbf{dl} \right| \left(\frac{a_0}{R_1}\right) = \frac{1}{2} \rho_\infty V_\infty^{\,2} a_0 \alpha_e dS \tag{5.3.5}$$

$$\alpha_e = \tan^{-1}\left(\frac{\mathbf{V} \cdot \mathbf{u}_n}{\mathbf{V} \cdot \mathbf{u}_a}\right) \tag{5.2.12}$$

$$\mathbf{V} = \mathbf{V}_\infty + \mathbf{w}_i \tag{5.2.9}$$

$$\mathbf{w}_i = \left(\frac{\pi A}{R_2}\right) \sum_{j=1}^{N} \frac{\Gamma_j}{4\pi} \left[\frac{(\mathbf{u}_\infty \times \mathbf{r}_{j2})}{r_{j2}\left(r_{j2} - \mathbf{u}_\infty \cdot \mathbf{r}_{j2}\right)} + \frac{(r_{j1} + r_{j2})(\mathbf{r}_{j1} \times \mathbf{r}_{j2})}{r_{j1} r_{j2}\left(r_{j1} r_{j2} + \mathbf{r}_{j1} \cdot \mathbf{r}_{j2}\right)} - \frac{(\mathbf{u}_\infty \times \mathbf{r}_{j1})}{r_{j1}\left(r_{j1} - \mathbf{u}_\infty \cdot \mathbf{r}_{j1}\right)} \right] \tag{5.3.4}$$

As was mentioned in the previous section, closed-form expressions for lift and drag coefficients of wings with elliptic lift distributions are not available based on this numerical formulation, so that they must be computed as a vector summation of the resultant force vectors at each control point. This procedure is the same for the modified formulation described here as for the original formulation of Phillips and Snyder [16] discussed in the previous section, so that the additional rotation of the resultant force vector due to the modifications presented in this section will be automatically accounted for in any application that correctly implements this method.

5.4 Results and Discussion

In order to evaluate the effectiveness of the modifications described above, the MachUp source code has been modified to include these changes, and an additional parameter has been added to the input file format to allow selection of the resistance values to be used in an analysis. Figure 5.4 shows a comparison of wing lift slopes computed using this modified version of MachUp with the analytical solutions from Chapter 4 for classical lifting line theory, the modified slender wing equation, and Eqs. (4.6.63)-(4.6.64). Results computed using Panair are also included.

The numerical model for these calculations was composed of 100 spanwise sections for one semispan with a symmetry boundary condition at the wing root. The uniform cross section was modeled as a thin symmetric airfoil with $a_0 = 2\pi$. The iterative nonlinear solver was used with a convergence tolerance of 10^{-10}. With these settings, the numerical results agree with the analytical results to at least four digits of precision. Additionally, the differences between the Panair results and the numerical results based on the formulation proposed by the present author are less than 1% for $A \geq 1$.

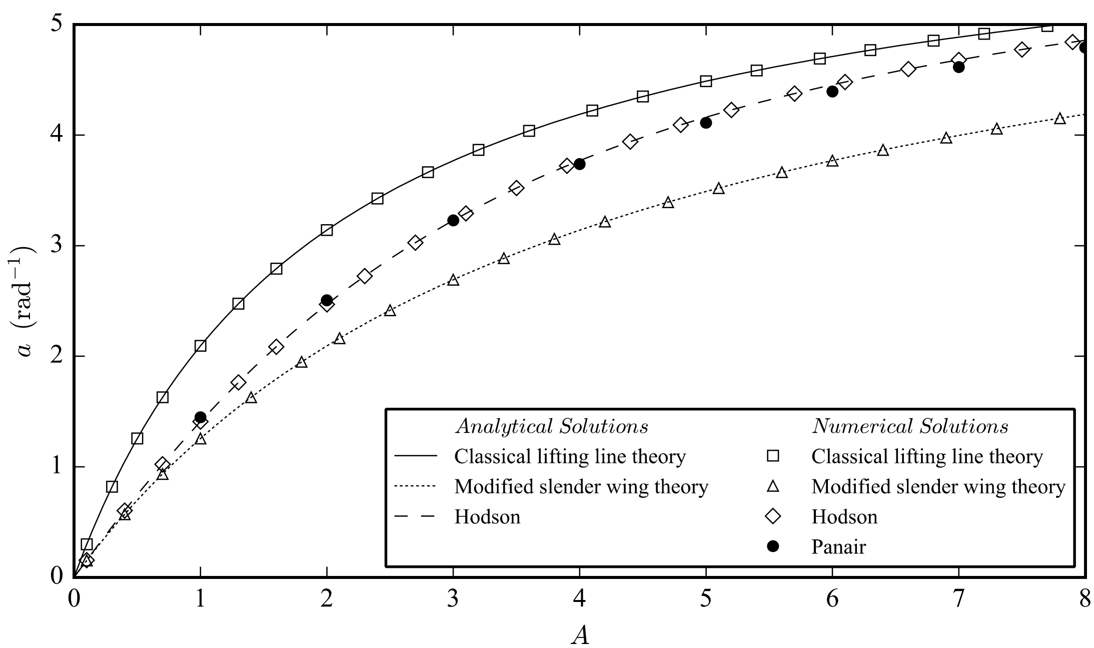


Figure 5.4 Comparison of analytical and numerical calculations for wing lift slope of elliptic wings.

Figures 5.5-5.7 compare the lift distributions for elliptic, rectangular, and tapered planforms predicted by the modified MachUp code to those predicted by Panair. The results are essentially identical to the analytical results shown in Figures 4.7-4.9. From these data we conclude that the modified equations described in Sec. 5.3 are the numerical equivalent to the analytical lifting line formulation given in Eq. (4.6.47).

This work has considered only isolated wings with no sweep; no geometric or aerodynamic twist; no dihedral; and only elliptic, rectangular, and tapered planforms. The formulation given in Sec. 5.3, however, is not limited to any of these constraints. Previous publications (see Refs. [12–14]) have applied the numerical lifting line algorithm described in Sec.5.2 to a variety of other wing designs with reasonable success. While beyond the scope of the present work, it is expected that the proposed method can produce useful results for arbitrary wing designs.

Figure 5.8 shows aspect ratio efficiency factors computed using MachUp and Panair. The Panair results were discussed in Sec. 4.7. The MachUp results are equivalent to the Pralines results shown in Figure 4.10. This provides added confidence in the derivations leading to Eq. (4.6.59) since that equation is used directly to calculate drag in Pralines but not in MachUp.

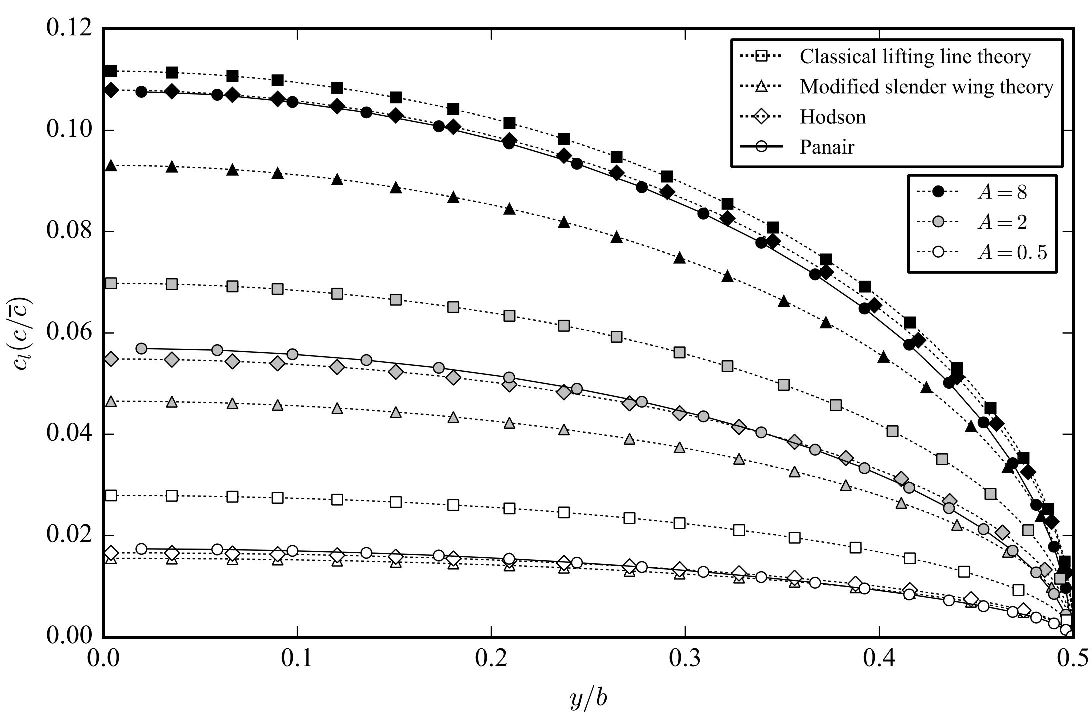


Figure 5.5 Comparison of spanwise lift distributions for elliptic wings.

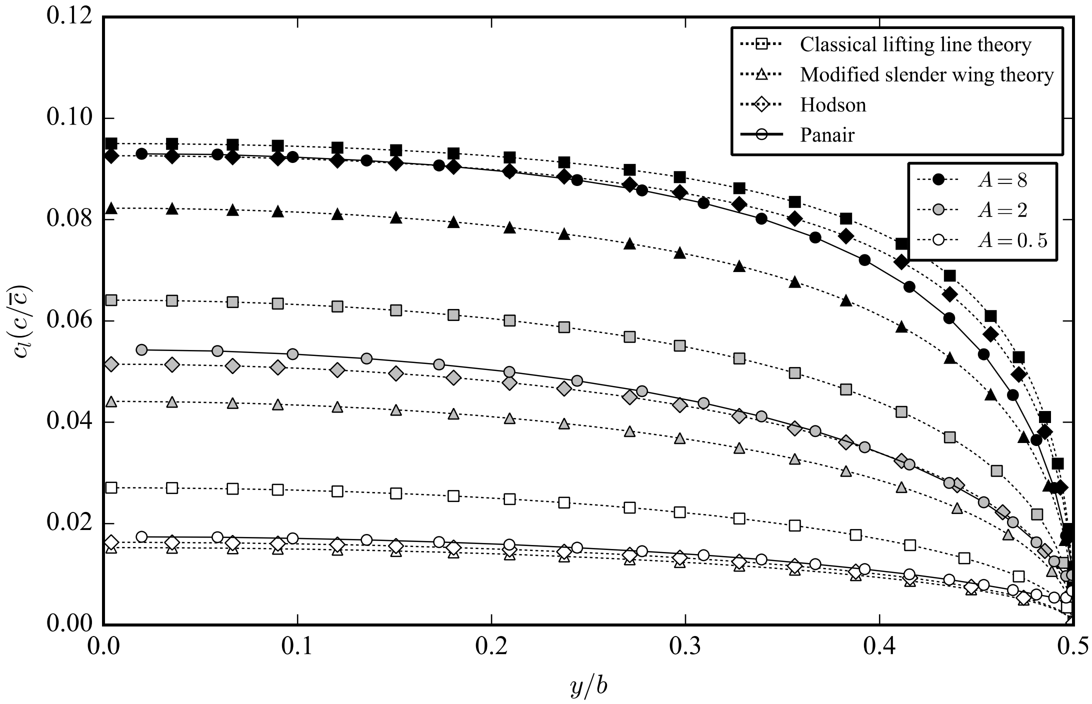


Figure 5.6 Comparison of spanwise lift distributions for rectangular wings.

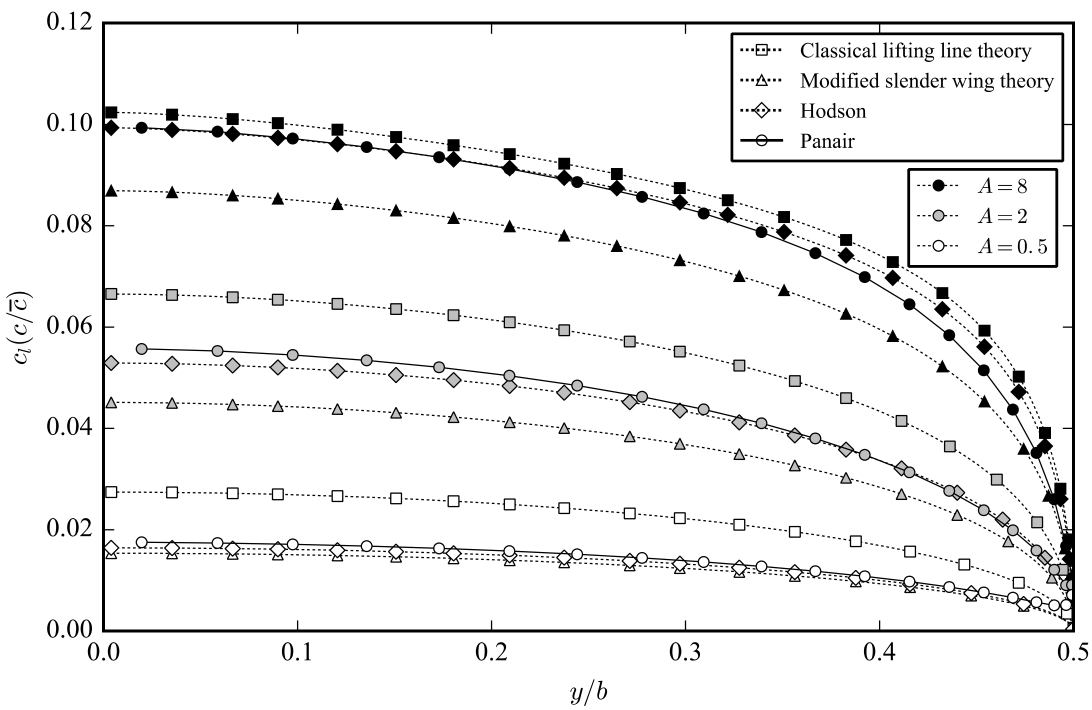


Figure 5.7 Comparison of spanwise lift distributions for tapered wings with $R_T = 0.75$.

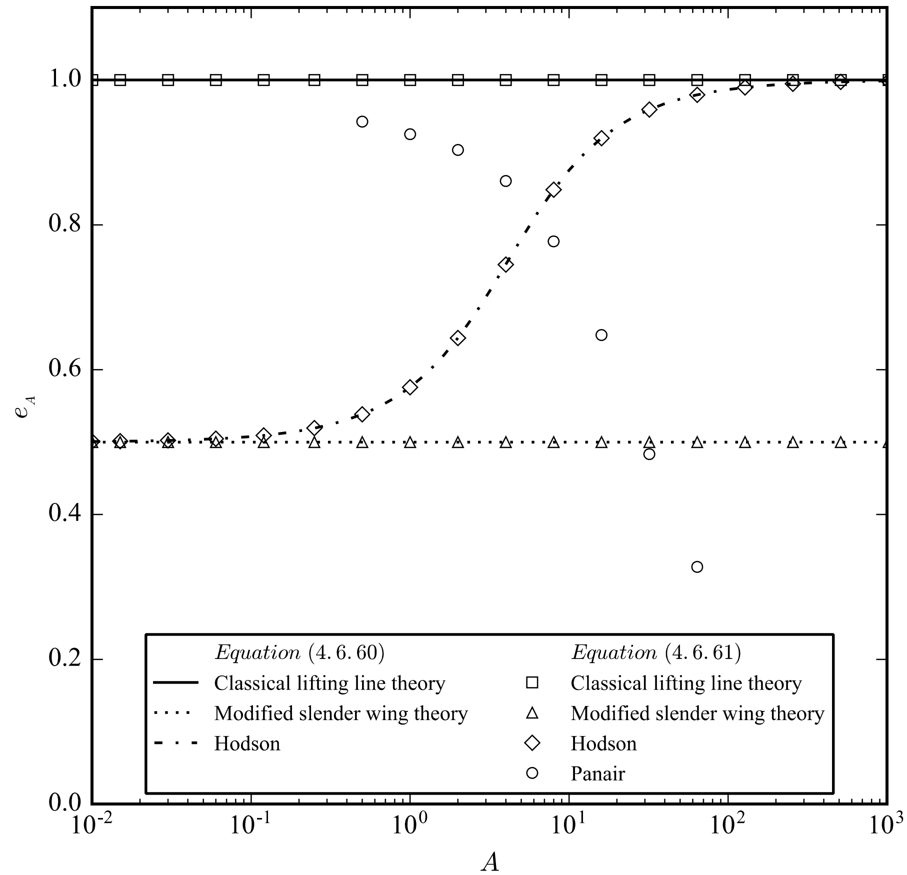


Figure 5.8 Sample calculations of the aspect ratio efficiency factor using MachUp and Panair.

In 2017, Hodson et al. [102] presented wind tunnel data for several subscale test articles of the variable-camber compliant wing (VCCW) developed at the U.S. Air Force Research Laboratory (AFRL). For a description of the VCCW, see Refs. [5,6]. Of particular interest here is the aspect ratio of the VCCW – $A = 3$ – which falls below the generally accepted range of validity for classical lifting line theory. The data presented by Hodson et al. [102] includes lift, drag, and pitching moment coefficients as functions of angle of attack for five subscale test articles generated from 3D scans of the VCCW in various morphed configurations, as well as three test articles generated from 3D CAD models with uniform NACA 0010, 2410, and 8410 airfoil cross-sections. For a description of each test article, see Ref. [102].

In order to compare numerical results generated with MachUp to the Hodson et al. [102] experimental data, an airfoil database needed to be created containing airfoil coefficients for the different cross sections in the models. For this analysis, the cross sections were approximated as NACA X410 series airfoils. XFOIL (see Refs. [51,52]) was used to generate viscous airfoil coefficients assuming a Reynolds number based on chord length of $Re_c = 2.4\times10^5$. The XFOIL results were tabulated for angles of attack between -10 deg and +10 deg. Airfoil coefficient tables for airfoils having 0%, 2%, 4%, 6%, and 8% maximum camber were generated and are listed in Appendix M. Properties for wing sections having maximum camber values between the values listed were linearly interpolated from the two closest airfoils.

Comparisons of lift coefficients for the test articles based on the 3D CAD models are given in Figures 5.9-5.11. Uncertainty bands representing 95% confidence intervals are included on the experimental data. For the CAD-0 and CAD-2 models, the corrected numerical results agree exceptionally well with the experimental data and outperform the numerical results calculated using the classical lifting line formulation. For the CAD-8 model, the corrected numerical results lie slightly below the experimental data but are still within the same range of accuracy as the classical lifting line results. Moreover, they provide a closer approximation to the lift slope for this data than the classical lifting line results provide.

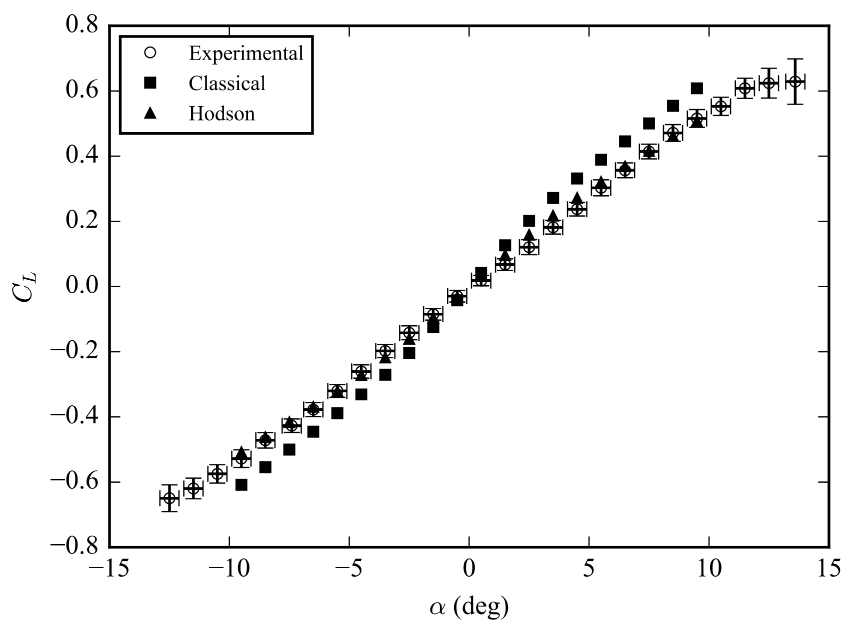


Figure 5.9 Comparison of numerical and experimental lift coefficients for the CAD-0 test.

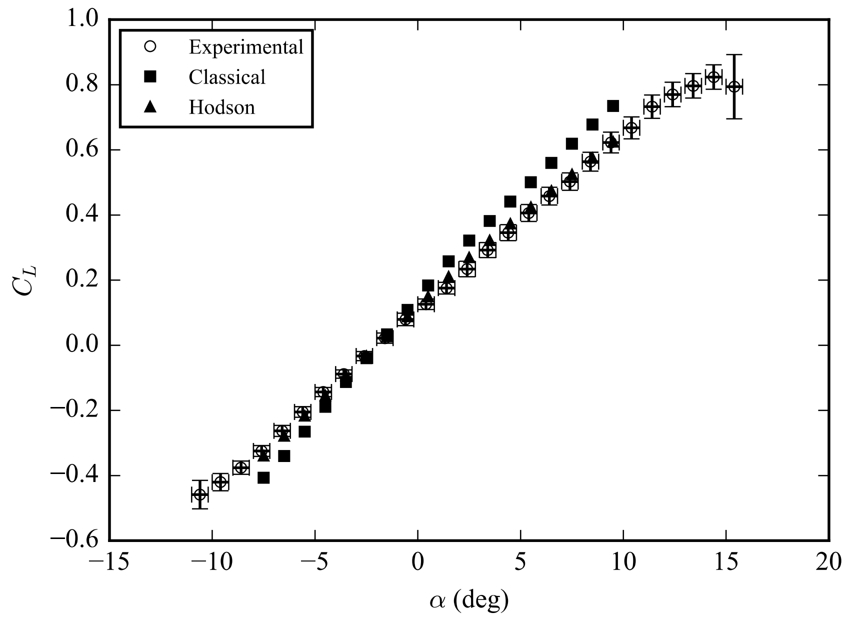


Figure 5.10 Comparison of numerical and experimental lift coefficients for the CAD-2 test.

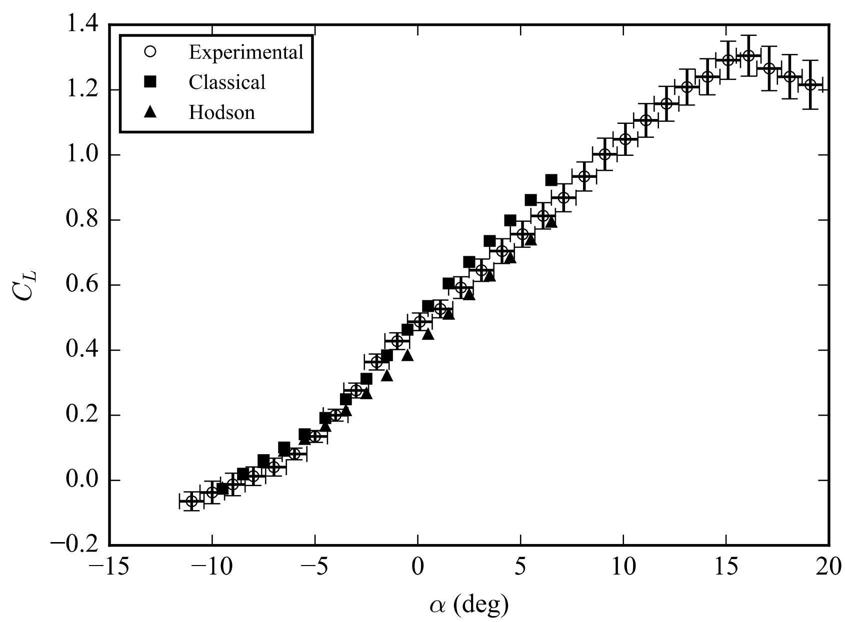


Figure 5.11 Comparison of numerical and experimental lift coefficients for the CAD-8 test.

Comparisons of lift coefficients for the subscale test articles generated from 3D scans of the VCCW are given in Figures 5.12-5.16. As with the CAD-0 and CAD-2 test articles, the corrected lifting line results for the VCCW-2 and VCCW-CD test articles are in excellent agreement with the experimental data. The corrected lifting line results for the VCCW-8 and VCCW-LW test articles lie slightly below the experimental data but have the correct lift slopes. The corrected numerical results for the VCCW-CU test article are the least satisfactory. They lie further below the experimental results than the classical lifting line results, and though the lift slope is slightly improved it is still too high. The reasons for the discrepancy in the VCCW-CU results are unknown, but there are several possibilities. The actual cross-sectional profiles of the VCCW-CU configuration may differ significantly from the NACA X410 profiles assumed in the numerical analysis. The XFOIL results for the airfoils used in the numerical analyses of the VCCW-CU may be in error, though this would likely also affect the comparisons of other test articles rather than be isolated to a single test article. There may also be an error in the experimental data that was not identified during the uncertainty analysis, though this too would likely also affect the comparisons of other test articles unless the error source was related directly to the setup and installation of the VCCW-CU test article in the wind tunnel. There also remains the possibility that the corrections to lifting line theory implemented here do not accurately account for the effects of thickness, camber, and viscosity, as none of these factors were considered in the theoretical development of the modifications. The VCCW-CU configuration may be such that these discrepancies are significantly amplified. Further investigation of this issue is needed.

Comparisons of drag coefficients for the test articles based on the 3D CAD models are given in Figures 5.17-5.19. In these plots, the numerical results using the corrected formulation and classical lifting line theory are quite similar because the only correction to drag was to the induced drag component, while most of the drag in these analyses comes from parasitic drag. In all three cases and for both numerical models the shape of the drag curve is slightly too concave. The largest source of error here is likely the parasitic drag coefficients determined from the XFOIL results.

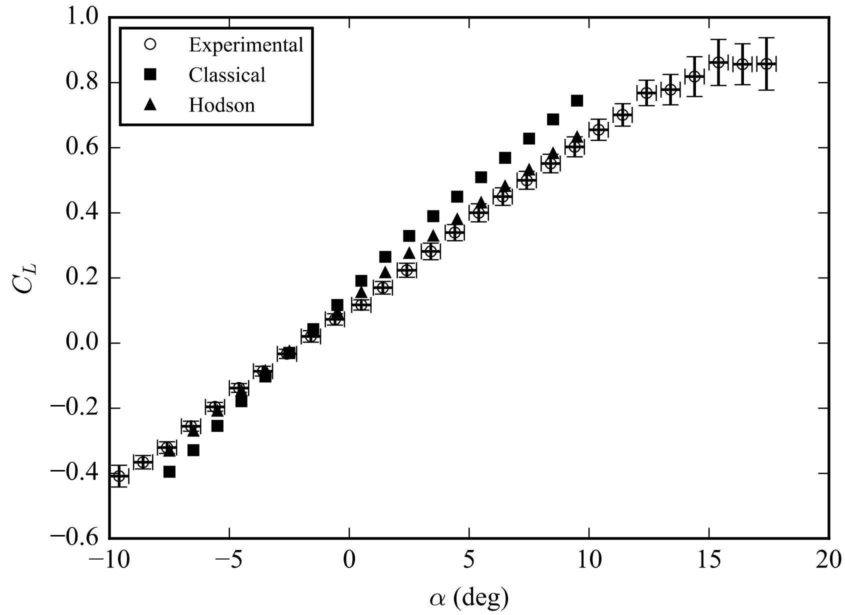


Figure 5.12 Comparison of numerical and experimental lift coefficients for the VCCW-2 test.

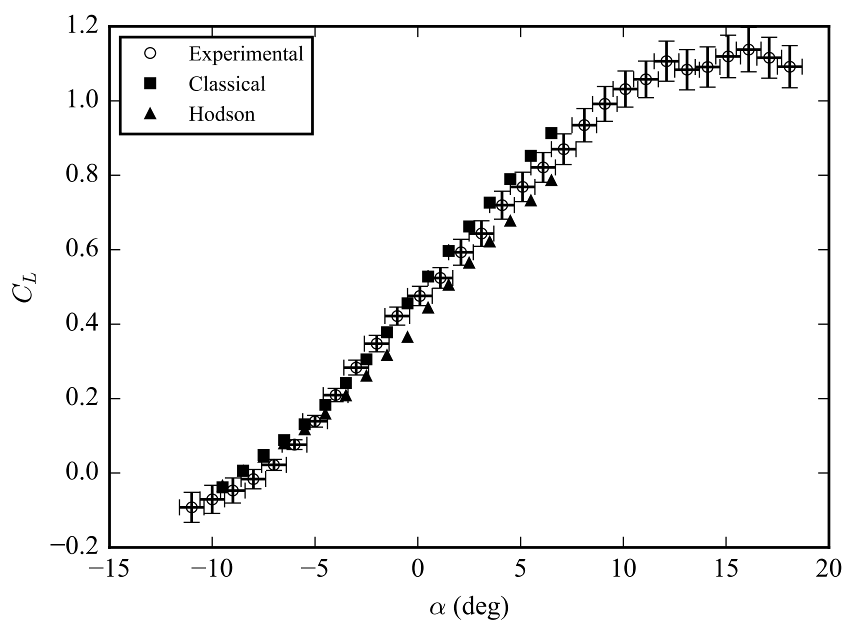


Figure 5.13 Comparison of numerical and experimental lift coefficients for the VCCW-8 test.

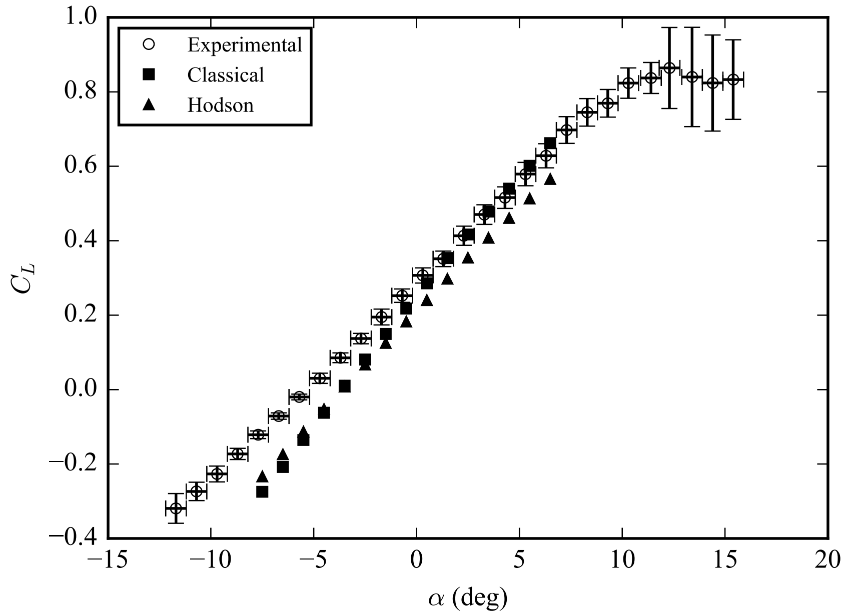


Figure 5.14 Comparison of numerical and experimental lift coefficients for the VCCW-CU test.

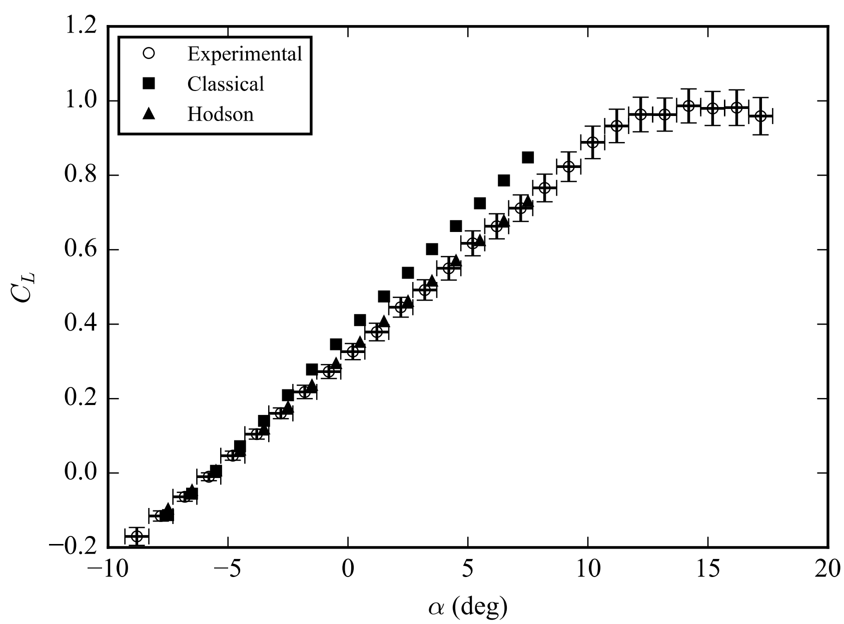


Figure 5.15 Comparison of numerical and experimental lift coefficients for the VCCW-CD test.

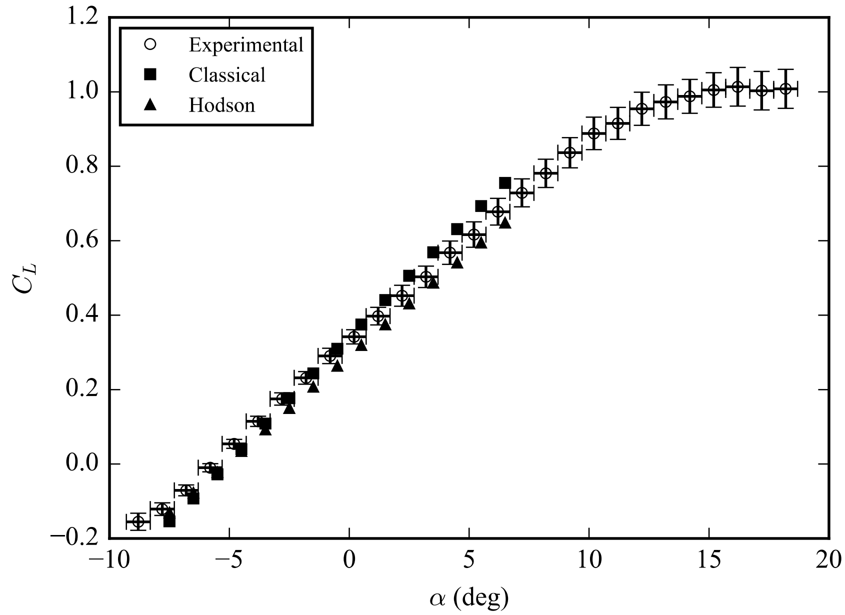


Figure 5.16 Comparison of numerical and experimental lift coefficients for the VCCW-LW test.

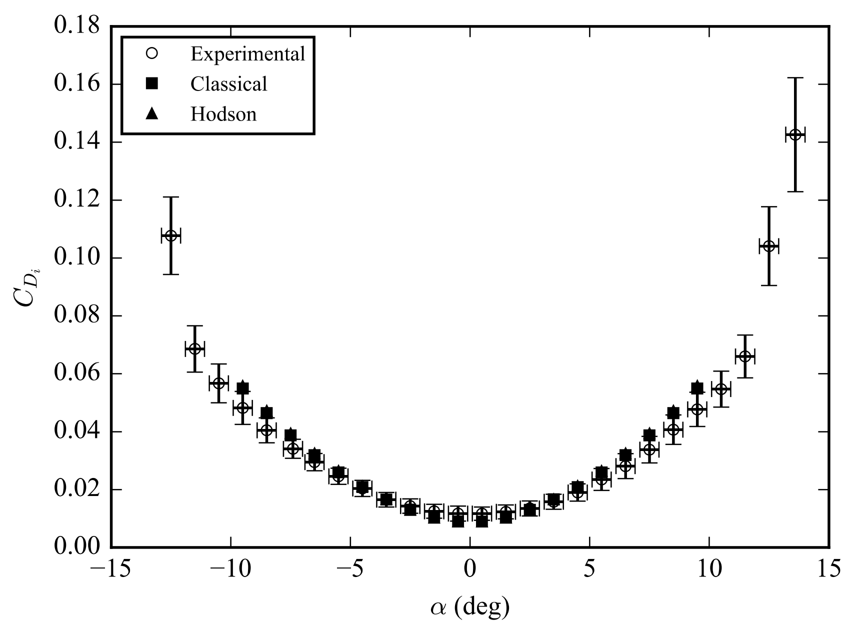


Figure 5.17 Comparison of numerical and experimental drag coefficients for the CAD-0 test.

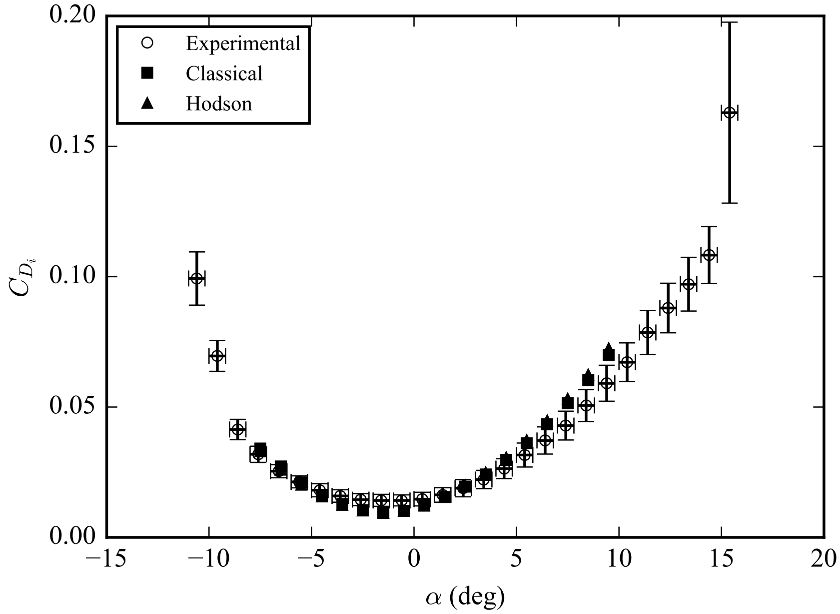


Figure 5.18 Comparison of numerical and experimental drag coefficients for the CAD-2 test.

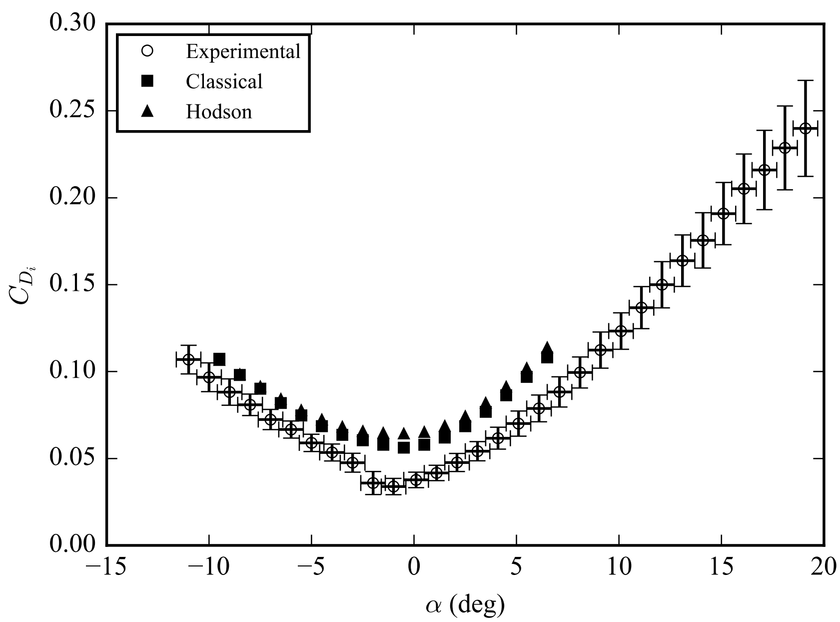


Figure 5.19 Comparison of numerical and experimental drag coefficients for the CAD-8 test.

Comparisons of lift coefficients for the subscale test articles generated from 3D scans of the VCCW are given in Figures 5.20-5.24. Again the differences between the two numerical formulations are mostly quite small, though the corrected model predicts a measurable increase in drag in the case of the VCCW-8 over the classical model. This is because $\partial C_{D_p} / \partial C_L$ is quite steep in the region at which this wing operates for the NACA 8410 airfoil, so that a relatively small change in lift results in a much larger change in drag. From these plots, only the results for the VCCW-CU model do a reasonable job of matching the experimental data. Results for the other models show too narrow of a drag curve, but this observation is true for both numerical formulations. The most likely source of error here is again the parasitic drag coefficients determined from the XFOIL results. This error is compounded by the fact that the cross-sectional profiles of the scanned models do not match the NACA X410 family of airfoils exactly. Large discrepancies between the cross-sectional profiles and the NACA X410 family of airfoils were noted especially in sections with high camber. Improved estimation of the parasitic drag properties of the airfoils used in these analyses is expected to resolve most of the error seen in these drag calculations.

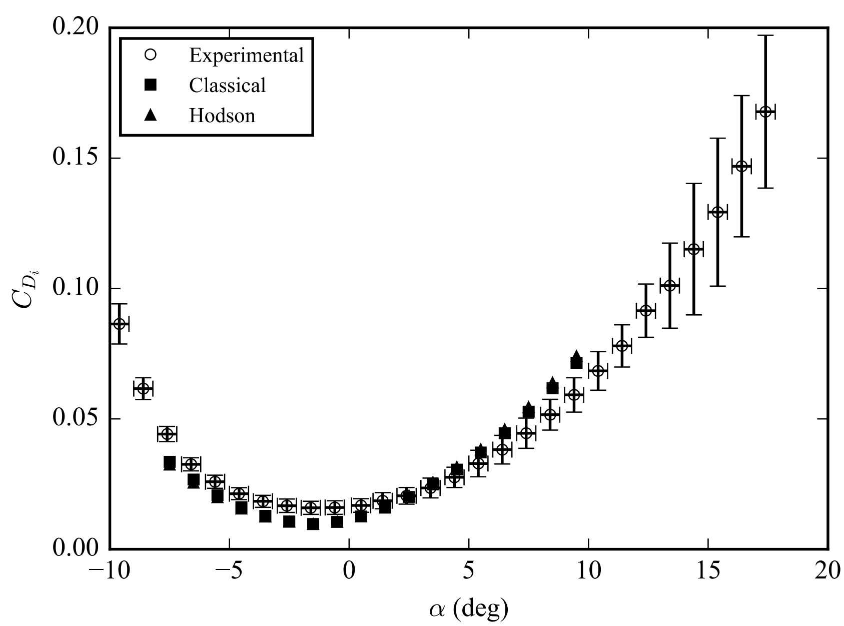


Figure 5.20 Comparison of numerical and experimental drag coefficients for the VCCW-2 test.

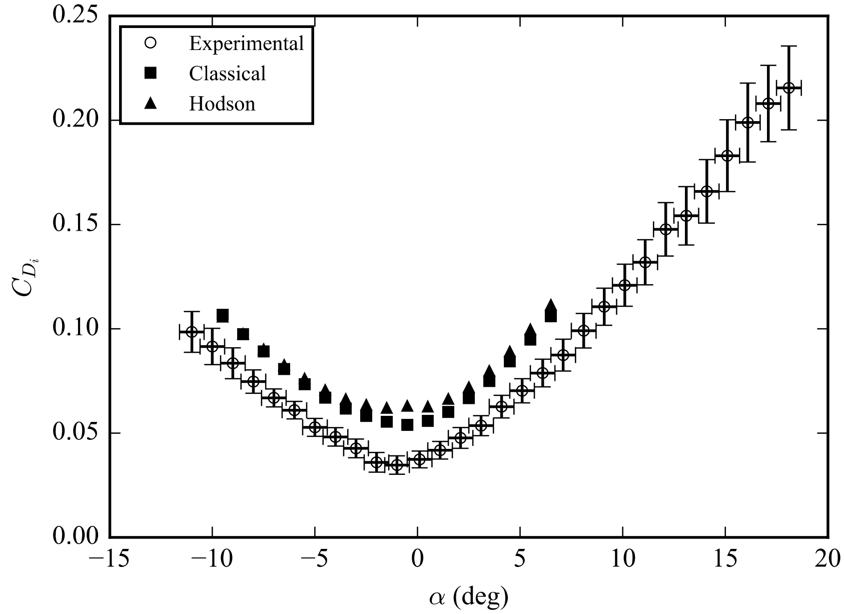


Figure 5.21 Comparison of numerical and experimental drag coefficients for the VCCW-8 test.

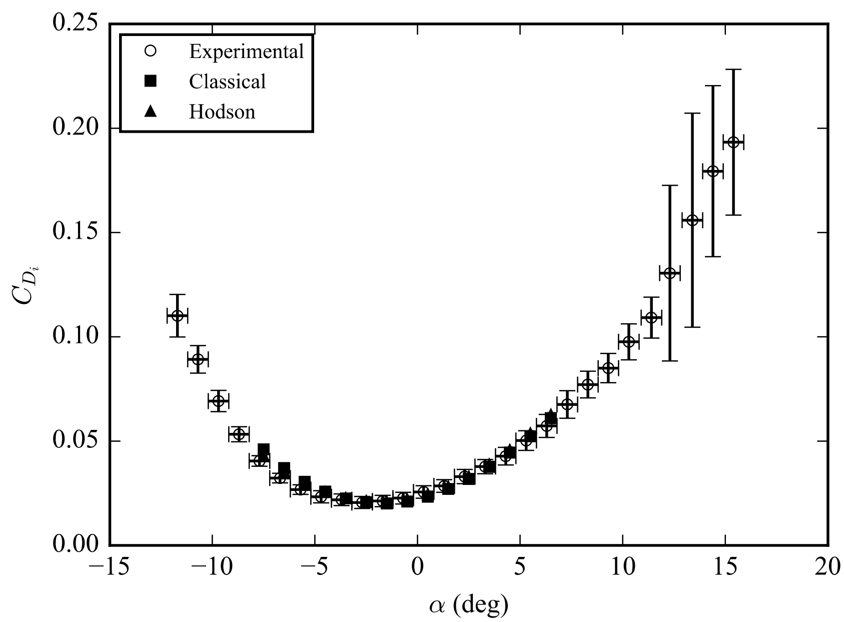


Figure 5.22 Comparison of numerical and experimental drag coefficients for the VCCW-CU test.

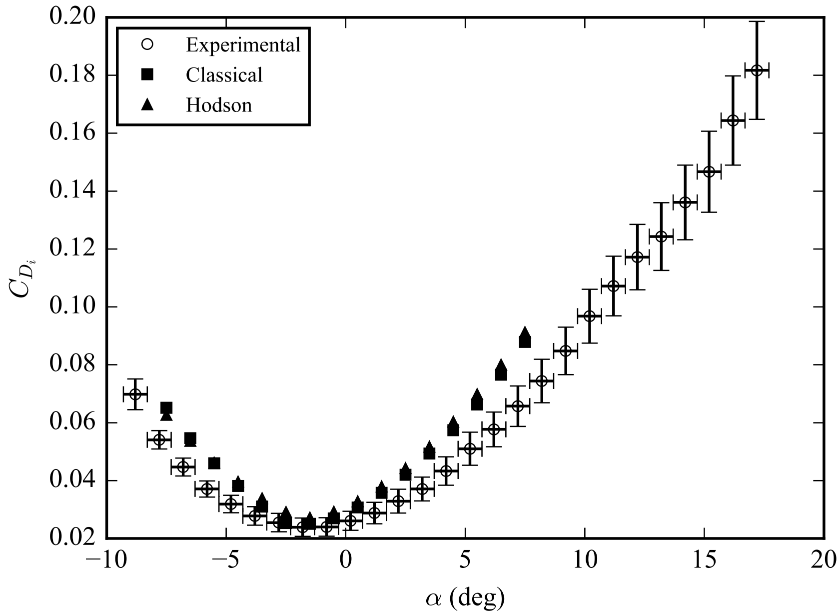


Figure 5.23 Comparison of numerical and experimental drag coefficients for the VCCW-CD test.

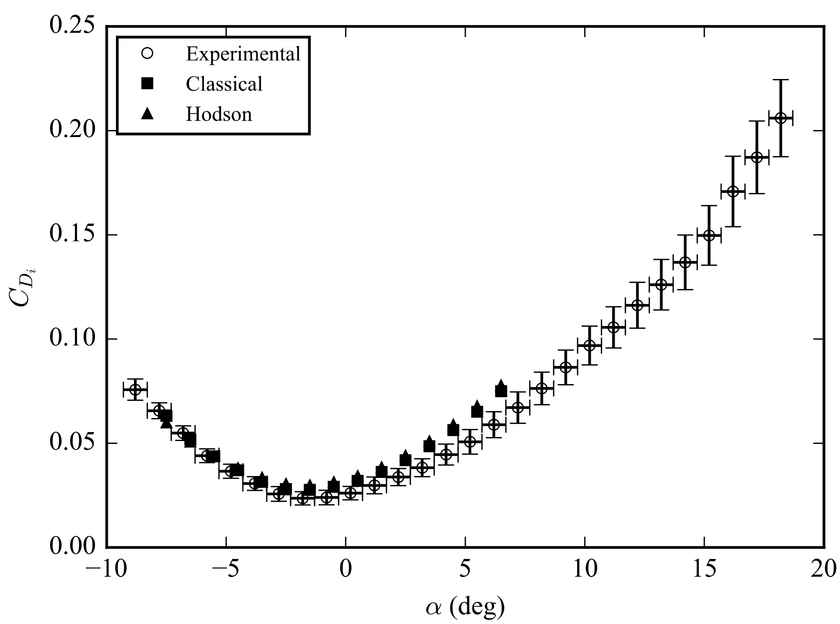


Figure 5.24 Comparison of numerical and experimental drag coefficients for the VCCW-LW test.

6 SUMMARY AND CONCLUSIONS

In this work, several limitations on analysis relative to wing shape optimization have been addressed. First, an efficient method for computing gradients with respect to design parameters in an aerodynamic analysis has been presented and integrated into MachUp, an open-source aerodynamic analysis tool based on a numerical lifting line method. The gradients are computed automatically using DNAD, a forward-mode automatic differentiation package that allows derivatives with respect to multiple input parameters to be computed in a single run. The DNAD module was integrated into the MachUp source code in such a way that inclusion or exclusion of the DNAD module can be controlled entirely at compile-time. In this way, the integration of the DNAD module does not affect the performance of MachUp when derivatives are not needed, and source code modifications are not required to convert between gradient-capable and non-gradient-capable versions of the code. To the author's knowledge, this is the first time this type of integration has been achieved with a practical engineering analysis tool.

Second, a process and suite of tools for performing efficient wing shape optimization has been presented and demonstrated. Optix, an open-source optimization framework that allows objective function evaluations to be run in parallel across available resources, was described in Sec. 3.2 and the source code is listed in Appendix F. Optix is written entirely in Python and is therefore cross-platform, easy to interface with, and easily customizable to a wide range of optimization problems. Its most significant features include the ability to execute independent function evaluations in parallel and to perform quadratic line-searching, both of which have the potential to significantly reduce the time required for complex optimization problems. It can also leverage automatic differentiation capabilities within objective functions to expedite efficient gradient calculations. Currently, constraints must be enforced through use of the penalty function method, and global minimization of non-convex design spaces cannot be guaranteed without the use of an outer wrapper implementing a globally convergent optimization method. These two limitations represent the primary areas of focus for future work in improving and expanding the capabilities of Optix.

Optix and MachUp were used together to solve several inviscid wing shape optimization problems with known solutions, and the solutions were shown to converge toward the correct solution as the number of degrees of freedom was increased. Solutions to several viscous wing shape optimization problems were also solved using this method.

The most significant source of error in the viscous results lies in the aerodynamic properties of the airfoils and the interpolation scheme used for determining aerodynamic properties of intermediate airfoils. Note that this is a problem with the data input to the analysis, however, and not a problem with the analysis tools themselves or the underlying methodology. Improving the accuracy of the airfoil aerodynamic properties and improving the interpolation scheme used for intermediate airfoils should provide the greatest improvement to the accuracy of these results.

Perhaps the most useful product of the optimization work presented in Chapter 3 is the contour plot shown in Figure 3.7. Using high-fidelity analysis tools such as CFD for wing shape optimization is not only computationally prohibitive but also precludes the ability to visualize the design and configuration space of a morphing wing. In this work, we have demonstrated a method for rapidly visualizing the design and configuration space of a finite morphing wing using MachUp. Figure 3.7 illustrates important relationships between lift, induced drag, and parasitic drag and demonstrates how changes in a finite wing design can impact these performance characteristics. The ability demonstrated here to quickly visualize and explore the design and configuration space of a morphing wing is a significant enabler in the push to develop advanced morphing wing technologies.

Although Chapter 3 only considered optimization of planform, geometric twist, and aerodynamic twist on a straight wing, the tools presented can be used to evaluate and optimize geometries of multiple interacting lifting surfaces as well as wings with more complex geometries. Examples of interacting lifting surfaces include the interaction of main wings and stabilizers, wings in formation, wings with winglets, and wings in ground effect. Due to modern composite manufacturing methods, very complex wing shapes can be designed and manufactured. The tools presented in this work can be used to optimize these complex wing designs in both single- and multi-wing systems.

Finally, development of new analytical and numerical formulations of lifting line theory have been presented. These formulations are based heavily on the classical lifting line theory of Prandtl [10,11] and the numerical lifting line method of Phillips and Snyder [16], but also draw on the works of several others (see Refs. [73,79,83,85,88–91,93]) to form a lifting line model accurate over the entire range of aspect ratios, from the slender wing to the infinite wing. The new formulations have been implemented in code and demonstrated to outperform classical lifting line theory in matching the lift results of an inviscid panel code for elliptic, rectangular, and tapered wings of arbitrary aspect ratio. They were also shown to outperform classical lifting line theory in matching the lift results of a viscous experimental investigation of the VCCW, a low-aspect-ratio rectangular wing with morphable cross-sections.

Drag predictions using the modified analytical and numerical formulations have been shown to agree with the theory presented, but results from Panair were contrary to the theory and no way to reconcile the discrepancies has yet been found. Additionally, drag predictions using the corrected numerical formulation compared to the VCCW experimental data set showed no improvement over classical lifting line theory predictions. Further research on the mechanisms that influence drag for low-aspect-ratio wings is needed to reconcile these data.

REFERENCES

[1] Baysal, O., and Eleshaky, M., “Aerodynamic Design Optimization Using Sensitivity Analysis and Computational Fluid Dynamics,” *AIAA Journal*, vol. 30, no. 3, Mar. 1992, pp. 718–725. doi:10.2514/3.10977.

[2] Samareh, J., “Aerodynamic Shape Optimization Based on Free-Form Deformation,” Albany, NY: AIAA, 2004.
doi:10.2514/6.2004-4630.

[3] Reuther, J., Jameson, A., Alonso, J., Rimlinger, M., and Saunders, D., “Constrained Multipoint Aerodynamic Shape Optimization Using an Adjoint Formulation and Parallel Computers, Part 2,” *Journal of Aircraft*, vol. 36, no. 1, 1999, pp. 61–74.
doi:10.2514/2.2414.

[4] Lyu, Z., Kenway, G., and Martins, J., “Aerodynamic Shape Optimization Investigations of the Common Research Model Wing Benchmark,” *AIAA Journal*, vol. 53, no. 4, Apr. 2015, pp. 968–985. doi:10.2514/1.J053318.

[5] Joo, J., Marks, C., Zientarski, L., and Culler, A., “Variable Camber Compliant Wing – Design,” *23rd AIAA/AHS Adaptive Structures Conference*, Kissimmee, FL: AIAA, 2015.
doi:10.2514/6.2015-1050.

[6] Marks, C., Zientarski, L., Culler, A., Hagen, B., Smyers, B., and Joo, J., “Variable Camber Compliant Wing – Wind Tunnel Testing,” *23rd AIAA/AHS Adaptive Structures Conference*, Kissimmee, FL: AIAA, 2015.
doi:10.2514/6.2015-1051.

[7] Su, W., Swei, S. S.-M., and Zhu, G., “Optimum Wing Shape of Highly Flexible Morphing Aircraft for Improved Flight Performance,” *Journal of Aircraft*, vol. 53, no. 5, Sep. 2016, pp. 1305–13016. doi:10.2514/1.C033490.

[8] Barbarino, S., Bilgen, O., and Ajaj, R., “A Review of Morphing Aircraft,” *Journal of Intelligent Material Systems and Structures*, vol. 22, Jun. 2011, pp. 823–877. doi:10.1177/1045389X11414084.

[9] Lanchester, F., *Aerodynamics: Constituting the First Volume of a Complete Work on Aerial Flight*, London: A. Constable, 1907.

[10] Prandtl, L., "Tragflügel Theorie," *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen*, 1918, pp. 451–477.

[11] Prandtl, L., *Applications of Modern Hydrodynamics to Aeronautics*, NACA, 1921.

[12] Phillips, W., "Lifting-Line Analysis for Twisted Wings and Washout-Optimized Wings," *Journal of Aircraft*, vol. 41, no. 1, Jan. 2004, pp. 128–136.
doi:10.2514/1.262.

[13] Phillips, W., Alley, N., and Goodrich, W., "Lifting-Line Analysis of Roll Control and Variable Twist," *Journal of Aircraft*, vol. 41, no. 5, Sep. 2004, pp. 1169–1176.
doi:10.2514/1.3846.

[14] Phillips, W. F., and Alley, N. R., "Predicting Maximum Lift Coefficient for Twisted Wings Using Lifting-Line Theory," *Journal of Aircraft*, vol. 44, no. 3, May 2007, pp. 898–910.
doi:10.2514/1.25640.

[15] Katz, J., and Plotkin, A., *Low-Speed Aerodynamics: From Wing Theory to Panel Methods*, McGraw-Hill, 1991.

[16] Phillips, W., and Snyder, D., "Modern Adaptation of Prandtl's Classic Lifting-Line Theory," *Journal of Aircraft*, vol. 37, no. 4, Jul. 2000, pp. 662–670.
doi:10.2514/2.2649.

[17] McCormick, B., "The Lifting Line Model," *Aerodynamics, Aeronautics, and Flight Mechanics*, New York: Wiley, 1994, p. 672.

[18] Anderson, J., Corda, S., and Van Wie, D., "Numerical Lifting-Line Theory Applied to Drooped Leading-Edge Wings Below and Above Stall," *Journal of Aircraft*, vol. 17, no. 12, Dec. 1980, pp. 898–904.
doi:10.2514/3.44690.

[19] Martins, J., "A Coupled-Adjoint Method for High-Fidelity Aero-Structural Optimization," Doctoral dissertation, Stanford, 2002.
http://aero-comlab.stanford.edu/Papers/martins.thesis.pdf.

[20] Simmons, G., *Calculus Gems: Brief Lives and Memorable Mathematics*, Washington, D.C.: The Mathematical Association of America, 2007.

[21] Courant, R., Friedrichs, K., and Lewy, H., "On the Partial Difference Equations of Mathematical Physics," *IBM Journal of Research and Development*, vol. 11, Mar. 1967, pp. 215–234. doi:10.1147/rd.112.0215.

[22] Hrennikoff, A., "Solution of Problems of Elasticity by the Framework Method," *Journal of Applied Mechanics*, vol. 8, no. 4, 1941, pp. A169–A175.

[23] Courant, R., "Variational Methods for the Solution of Problems of Equilibrium and Vibrations," *Bulletin of the American Mathematical Society*, vol. 49, pp. 1–23. doi:10.1090/s0002-9904-1943-07818-4.

[24] Thomée, V., "From Finite Differences to Finite Elements: A Short History of Numerical Analysis of Partial Differential Equations," *Journal of Computational and Applied Mathematics*, vol. 128, no. 1–2, Mar. 2001, pp. 1–54. doi:10.1016/S0377-0427(00)00507-0.

[25] Wengert, R., "A Simple Automatic Derivative Evaluation Program," *Communications of the ACM*, vol. 7, no. 8, Aug. 1964, pp. 463–464. doi:10.1145/355586.364791.

[26] Griewank, A., and Walther, A., *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Philadelphia, PA: Society of Industrial and Applied Mathematics, 2008. doi:10.1137/1.9780898717761.

[27] Rall, L., *Automatic Differentiation: Techniques and Applications*, New York, NY: Springer-Verlag, 1981.

[28] Corliss, G., Faure, C., Griewank, A., Hascoët, L., and Naumann, U., *Automatic Differentiation of Algorithms: From Simulation to Optimization*, New York, NY: Springer-Verlag New York, Inc., 2002.

[29] Martins, J., and Hwang, J., "Review and Unification of Methods for Computing Derivatives of Multidisciplinary Computational Models," *AIAA Journal*, vol. 51, no. 11, Sep. 2013, pp. 2582–2599. doi:10.2514/1.J052184.

[30] Šrajer, F., Kukelova, Z., and Fitzgibbon, A., "A Benchmark of Selected Algorithmic Differentiation Tools on Some Problems in Computer Vision and Machine Learning," *Optimization Methods and Software*, vol. 33, no. 4–6, Feb. 2018, pp. 889–906. doi:10.1080/10556788.2018.1435651.

[31] Yu, W., and Blair, M., "DNAD, A Simple Tool for Automatic Differentiation of Fortran Codes Using Dual Numbers," *Computer Physics Communications*, vol. 184, no. 5, May 2013, pp. 1446–1452. doi:10.1016/j.cpc.2012.12.025.

[32] Spall, R., and Yu, W., "Imbedded Dual-Number Automatic Differentiation for Computational Fluid Dynamics Sensitivity Analysis," *Journal of Fluids Engineering*, vol. 135, no. 1, Dec. 2012. doi:10.1115/1.4023074.

[33] Pakalapati, S., Sezer, H., and Celik, I., "Implications of using Dual Number Derivatives with a Numerical Solution," *Proceedings of the ASME 2013 FLuids Engineering Division Summer Meeting*, Jul. 2013.

[34] LeVeque, R., *Finite Difference Methods for Ordinary and Partial Differential Equations*, Philadelphia, PA: Society of Industrial and Applied Mathematics, 2007.

[35] Hascoët, L., and Pascual, V., "The Tapenade Automatic Differentiation Tool: Principles, Model, and Specifications," *ACM Transactions on Mathematical Software*, vol. 39, no. 3, Apr. 2013, pp. 1–20. doi:10.1145/2450153.2450158.

[36] Naumann, U., Utke, J., Heimbach, P., Hill, C., Ozyurt, D., Wunsch, C., Fagan, M., Tallent, N., and Strout, M., "Adjoint Code by Source Transformation with OpenAD/F," *Proceedings of the European Conference on Computational Fluid Dynamics*, 2006.

[37] Abdel-Khalik, H., Hovland, P., Lyons, A., Stover, T., and Utke, J., "A Low Rank Approach to Automatic Differentiation," *Advances in Automatic Differentiation*, vol. 64, 2008, pp. 55–65. doi:10.1007/978-3-540-68942-3_6.

[38] Qiang, C., Linbo, Z., and Bin, W., "Model Adjointization and its Cost," *Science in China*, vol. 47, no. 5, Oct. 2004, pp. 587–611. doi:10.1360/02yf0440.

[39] Bischof, C., Carle, A., Khademi, P., and Mauer, A., “ADIFOR 2.0: Automatic Differentiation of Fortran 77 Programs,” *IEEE Computational Science and Engineering*, vol. 3, no. 3, 1996, pp. 18–32. doi:10.1109/99.537089.

[40] Lyness, J., “Numerical Algorithms Based on the Theory of Complex Variable,” *Proceedings of the 1967 22nd ACM National Conference*, 1967, pp. 125–133. doi:10.1145/800196.805983.

[41] Martins, J., Sturdza, P., and Alonso, J., “The Complex-Step Derivative Approximation,” *ACM Transactions on Mathematical Software*, vol. 29, no. 3, Sep. 2003, pp. 245–262. doi:10.1145/838250.838251.

[42] Straka, C., “ADF95: Tool for Automatic Differentiation of a Fortran Code Designed for Large Numbers of Independent Variables,” *Computer Physics Communications*, vol. 168, no. 2, Jun. 2005, pp. 123–139. doi:10.1016/j.cpc.2005.01.011.

[43] Stamatiadis, S., Prosmiti, R., and Farantos, S., “AUTO_DERIV: Tool for Automatic Differentiation of a Fortran Code,” *Computer Physics Communications*, vol. 127, no. 2–3, May 2000, pp. 3443–355. doi:10.1016/S0010-4655(99)00513-5.

[44] Berz, M., Makino, K., Shamseddine, K., Hoffstätter, G., and Weishi, W., “COSY INFINITY and Its Applications in Nonlinear Dynamics,” *Computational Differentiation: Techniques, Applications, and Tools*, M. Berz, C. Bischof, G. Corliss, and A. Griewank, eds., Philadelphia, PA: Society of Industrial and Applied Mathematics, 1996, pp. 363–367.

[45] Pryce, J., and Reid, J., *AD01, A Fortran 90 Code for Automatic Differentiation*, Oxfordshire, England: Rutherford Appleton Laboratory, 1998. ftp://ftp.numerical.rl.ac.uk/pub/reports/prRAL98057.pdf. Accessed January 21, 2019.

[46] Clifford, W., “Preliminary Sketch of Biquaternions,” *Proceedings of the London Mathematical Society*, vol. 4, no. 64, 1973, pp. 381–395. doi:10.1112/plms/s1-4.1.381.

[47] Fike, J., and Alonso, J., "The Development of Hyper-Dual Numbers for Exact Second-Derivative Calculations," *49th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, Jan. 2011.

[48] Thomas, L., *Elliptic Problems in Linear Differential Equations over a Network*, New York: Columbia University, 1949.

[49] Hunsaker, D., and Snyder, D., "A Lifting-Line Approach to Estimating Propeller/Wing Interactions," *24th Applied Aerodynamics Conference*, Jun. 2006.

[50] Phillips, W., and Hunsaker, D., "Lifting-Line Predictions for Induced Drag and Lift in Ground Effect," *Journal of Aircraft*, vol. 50, no. 4, Aug. 2013, pp. 1226–1233.
doi:10.2514/1.C032152.

[51] Drela, M., "XFOIL: An Analysis and Design System for Low Reynolds Number Airfoils," *Low Reynolds Number Aerodynamics*, T.J. Mueller, ed., Berlin, Heidelberg: Springer-Verlag, 1989, pp. 1–12.

[52] Drela, M., and Giles, M., "Viscous-Inviscid Analysis of Transonic and Low Reynolds Number Airfoils," *AIAA Journal*, vol. 25, no. 10, 1987, pp. 1347–1355.
doi:10.2514/3.9789.

[53] Phillips, W., *Mechanics of Flight*, Hoboken, NJ: John Wiley and Sons, Inc., 2010.

[54] Oliphant, T., *Guide to NumPy*, USA: Trelgol Publishing, 2006.

[55] Broyden, C., "The Convergence of a Class of Double-Rank Minimization Algorithms," *Journal of the Institute of Mathematics and Its Applications*, vol. 6, 1970, pp. 76–90.
doi:10.1093/imamat/6.1.76.

[56] Fletcher, R., "A New Approach to Variable Metric Algorithms," *Computer Journal*, vol. 13, no. 3, 1970, pp. 317–322.
doi:10.12691/ajams-4-5-1.

[57] Goldfarb, D., "A Family of Variable Metric Updates Derived by Variational Means," *Mathematics of Computation*, vol. 24, no. 109, 1970, pp. 23–26.
doi:10.2307/2004873.

[58] Shanno, D., "Conditioning of Quasi-Newton Methods for Function Minimization," *Mathematics of Computation*, vol. 24, no. 111, 1970, pp. 647–656. doi:10.2307/2004840.

[59] Nocedal, J., and Wright, S., *Numerical Optimization*, Berlin, New York: Springer-Verlag, 2006.

[60] Wolfe, P., "Convergence Conditions for Ascent Methods," *SIAM Review*, vol. 11, no. 2, 1969, pp. 226–235. doi:10.1137/1011036.

[61] Wolfe, P., "Convergence Conditions for Ascent Methods. II: Some Corrections," *SIAM Review*, vol. 13, no. 2, 1971, pp. 185–188. doi:10.1137/1013035.

[62] Hestenes, M., and Stiefel, E., "Methods of Conjugate Gradients for Solving Linear Systems," *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, Dec. 1952, pp. 409–436. doi:10.6028/jres.049.044.

[63] Goldstein, A., "Cauchy's Method of Minimization," *Numerische Mathematik*, vol. 4, no. 1, Dec. 1962, pp. 146–150. doi:10.1007/BF01386306.

[64] Armijo, L., "Minimization of Functions Having Lipschitz Continuous First Partial Derivatives," *Pacific Journal of Mathematics*, vol. 16, no. 1, 1966, pp. 1–3. doi:10.2140/pjm.1966.16.1.

[65] Smith, A., and Coit, D., "Constraint-Handling Techniques – Penalty Functions," *Handbook of Evolutionary Computation*, Bristol, United Kingdom: Institute of Physics Publishing and Oxford University Press, 1997, p. C5.2:1-6.

[66] Griva, I., Nash, S., and Sofer, A., *Linear and Nonlinear Optimization*, Society of Industrial and Applied Mathematics, 2009.

[67] Svanberg, K., "A Class of Globally Convergent Optimization Methods Based on Conservative Convex Separable Approximations," *SIAM Journal on Optimization*, vol. 12, no. 2, 2002, pp. 555–573. doi:10.1137/S1052623499362822.

[68] Locatelli, M., and Schoen, F., *Global Optimization*, Philadelphia, PA: Society of Industrial and Applied Mathematics, 2013.

[69] Phillips, W. F., "Lifting-Line Analysis for Twisted Wings and Washout-Optimized Wings," *Journal of Aircraft*, vol. 41, no. 1, Jan. 2004, pp. 128–136.
doi:10.2514/1.262.

[70] Birnbaum, W., "Die tragende Wirbelfläche als Hilfsmittel zur Behandlung des ebenen Problems der Tragflügeltheorie," *Zeitschrift für angewandte Mathematik und Mechanik*, vol. 3, no. 4, 1923, pp. 290–297.
doi:10.1002/zamm.19230030408.

[71] Blenk, H., "Der Eindecker als tragende Wirbelfläche," *Zeitschrift für angewandte Mathematik und Mechanik*, vol. 5, no. 1, 1925, pp. 36–47.
doi:10.1002/zamm.19250050104.

[72] Multhopp, H., *Methods for Calculating the Lift Distribution of Wings (Subsonic Lifting-Surface Theory)*, London: Aeronautical Research Council, 1955.

[73] Kuchemann, D., *A Simple Method for Calculating the Span and Chordwise Loading on Straight and Swept Wings of any Given Aspect Ratio at Subsonic Speeds.*, Aeronautical Research Council, London, 1952.
http://www.dtic.mil/docs/citations/ADA951920. Accessed November 6, 2018.

[74] Helmholtz, H., "Theorie der Luftschwingungen in Röhren mit offenen Enden," *Journal für die Reine und Angewandte Mathematik*, vol. 57, no. 1, 1860, pp. 1–72.

[75] Anderson, J. D., *Fundamentals of Aerodynamics*, McGraw-Hill, .

[76] Karamcheti, K., *Principles of Ideal-Fluid Aerodynamics*, New York: Wiley, 1966.

[77] Munk, M., *The Aerodynamic Forces on Airship Hulls*, NACA, 1924.

[78] Bollay, W., "A Non-linear Wing Theory and its Application to Rectangular Wings of Small Aspect Ratio," *Zeitschrift für angewandte Mathematik und Mechanik*, vol. 19, no. 1, Feb. 1939, pp. 21–35.
doi:10.1002/zamm.19390190103.

[79] Jones, R., *Properties of Low-Aspect-Ratio Pointed Wings at Speeds Below and Above the Speed of Sound*, Langley Field, VA: NACA Langley Aeronautical Lab., 1946.

[80] Zimmerman, C., *Characteristics of CLark Y Airfoils of Small Aspect Ratios*, Langley Field, VA: NACA Langley Aeronautical Lab., 1933.

[81] Zimmerman, C., *Aerodynamic Characteristics of Several Airfoils of Low Aspect Ratio*, Langley Field, VA: NACA Langley Aeronautical Lab., 1935.

[82] Winter, H., *Flow Phenomena on Plates and Airfoils of Short Span*, Washington, D.C.: NACA, 1936.

[83] Hauptman, A., and Miloh, T., "On the Exact Solution of the Linearized Lifting-Surface Problem of an Elliptic Wing," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 39, no. 1, Feb. 1986, pp. 41–66.
doi:10.1093/qjmam/39.4.580.

[84] Prandtl, L., *Recent Work on Airfoil Theory*, Washington, D.C.: NACA, 1940.

[85] Hauptman, A., and Miloh, T., "Aerodynamic Coefficients of a Thin Elliptic Wing in Unsteady Motion," *AIAA Journal*, vol. 25, no. 6, Jun. 1987, pp. 769–774.

[86] Smith, J., "Comment on 'Exact and Asymptotic Expressions of the Lift Slope Coefficient of an Elliptic Wing,'" *AIAA Journal*, vol. 28, no. 6, Jun. 1990, p. 1146.
doi:10.2514/3.48881.

[87] Laitone, E., "Lift-Curve Slope for Finite-Aspect-Ratio Wings," *Journal of Aircraft*, vol. 26, no. 8, Aug. 1989, pp. 789–790.
doi:10.2514/3.45841.

[88] Jones, R., *Correction of the Lifting-Line Theory for the Effect of the Chord*, Washington, D.C.: NACA, 1941.

[89] Helmbold, H., "Der Unverwundene Ellipsenflügel als Tragende Fläche," *Jahrbuch 1942 der Deutschen Luftfahrtforschung*, 1942, pp. 2–4.

[90] Van Dyke, M., "Some Inviscid Singular Perturbation Problems," *Perturbation Methods in Fluid Mechanics*, Stanford, CA: The Parabolic Press, 1975, pp. 167-193. Also see Note 13, pp. 239–241.

[91] Germain, P., "Recent Evolution in Problems and Methods in Aerodynamics," *Journal of the Royal Aeronautical Society*, vol. 71, no. 682, Oct. 1967, pp. 673–691.
doi:10.1017/S0001924000054324.

[92] Kida, T., and Miyai, Y., “An Alternative Treatment of Lifting-Line Theory as a Perturbation Problem,” *Journal of Applied Mathematics and Physics*, vol. 29, no. 4, Jul. 1978, pp. 591–607. doi:10.1007/BF01601487.

[93] Van Dyke, M., *Perturbation Methods in Fluid Mechanics*, Stanford, CA: Parabolic Press, 1975.

[94] Kinner, W., “Über Tragflügel mit kreisförmigem Grundriss,” *Zeitschrift für Angewandte Mathematik und Mechanik*, vol. 16, no. 6, Dec. 1936, pp. 349–352.

[95] Krienes, K., *The Elliptic Wing Based on the Potential Theory*, Washington, D.C.: NACA, 1941.

[96] Jordan, P., “On Lifting Wings with Parabolic Tips,” *Zeitschrift für Angewandte Mathematik und Mechanik*, vol. 54, no. 7, 1974, pp. 463–477.

[97] Medan, R., *Improvements to the Kernel Function Method of Steady, Subsonic Lifting Surface Theory*, Moffett Field, CA: NASA Ames Research Center, 1974.

[98] Derbyshire, T., and Sidwell, K. W., *PAN AIR Summary Document (version 1.0)*, Moffett Field, CA: NASA Ames Research Center, 1982. https://ntrs.nasa.gov/search.jsp?R=19840020672. Accessed August 24, 2018.

[99] Epton, M. A., and Magnus, A. E., *PAN AIR: A computer program for predicting subsonic or supersonic linear potential flows about arbitrary configurations using a higher order panel method. Volume 2: User's Manual (version 1.0)*, Moffett Field, CA: NASA Ames Research Center, 1981. https://ntrs.nasa.gov/search.jsp?R=19840019629. Accessed August 24, 2018.

[100] Epton, M. A., and Magnus, A. E., *PAN AIR: A computer program for predicting subsonic or supersonic linear potential flows about arbitrary configurations using a higher order panel method. Volume 1: Theory document (version 1.1)*, Moffett Field, CA: NASA Ames Research Center, 1981. https://ntrs.nasa.gov/search.jsp?R=19840019629. Accessed August 24, 2018.

[101] Saffman, P., *Vortex Dynamics*, Cambridge: Cambridge University Press, 1993.

[102] Hodson, J., Hunsaker, D., Andrews, B., and Joo, J., “Experimental Results for a Variable Camber Compliant Wing,” Denver, CO: AIAA, 2017, pp. 1–21. doi:10.2514/6.2017-4222.

APPENDICES

A MODIFIED DNAD SOURCE CODE

```
!*****************************************************************************
!* Dual Number Automatic Differentiation (DNAD) of Fortran Codes
!*----------------------------------------------------------------------------
!* COPYRIGHT (c) Joshua Hodson, All rights reserved, you are free to copy,
!* modify, or translate this code to other languages such as c/c++. This is a
!* fork of the original Fortran DNAD module developed by Dr. Wenbin Yu. See
!* original copyright information below. You can download the original version
!* at https://cdmhub.org/resources/374
!*
!* COPYRIGHT (c) Wenbin Yu, All rights reserved, you are free to copy,
!* modify or translate this code to other languages such as c/c++. If
!* you find a bug please let me know through wenbinyu.heaven@gmail.com. If
!* you added new functions and want to share with others, please let me know
!* too. You are welcome to share your successful stories with us through
!* http://groups.google.com/group/hifi-comp.
!*****************************************************************************
!* Acknowledgements
!*----------------------------------------------------------------------------
!* The development of DNAD is supported, in part, by the Chief Scientist
!* Innovative Research Fund at AFRL/RB WPAFB, and by Department of Army
!* SBIR (Topic A08-022) through Advanced Dynamics Inc. The views and
!* conclusions contained herein are those of the authors and should not be
!* interpreted as necessarily representing the official policies or
!* endorsement, either expressed or implied, of the funding agency.
!*
!* Additional development of DNAD has been supported under a Department of
!* Energy (DOE) Nuclear Energy University Program (NEUP) Graduate Fellowship.
!* Any opinions, findings, conclusions or recommendations expressed in this
!* publication are those of the authors and do not necessarily reflect the
!* views of the Department of Energy Office of Nuclear Energy.
!*****************************************************************************
!* Citation
!*----------------------------------------------------------------------------
!* Your citation of the following two papers is appreciated:
!* Yu, W. and Blair, M.: "DNAD, a Simple Tool for Automatic Differentiation of
!* Fortran Codes Using Dual Numbers," Computer Physics Communications, vol.
!* 184, 2013, pp. 1446-1452.
!*
!* Spall, R. and Yu, W.: "Imbedded Dual-Number Automatic Differentiation for
!* CFD Sensitivity Analysis," Journal of Fluids Engineering, vol. 135, 2013,
!* 014501.
!*****************************************************************************
!* Quick Start Guide
!*----------------------------------------------------------------------------
!* To integrate DNAD into an existing Fortran program, do the following:
!*
!*   1. Include the DNAD module in the source files by adding "use dnadmod" to
!*      the beginning of all modules, global functions, and global subroutines
!*      that include definitions of floating-point variables.
!*   2. Redefine all floating-point variables as type(dual). This can be done
!*      using precompiler directives so that the integration can be turned on
!*      or off at compile-time, eliminating the need for maintaining two
!*      separate code bases for the same project.
!*   3. All I/O involving floating-point variables will need to be examined.
!*      A method will need to be determined for inputting and outputting
!*      derivative values. This customization is typically unique for each
!*      piece of software and needs to be determined on a case-by-case basis.
!*   4. When compiling DNAD, use the compiler option "-Dndv=#", where # is the
!*      number of design variables desired. This sizes the derivative array
```

```
!*      that is stored with each floating point number.
!*   5. When compiling DNAD, use compiler options to specify precision. If no
!*      compiler options are specified, DNAD will default to single-precision
!*      floating-point arithmetic. Most popular Fortran compilers provide
!*      options for specifying precision at compile-time so that it does not
!*      have to be hard-coded into the source code. For example, use the
!*      "-fdefault-real-8" compiler in gfortran or the "-r8" compiler option
!*      with Intel Fortran to compile DNAD as double-precision.
!*   6. Modify the compilation process for the target software to include the
!*      DNAD module in the resulting executable or library.
!******************************************************************************
!* Change Log
!*-----------------------------------------------------------------------------
!*
!*  2016-04-29  Joshua Hodson
!*  - Updated copyright, acknowledgments, and quick start guide.
!*  - Removed overloads for single-precision reals.
!*  - Added tan, dtan, atan, and atan2 intrinsic function overloads.
!*  - Removed macro for precision and defined all floating-point variables as
!*    default real. Compiler options can now be used to set precision.
!*  - Added checks for undefined derivatives when only constants are used in
!*    the calculation (i.e. all partial derivatives are zero). This limits the
!*    perpetuation of NaN values in the code.
!*  - Combined the header and source files into a single file.
!*
!*  2015-07-29  Joshua Hodson
!*  - Added maxloc intrinsic function overload.
!*  - Converted UPPERCASE to lowercase for readability.
!*  - Added macros for defining precision and number of design variables.
!*  - Renamed module from Dual_Num_Auto_Diff to dnadmod
!*  - Renamed dual number type from DUAL_NUM to dual
!*  - Renamed components of dual number type from (xp_ad_, xp_ad_) to (x, dx)
!*
!*  2014-06-05  Wenbin Yu
!*  - Forked from original DNAD repository, see https://cdmhub.org/resources/374
!*
!******************************************************************************

! Number of design variables (default = 1)
#ifndef ndv
#define ndv 1
#endif

module dnadmod

    implicit none

    private

    real :: negative_one = -1.0
    type,public:: dual  ! make this private will create difficulty to use the
                        ! original write/read commands, hence x and dx are
                        ! variables which can be accessed using D%x and D%dx in
                        ! other units using this module in which D is defined
                        ! as type(dual).
        sequence
        real :: x  ! functional value
        real :: dx(ndv)  ! derivative
    end type dual


```

```
!******** Interfaces for operator overloading
    public assignment (=)
    interface assignment (=)
        module procedure assign_di  ! dual=integer, elemental
        module procedure assign_dr  ! dual=real, elemental
        module procedure assign_id  ! integer=dual, elemental
    end interface


    public operator (+)
    interface operator (+)
        module procedure add_d   ! +dual number, elemental
        module procedure add_dd  ! dual + dual, elemental
        module procedure add_di  ! dual + integer, elemental
        module procedure add_dr  ! dual + real, elemental
        module procedure add_id  ! integer + dual, elemental
        module procedure add_rd  ! real + dual, elemental
    end interface

    public operator (-)
    interface operator (-)
        module procedure minus_d   ! negate a dual number,elemental
        module procedure minus_dd  ! dual -dual,elemental
        module procedure minus_di  ! dual-integer,elemental
        module procedure minus_dr  ! dual-real,elemental
        module procedure minus_id  ! integer-dual,elemental
        module procedure minus_rd  ! real-dual,elemental
    end interface

    public operator (*)
    interface operator (*)
        module procedure mult_dd    ! dual*dual, elemental
        module procedure mult_di    ! dual*integer,elemental
        module procedure mult_dr    ! dual*real,elemental
        module procedure mult_id    ! integer*dual,elemental
        module procedure mult_rd    ! real*dual,elemental
    end interface

    public operator (/)
    interface operator (/)
        module procedure div_dd ! dual/dual,elemental
        module procedure div_di ! dual/integer, elemental
        module procedure div_dr ! dual/real,emental
        module procedure div_id ! integer/dual, elemental
        module procedure div_rd ! real/dual, elemental
    end interface

    public operator (**)
    interface operator (**)
        module procedure pow_i ! dual number to an integer power,elemental
        module procedure pow_r ! dual number to a real power, elemental
        module procedure pow_d ! dual number to a dual power, elemental
    end interface

    public operator (==)
    interface operator (==)
        module procedure eq_dd ! compare two dual numbers, elemental
        module procedure eq_di ! compare a dual and an integer, elemental
        module procedure eq_dr ! compare a dual and a real, elemental
        module procedure eq_id ! compare integer with a dual number, elemental
        module procedure eq_rd ! compare a real with a dual number, elemental
```

```
    end interface

    public operator (<=)
    interface operator (<=)
        module procedure le_dd  ! compare two dual numbers, elemental
        module procedure le_di  ! compare a dual and an integer, elemental
        module procedure le_dr  ! compare a dual and a real,elemental
        module procedure le_id ! compare integer with a dual number, elemental
        module procedure le_rd ! compare a real with a dual number, elemental
    end interface

    public operator (<)
    interface operator (<)
        module procedure lt_dd  !compare two dual numbers, elemental
        module procedure lt_di  !compare a dual and an integer, elemental
        module procedure lt_dr  !compare dual with a real, elemental
        module procedure lt_id ! compare integer with a dual number, elemental
        module procedure lt_rd ! compare a real with a dual number, elemental
    end interface

    public operator (>=)
    interface operator (>=)
        module procedure ge_dd ! compare two dual numbers, elemental
        module procedure ge_di ! compare dual with integer, elemental
        module procedure ge_dr ! compare dual with a real number, elemental
        module procedure ge_id ! compare integer with a dual number, elemental
        module procedure ge_rd ! compare a real with a dual number, elemental
    end interface

    public operator (>)
    interface operator (>)
        module procedure gt_dd  !compare two dual numbers, elemental
        module procedure gt_di  !compare a dual and an integer, elemental
        module procedure gt_dr  !compare dual with a real, elemental
        module procedure gt_id ! compare integer with a dual number, elemental
        module procedure gt_rd ! compare a real with a dual number, elemental
    end interface

    public operator (/=)
    interface operator (/=)
        module procedure ne_dd  !compare two dual numbers, elemental
        module procedure ne_di  !compare a dual and an integer, elemental
        module procedure ne_dr  !compare dual with a real, elemental
        module procedure ne_id ! compare integer with a dual number, elemental
        module procedure ne_rd ! compare a real with a dual number, elemental
    end interface


!------------------------------------------------
! Interfaces for intrinsic functions overloading
!------------------------------------------------
    public abs
    interface abs
        module procedure abs_d  ! absolute value of a dual number, elemental
    end interface

    public dabs
    interface dabs
        module procedure abs_d ! same as abs, used for some old fortran commands
    end interface

```

```fortran
    public acos
    interface acos
        module procedure acos_d ! arccosine of a dual number, elemental
    end interface

    public asin
    interface asin
        module procedure asin_d ! arcsine of a dual number, elemental
    end interface

    public atan
    interface atan
        module procedure atan_d ! arctan of a dual number, elemental
    end interface

    public atan2
    interface atan2
        module procedure atan2_d ! arctan of a dual number, elemental
    end interface

    public cos
    interface cos
        module procedure cos_d ! cosine of a dual number, elemental
    end interface

    public dcos
    interface dcos
        module procedure cos_d ! cosine of a dual number, elemental
    end interface

    public dot_product
    interface dot_product
        module procedure dot_product_dd ! dot product two dual number vectors
    end interface

    public exp
    interface exp
        module procedure exp_d ! exponential of a dual number, elemental
    end interface

    public int
    interface int
        module procedure int_d ! integer part of a dual number, elemental
    end interface

    public log
    interface log
        module procedure log_d ! log of a dual number, elemental
    end interface

    public log10
    interface log10
        module procedure log10_d ! log of a dual number, elemental
    end interface

    public matmul
    interface matmul
        module procedure matmul_dd ! multiply two dual matrices
        module procedure matmul_dv ! multiply a dual matrix with a dual vector
        module procedure matmul_vd ! multiply a dual vector with a dual matrix
    end interface
```

```
    public max
    interface max
        module procedure max_dd ! max of from two to four dual numbers, elemental
        module procedure max_di ! max of a dual number and an integer, elemental
        module procedure max_dr ! max of a dual number and a real, elemental
        module procedure max_rd ! max of a real,and a dual number,  elemental
    end interface

    public dmax1
    interface dmax1
        module procedure max_dd ! max of from two to four dual numbers, elemental
    end interface

    public maxval
    interface maxval
        module procedure maxval_d ! maxval of a dual number vector
    end interface

    public min
    interface min
        module procedure min_dd ! min of from two to four dual numbers, elemental
        module procedure min_dr ! min of a dual and a real, elemental
    end interface

    public dmin1
    interface dmin1
        module procedure min_dd ! min of from two to four dual numbers, elemental
    end interface

    public minval
    interface minval
        module procedure minval_d ! obtain the maxval  of a dual number vectgor
    end interface

    public nint
    interface nint
        module procedure nint_d ! nearest integer to the argument, elemental
    end interface

    public sign
    interface  sign
      module procedure  sign_dd ! sign(a,b) with two dual numbers, elemental
      module procedure  sign_rd ! sign(a,b) with a real and a dual, elemental
    end interface

    public sin
    interface sin
        module procedure sin_d ! obtain sine of a dual number, elemental
    end interface

    public dsin
    interface dsin
        module procedure sin_d ! obtain sine of a dual number, elemental
    end interface

```

```
    public tan
    interface tan
        module procedure tan_d ! obtain sine of a dual number, elemental
    end interface

    public dtan
    interface dtan
        module procedure tan_d ! obtain sine of a dual number, elemental
    end interface

    public sqrt
    interface sqrt
        module procedure sqrt_d ! obtain the sqrt of a dual number, elemental
    end interface

    public sum
    interface sum
        module procedure sum_d ! sum a dual array
    end interface

    public maxloc
    interface maxloc
        module procedure maxloc_d ! location of max in a dual array
    end interface

contains

!*********Begin: functions/subroutines for overloading operators

!******* Begin: (=)
!---------------------

    !----------------------------------------
    ! dual = integer
    ! <u, du> = <i, 0>
    !----------------------------------------
    elemental subroutine assign_di(u, i)
         type(dual), intent(out) :: u
         integer, intent(in) :: i

         u%x = real(i)  ! This is faster than direct assignment
         u%dx = 0.0

    end subroutine assign_di


    !----------------------------------------
    ! dual = real(double)
    ! <u, du> = <r, 0>
    !----------------------------------------
    elemental subroutine assign_dr(u, r)
        type(dual), intent(out) :: u
        real, intent(in) :: r

        u%x = r
        u%dx = 0.0

    end subroutine assign_dr


    !----------------------------------------
```

```
    ! integer = dual
    ! i = <u, du>
    !----------------------------------------
    elemental subroutine assign_id(i, v)
        type(dual), intent(in) :: v
        integer, intent(out) :: i

        i = int(v%x)

    end subroutine assign_id

!******* End: (=)
!---------------------


!******* Begin: (+)
!---------------------

    !----------------------------------------
    ! Unary positive
    ! <res, dres> = +<u, du>
    !----------------------------------------
    elemental function add_d(u) result(res)
        type(dual), intent(in) :: u
        type(dual) :: res

        res = u  ! Faster than assigning component wise

    end function add_d


    !----------------------------------------
    ! dual + dual
    ! <res, dres> = <u, du> + <v, dv> = <u + v, du + dv>
    !----------------------------------------
    elemental function add_dd(u, v) result(res)
        type(dual), intent(in) :: u, v
        type(dual) :: res

        res%x = u%x + v%x
        res%dx = u%dx + v%dx

    end function add_dd


    !----------------------------------------
    ! dual + integer
    ! <res, dres> = <u, du> + i = <u + i, du>
    !----------------------------------------
    elemental function add_di(u, i) result(res)
        type(dual), intent(in) :: u
        integer, intent(in) :: i
        type(dual) :: res

        res%x = real(i) + u%x
        res%dx = u%dx

    end function add_di


    !----------------------------------------
```

```
    ! dual + double
    ! <res, dres> = <u, du> + <r, 0> = <u + r, du>
    !----------------------------------------
    elemental function add_dr(u, r) result(res)
        type(dual), intent(in) :: u
        real, intent(in) :: r
        type(dual) :: res

        res%x = r + u%x
        res%dx = u%dx

    end function add_dr


    !----------------------------------------
    ! integer + dual
    ! <res, dres> = <i, 0> + <v, dv> = <i + v, dv>
    !----------------------------------------
    elemental function add_id(i, v) result(res)
        integer, intent(in) :: i
        type(dual), intent(in) :: v
        type(dual) :: res

        res%x = real(i) + v%x
        res%dx = v%dx

    end function add_id


    !----------------------------------------
    ! double + dual
    ! <res, dres> = <r, 0> + <v, dv> = <r + v, dv>
    !----------------------------------------
    elemental function add_rd(r, v) result(res)
        real, intent(in) :: r
        type(dual), intent(in) :: v
        type(dual) :: res

        res%x = r + v%x
        res%dx = v%dx

    end function add_rd

!******* End: (+)
!---------------------


!******* Begin: (-)
!---------------------

    !------------------------------------------------
    ! negate a dual
    ! <res, dres> = -<u, du>
    !------------------------------------------------
    elemental function minus_d(u) result(res)
        type(dual), intent(in) :: u
        type(dual) :: res

        res%x = -u%x
        res%dx = -u%dx

```

```
    end function minus_d


    !-----------------------------------------------
    ! dual - dual
    ! <res, dres> = <u, du> - <v, dv> = <u - v, du - dv>
    !-----------------------------------------------
    elemental function minus_dd(u, v) result(res)
        type(dual), intent(in) :: u, v
        type(dual) :: res

        res%x = u%x - v%x
        res%dx = u%dx - v%dx

    end function minus_dd

    !-----------------------------------------------
    ! dual - integer
    ! <res, dres> = <u, du> - i = <u - i, du>
    !-----------------------------------------------
    elemental function minus_di(u, i) result(res)
        type(dual), intent(in) :: u
        integer, intent(in) :: i
        type(dual) :: res

        res%x = u%x - real(i)
        res%dx = u%dx

    end function minus_di


    !-----------------------------------------------
    ! dual - double
    ! <res, dres> = <u, du> - r = <u - r, du>
    !-----------------------------------------------
    elemental function minus_dr(u, r) result(res)
        type (dual), intent(in) :: u
        real,intent(in) :: r
        type(dual) :: res

        res%x = u%x - r
        res%dx = u%dx

    end function minus_dr


    !-----------------------------------------------
    ! integer - dual
    ! <res, dres> = i - <v, dv> = <i - v, -dv>
    !-----------------------------------------------
    elemental function minus_id(i, v) result(res)
        integer, intent(in) :: i
        type(dual), intent(in) :: v
        type(dual) :: res

        res%x = real(i) - v%x
        res%dx = -v%dx

    end function minus_id
```

```
    !--------------------------------------------------
    ! double - dual
    ! <res, dres> = r - <v, dv> = <r - v, -dv>
    !--------------------------------------------------
    elemental function minus_rd(r, v) result(res)
         real, intent(in) :: r
         type(dual), intent(in) :: v
         type(dual) :: res

        res%x = r - v%x
        res%dx = -v%dx

    end function minus_rd

!******* END: (-)
!---------------------


!******* BEGIN: (*)
!---------------------

    !----------------------------------------
    ! dual * dual
    ! <res, dres> = <u, du> * <v, dv> = <u * v, u * dv + v * du>
    !----------------------------------------
    elemental function mult_dd(u, v) result(res)
        type(dual), intent(in) :: u, v
        type(dual) :: res

        res%x = u%x * v%x
        res%dx = u%x * v%dx + v%x * u%dx

    end function mult_dd


    !-----------------------------------------
    ! dual * integer
    ! <res, dres> = <u, du> * i = <u * i, du * i>
    !-----------------------------------------
    elemental function mult_di(u, i) result(res)
        type(dual), intent(in) :: u
        integer, intent(in) :: i
        type(dual) :: res

        real :: r

        r = real(i)
        res%x = r * u%x
        res%dx = r * u%dx

    end function mult_di

    !-----------------------------------------
    ! dual * double
    ! <res, dres> = <u, du> * r = <u * r, du * r>
    !----------------------------------------
    elemental function mult_dr(u, r) result(res)
        type(dual), intent(in) :: u
        real, intent(in) :: r
        type(dual) :: res

```

```
        res%x = u%x * r
        res%dx = u%dx * r

    end function mult_dr


    !-----------------------------------------
    ! integer * dual
    ! <res, dres> = i * <v, dv> = <i * v, i * dv>
    !-----------------------------------------
    elemental function mult_id(i, v) result(res)
        integer, intent(in) :: i
        type(dual), intent(in) :: v
        type(dual) :: res

        real :: r

        r = real(i)
        res%x = r * v%x
        res%dx = r * v%dx

    end function mult_id


    !-----------------------------------------
    ! double * dual
    ! <res, dres> = r * <v, dv> = <r * v, r * dv>
    !-----------------------------------------
    elemental function mult_rd(r, v) result(res)
        real, intent(in) :: r
        type(dual), intent(in) :: v
        type(dual) :: res

        res%x = r * v%x
        res%dx = r * v%dx

    end function mult_rd

!******* END: (*)
!---------------------


!******* BEGIN: (/)
!---------------------

    !-----------------------------------------
    ! dual / dual
    ! <res, dres> = <u, du> / <v, dv> = <u / v, du / v - u * dv / v^2>
    !-----------------------------------------
    elemental function div_dd(u, v) result(res)
        type(dual), intent(in) :: u, v
        type(dual) :: res

        real :: inv

        inv = 1.0 / v%x
        res%x = u%x * inv
        res%dx = (u%dx - res%x * v%dx) * inv

    end function div_dd

```

```

    !----------------------------------------
    ! dual / integer
    ! <res, dres> = <u, du> / i = <u / i, du / i>
    !----------------------------------------
    elemental function div_di(u, i) result(res)
        type(dual), intent(in) :: u
        integer, intent(in) :: i
        type(dual) :: res

        real :: inv

        inv = 1.0 / real(i)
        res%x = u%x * inv
        res%dx = u%dx * inv

    end function div_di


    !----------------------------------------
    ! dual / double
    ! <res, dres> = <u, du> / r = <u / r, du / r>
    !----------------------------------------
    elemental function div_dr(u, r) result(res)
        type(dual), intent(in) :: u
        real, intent(in) :: r
        type(dual):: res

        real :: inv

        inv = 1.0 / r
        res%x = u%x * inv
        res%dx = u%dx * inv

    end function div_dr


    !----------------------------------------
    ! integer / dual
    ! <res, dres> = i / <v, dv> = <i / v, -i / v^2 * du>
    !----------------------------------------
    elemental function div_id(i, v) result(res)
        integer, intent(in) :: i
        type(dual), intent(in) :: v
        type(dual) :: res

        real :: inv

        inv = 1.0 / v%x
        res%x = real(i) * inv
        res%dx = -res%x * inv * v%dx

    end function div_id


    !----------------------------------------
    ! double / dual
    ! <res, dres> = r / <u, du> = <r / u, -r / u^2 * du>
    !----------------------------------------
    elemental function div_rd(r, v) result(res)
        real, intent(in) :: r
```

```
        type(dual), intent(in) :: v
        type(dual) :: res

        real :: inv

        inv = 1.0 / v%x
        res%x = r * inv
        res%dx = -res%x * inv * v%dx

    end function div_rd

!******* END: (/)
!---------------------

!******* BEGIN: (**)
!---------------------

    !----------------------------------------
    ! power(dual, integer)
    ! <res, dres> = <u, du> ^ i = <u ^ i, i * u ^ (i - 1) * du>
    !----------------------------------------
    elemental function pow_i(u, i) result(res)
        type(dual), intent(in) :: u
        integer, intent(in) :: i
        type(dual) :: res

        real :: pow_x

        pow_x = u%x ** (i - 1)
        res%x = u%x * pow_x
        res%dx = real(i) * pow_x * u%dx

    end function pow_i

    !----------------------------------------
    ! power(dual, double)
    ! <res, dres> = <u, du> ^ r = <u ^ r, r * u ^ (r - 1) * du>
    !----------------------------------------
    elemental function pow_r(u, r) result(res)
        type(dual), intent(in) :: u
        real, intent(in) :: r
        type(dual) :: res

        real :: pow_x

        pow_x = u%x ** (r - 1.0)
        res%x = u%x * pow_x
        res%dx = r * pow_x * u%dx

    end function pow_r

    !----------------------------------------
    ! POWER dual numbers to a dual power
    ! <res, dres> = <u, du> ^ <v, dv>
    !     = <u ^ v, u ^ v * (v / u * du + Log(u) * dv)>
    !----------------------------------------
    elemental function pow_d(u, v) result(res)
        type(dual), intent(in)::u, v
        type(dual) :: res

        res%x = u%x ** v%x
```

```
            res%dx = res%x * (v%x / u%x * u%dx + log(u%x) * v%dx)

        end function pow_d

!******* END: (**)
!---------------------


!******* BEGIN: (==)
!---------------------
    !----------------------------------------
    ! compare two dual numbers,
    ! simply compare the functional value.
    !----------------------------------------
    elemental function eq_dd(lhs, rhs) result(res)
         type(dual), intent(in) :: lhs, rhs
         logical :: res

         res = (lhs%x == rhs%x)

    end function eq_dd


    !----------------------------------------
    ! compare a dual with an integer,
    ! simply compare the functional value.
    !----------------------------------------
    elemental function eq_di(lhs, rhs) result(res)
         type(dual), intent(in) :: lhs
         integer, intent(in) :: rhs
         logical :: res

         res = (lhs%x == real(rhs))

    end function eq_di


    !----------------------------------------
    ! compare a dual number with a real number,
    ! simply compare the functional value.
    !----------------------------------------
    elemental function eq_dr(lhs, rhs) result(res)
        type(dual), intent(in) :: lhs
        real, intent(in) :: rhs
        logical::res

        res = (lhs%x == rhs)

    end function eq_dr


    !----------------------------------------
    ! compare an integer with a dual,
    ! simply compare the functional value.
    !---------------------------------------
    elemental function eq_id(lhs, rhs) result(res)
         integer, intent(in) :: lhs
         type(dual), intent(in) :: rhs
         logical :: res

         res = (lhs == rhs%x)
```

```

    end function eq_id


    !----------------------------------------
    ! compare a real with a dual,
    ! simply compare the functional value.
    !----------------------------------------
    elemental function eq_rd(lhs, rhs) result(res)
         real, intent(in) :: lhs
         type(dual), intent(in) :: rhs
         logical :: res

         res = (lhs == rhs%x)

    end function eq_rd

!******* END: (==)
!---------------------


!******* BEGIN: (<=)
!---------------------
    !----------------------------------------
    ! compare two dual numbers, simply compare
    ! the functional value.
    !----------------------------------------
    elemental function le_dd(lhs, rhs) result(res)
         type(dual), intent(in) :: lhs, rhs
         logical :: res

         res = (lhs%x <= rhs%x)

    end function le_dd


    !----------------------------------------
    ! compare a dual with an integer,
    ! simply compare the functional value.
    !----------------------------------------
    elemental function le_di(lhs, rhs) result(res)
         type(dual), intent(in) :: lhs
         integer, intent(in) :: rhs
         logical :: res

         res = (lhs%x <= rhs)

    end function le_di


    !----------------------------------------
    ! compare a dual number with a real number,
    ! simply compare the functional value.
    !----------------------------------------
    elemental function le_dr(lhs, rhs) result(res)
         type(dual), intent(in) :: lhs
         real, intent(in) :: rhs
         logical :: res

         res = (lhs%x <= rhs)

```

```
    end function le_dr


    !----------------------------------------
    ! compare a dual number with an integer,
    ! simply compare the functional value.
    !----------------------------------------
    elemental function le_id(i, rhs) result(res)
         integer, intent(in) :: i
         type(dual), intent(in) :: rhs
         logical :: res

         res = (i <= rhs%x)

    end function le_id


    !----------------------------------------
    ! compare a real with a dual,
    ! simply compare the functional value.
    !----------------------------------------
    elemental function le_rd(lhs, rhs) result(res)
         real, intent(in) :: lhs
         type(dual), intent(in) :: rhs
         logical :: res

         res = (lhs <= rhs%x)

    end function le_rd

!******* END: (<=)
!---------------------

!******* BEGIN: (<)
!---------------------
    !----------------------------------------
    ! compare two dual numbers, simply compare
    ! the functional value.
    !----------------------------------------
    elemental function lt_dd(lhs, rhs) result(res)
        type(dual), intent(in) :: lhs, rhs
        logical :: res

        res = (lhs%x < rhs%x)

    end function lt_dd

    !----------------------------------------
    ! compare a dual with an integer,
    ! simply compare the functional value.
    !----------------------------------------
    elemental function lt_di(lhs, rhs) result(res)
        type(dual), intent(in) :: lhs
        integer, intent(in) :: rhs
        logical :: res

        res = (lhs%x < rhs)

    end function lt_di
```

```
    !----------------------------------------
    ! compare a dual number with a real number, simply compare
    ! the functional value.
    !---------------------------------------
    elemental function lt_dr(lhs, rhs) result(res)
        type(dual), intent(in) :: lhs
        real, intent(in) :: rhs
        logical :: res

        res = (lhs%x < rhs)

    end function lt_dr


    !----------------------------------------
    ! compare a dual number with an integer
    !----------------------------------------
    elemental function lt_id(i, rhs) result(res)
         integer, intent(in) :: i
         type(dual), intent(in) :: rhs
         logical :: res

         res = (i < rhs%x)

    end function lt_id


    !----------------------------------------
    ! compare a real with a dual
    !---------------------------------------
    elemental function lt_rd(lhs, rhs) result(res)
         real, intent(in) :: lhs
         type(dual), intent(in) :: rhs
         logical :: res

         res = (lhs < rhs%x)

    end function lt_rd

!******* END: (<)
!---------------------

!******* BEGIN: (>=)
!---------------------
    !----------------------------------------
    ! compare two dual numbers, simply compare
    ! the functional value.
    !---------------------------------------
    elemental function ge_dd(lhs, rhs) result(res)
        type(dual), intent(in) :: lhs, rhs
        logical :: res

        res = (lhs%x >= rhs%x)

    end function ge_dd


    !----------------------------------------
    ! compare a dual with an integer
    !----------------------------------------
    elemental function ge_di(lhs, rhs) result(res)
```

```
        type(dual), intent(in) :: lhs
        integer, intent(in) :: rhs
        logical :: res

        res = (lhs%x >= rhs)

    end function ge_di


    !-----------------------------------------
    ! compare a dual number with a real number, simply compare
    ! the functional value.
    !-----------------------------------------
    elemental function ge_dr(lhs, rhs) result(res)
        type(dual), intent(in) :: lhs
        real, intent(in) :: rhs
        logical :: res

        res = (lhs%x >= rhs)

    end function ge_dr


    !-----------------------------------------
    ! compare a dual number with an integer
    !-----------------------------------------
    elemental function ge_id(i, rhs) result(res)
        integer, intent(in) :: i
        type(dual), intent(in) :: rhs
        logical :: res

        res = (i >= rhs%x)

    end function ge_id


    !-----------------------------------------
    ! compare a real with a dual
    !-----------------------------------------
    elemental function ge_rd(lhs, rhs) result(res)
         real, intent(in) :: lhs
         type(dual), intent(in) :: rhs
         logical :: res

         res = (lhs >= rhs%x)

    end function ge_rd

!******* END: (>=)
!---------------------

!******* BEGIN: (>)
!---------------------
    !-----------------------------------------
    ! compare two dual numbers, simply compare
    ! the functional value.
    !-----------------------------------------
    elemental function gt_dd(lhs, rhs) result(res)
        type(dual), intent(in) :: lhs, rhs
        logical :: res

```

```
        res = (lhs%x > rhs%x)

    end function gt_dd


    !-----------------------------------------
    ! compare a dual with an integer
    !-----------------------------------------
    elemental function gt_di(lhs, rhs) result(res)
        type(dual), intent(in) :: lhs
        integer, intent(in) :: rhs
        logical :: res

        res = (lhs%x > rhs)

    end function gt_di


    !-----------------------------------------
    ! compare a dual number with a real number, simply compare
    ! the functional value.
    !-----------------------------------------
    elemental function gt_dr(lhs, rhs) result(res)
        type(dual), intent(in) :: lhs
        real, intent(in) :: rhs
        logical :: res

        res = (lhs%x > rhs)

    end function gt_dr


    !-----------------------------------------
    ! compare a dual number with an integer
    !-----------------------------------------
    elemental function gt_id(i, rhs) result(res)
        integer, intent(in) :: i
        type(dual), intent(in) :: rhs
        logical :: res

        res = (i > rhs%x)

    end function gt_id


    !-----------------------------------------
    ! compare a real with a dual
    !-----------------------------------------
    elemental function gt_rd(lhs, rhs) result(res)
         real, intent(in) :: lhs
         type(dual), intent(in) :: rhs
         logical :: res

         res = (lhs > rhs%x)

    end function gt_rd

!******* END: (>)
!---------------------

!******* BEGIN: (/=)
```

```fortran
!---------------------
    !-----------------------------------------
    ! compare two dual numbers, simply compare
    ! the functional value.
    !-----------------------------------------
    elemental function ne_dd(lhs, rhs) result(res)
        type(dual), intent(in) :: lhs, rhs
        logical :: res

        res = (lhs%x /= rhs%x)

    end function ne_dd


    !-----------------------------------------
    ! compare a dual with an integer
    !-----------------------------------------
    elemental function ne_di(lhs, rhs) result(res)
        type(dual), intent(in) :: lhs
        integer, intent(in) :: rhs
        logical :: res

        res = (lhs%x /= rhs)

    end function ne_di


    !-----------------------------------------
    ! compare a dual number with a real number, simply compare
    ! the functional value.
    !-----------------------------------------
    elemental function ne_dr(lhs, rhs) result(res)
        type(dual), intent(in) :: lhs
        real, intent(in) :: rhs
        logical :: res

        res = (lhs%x /= rhs)

    end function ne_dr


    !-----------------------------------------
    ! compare a dual number with an integer
    !-----------------------------------------
    elemental function ne_id(i, rhs) result(res)
        integer, intent(in) :: i
        type(dual), intent(in) :: rhs
        logical :: res

        res = (i /= rhs%x)

    end function ne_id


    !-----------------------------------------
    ! compare a real with a dual
    !-----------------------------------------
    elemental function ne_rd(lhs, rhs) result(res)
        real, intent(in) :: lhs
        type(dual), intent(in) :: rhs
        logical :: res
```

```

        res = (lhs /= rhs%x)

    end function ne_rd

!******* END: (/=)
!---------------------

    !----------------------------------------------------
    ! Absolute value of dual numbers
    ! <res, dres> = abs(<u, du>) = <abs(u), du * sign(u)>
    !----------------------------------------------------
    elemental function abs_d(u) result(res)
         type(dual), intent(in) :: u
         type(dual) :: res
         integer :: i

         if(u%x > 0) then
            res%x = u%x
            res%dx = u%dx
         else if (u%x < 0) then
            res%x = -u%x
            res%dx = -u%dx
         else
            res%x = 0.0
            do i = 1, ndv
                if (u%dx(i) .eq. 0.0) then
                    res%dx(i) = 0.0
                else
                    res%dx(i) = set_NaN()
                end if
            end do
         endif

    end function abs_d


    !-----------------------------------------
    ! ACOS of dual numbers
    ! <res, dres> = acos(<u, du>) = <acos(u), -du / sqrt(1 - u^2)>
    !----------------------------------------
    elemental function acos_d(u) result(res)
        type(dual), intent(in) :: u
        type(dual) :: res

        res%x = acos(u%x)
        if (u%x == 1.0 .or. u%x == -1.0) then
            res%dx = set_Nan()  ! Undefined derivative
        else
            res%dx = -u%dx / sqrt(1.0 - u%x**2)
        end if

    end function acos_d


    !-----------------------------------------
    ! ASIN of dual numbers
    ! <res, dres> = asin(<u, du>) = <asin(u), du / sqrt(1 - u^2)>
    !----------------------------------------
    elemental function asin_d(u) result(res)
        type(dual), intent(in) :: u
```

```
        type(dual) :: res

        res%x = asin(u%x)
        if (u%x == 1.0 .or. u%x == -1.0) then
            res%dx = set_NaN()  ! Undefined derivative
        else
            res%dx = u%dx / sqrt(1.0 - u%x**2)
        end if

    end function asin_d


    !-----------------------------------------
    ! ATAN of dual numbers
    ! <res, dres> = atan(<u, du>) = <atan(u), du / (1 + u^2)>
    !----------------------------------------
    elemental function atan_d(u) result(res)
        type(dual), intent(in) :: u
        type(dual) :: res

        res%x = atan(u%x)
        res%dx = u%dx / (1.0 + u%x**2)

    end function atan_d


    !-----------------------------------------
    ! ATAN2 of dual numbers
    ! <res, dres> = atan2(<u, du>, <v, dv>)
    !             = <atan2(u, v), v / (u^2 + v^2) * du - u / (u^2 + v^2) * dv>
    !----------------------------------------
    elemental function atan2_d(u, v) result(res)
        type(dual), intent(in) :: u, v
        type(dual) :: res

        real :: usq_plus_vsq

        res%x = atan2(u%x, v%x)

        usq_plus_vsq = u%x**2 + v%x**2
        res%dx = v%x / usq_plus_vsq * u%dx - u%x / usq_plus_vsq * v%dx

    end function atan2_d


    !-----------------------------------------
    ! COS of dual numbers
    ! <res, dres> = cos(<u, du>) = <cos(u), -sin(u) * du>
    !----------------------------------------
    elemental function cos_d(u) result(res)
        type(dual), intent(in) :: u
        type(dual) :: res

        res%x = cos(u%x)
        res%dx = -sin(u%x) * u%dx

    end function cos_d


    !-----------------------------------------
    ! DOT PRODUCT two dual number vectors
```

```
! <res, dres> = <u, du> . <v, dv> = <u . v, u . dv + v . du>
!-----------------------------------------
function dot_product_dd(u, v) result(res)
    type(dual), intent(in) :: u(:), v(:)
    type(dual) :: res

    integer :: i

    res%x = dot_product(u%x, v%x)
    do i = 1, ndv
        res%dx(i) = dot_product(u%x, v%dx(i)) + dot_product(v%x, u%dx(i))
    end do

end function dot_product_dd


!-----------------------------------------
! EXPONENTIAL OF dual numbers
! <res, dres> = exp(<u, du>) = <exp(u), exp(u) * du>
!-----------------------------------------
elemental function exp_d(u) result(res)
    type(dual), intent(in) :: u
    type(dual) :: res

    real :: exp_x

    exp_x = exp(u%x)
    res%x = exp_x
    res%dx = u%dx * exp_x

end function exp_d


!-----------------------------------------
! Convert dual to integer
! i = int(<u, du>) = int(u)
!----------------------------------------
elemental function int_d(u) result(res)
     type(dual), intent(in) :: u
     integer :: res

     res = int(u%x)

end function int_d


!-----------------------------------------
! LOG OF dual numbers,defined for u%x>0 only
! the error control should be done in the original code
! in other words, if u%x<=0, it is not possible to obtain LOG.
! <res, dres> = log(<u, du>) = <log(u), du / u>
!----------------------------------------
elemental function log_d(u) result(res)
    type(dual), intent(in) :: u
    type(dual) :: res

    real :: inv

    inv = 1.0 / u%x
    res%x = log(u%x)
    res%dx = u%dx * inv
```

```fortran

    end function log_d


    !----------------------------------------
    ! LOG10 OF dual numbers,defined for u%x>0 only
    ! the error control should be done in the original code
    ! in other words, if u%x<=0, it is not possible to obtain LOG.
    ! <res, dres> = log10(<u, du>) = <log10(u), du / (u * log(10))>
    ! LOG<u,up>=<LOG(u),up/u>
    !----------------------------------------
    elemental function log10_d(u) result(res)
        type(dual), intent(in) :: u
        type(dual) :: res

        real :: inv

        inv = 1.0 / (u%x * log(10.0))
        res%x = log10(u%x)
        res%dx = u%dx * inv

    end function log10_d


    !----------------------------------------
    ! MULTIPLY two dual number matrices
    ! <res, dres> = <u, du> . <v, dv> = <u . v, du . v + u . dv>
    !----------------------------------------
    function matmul_dd(u,v) result(res)
        type(dual), intent(in) :: u(:,:), v(:,:)
        type(dual) :: res(size(u,1), size(v,2))

        integer :: i

        res%x = matmul(u%x, v%x)
        do i = 1, ndv
            res%dx(i) = matmul(u%dx(i), v%x) + matmul(u%x, v%dx(i))
        end do

    end function matmul_dd


    !----------------------------------------
    ! MULTIPLY a dual number matrix with a dual number
    ! vector
    !
    ! <u,up>.<v,vp>=<u.v,up.v+u.vp>
    !----------------------------------------
    function matmul_dv(u, v) result(res)
        type(dual), intent(in) :: u(:,:), v(:)
        type(dual) :: res(size(u,1))
        integer :: i

        res%x = matmul(u%x, v%x)
        do i = 1, ndv
            res%dx(i) = matmul(u%dx(i), v%x) + matmul(u%x, v%dx(i))
        end do

    end function matmul_dv

```

```fortran
!-----------------------------------------
! MULTIPLY a dual vector with a  dual matrix
!
! <u,up>.<v,vp>=<u.v,up.v+u.vp>
!----------------------------------------
function matmul_vd(u, v) result(res)
    type(dual), intent(in) :: u(:), v(:,:)
    type(dual) :: res(size(v, 2))
    integer::i

    res%x = matmul(u%x, v%x)
    do i = 1, ndv
        res%dx(i) = matmul(u%dx(i), v%x) + matmul(u%x, v%dx(i))
    end do

end function matmul_vd

!-----------------------------------------
! Obtain the max of 2 to 5 dual numbers
!----------------------------------------
elemental function max_dd(val1, val2, val3, val4,val5) result(res)
    type(dual), intent(in) :: val1, val2
    type(dual), intent(in), optional :: val3, val4,val5
    type(dual) :: res

    if (val1%x > val2%x) then
        res = val1
    else
        res = val2
    endif
    if(present(val3))then
       if(res%x < val3%x) res = val3
    endif
    if(present(val4))then
       if(res%x < val4%x) res = val4
    endif
    if(present(val5))then
       if(res%x < val5%x) res = val5
    endif

end function max_dd


!-----------------------------------------
! Obtain the max of a dual number and an integer
!----------------------------------------
elemental function max_di(u, i) result(res)
    type(dual), intent(in) :: u
    integer, intent(in) :: i
    type(dual) :: res

    if (u%x > i) then
        res = u
    else
        res = i
    endif

end function max_di

!-----------------------------------------
! Obtain the max of a dual number and a real number
```

```
!----------------------------------------
elemental function max_dr(u, r) result(res)
    type(dual), intent(in) :: u
    real, intent(in) :: r
    type(dual) :: res

    if (u%x > r) then
        res = u
    else
        res = r
    endif

end function max_dr


!---------------------------------------------------
! Obtain the max of a real and a dual
!---------------------------------------------------
 elemental function max_rd(n, u) result(res)
    real, intent(in) :: n
    type(dual), intent(in) :: u
    type(dual) :: res

    if (u%x > n) then
        res = u
    else
        res = n
    endif

end function max_rd


!-----------------------------------------
! Obtain the max value of vector u
!----------------------------------------
function maxval_d(u) result(res)
    type(dual), intent(in) :: u(:)
    integer :: iloc(1)
    type(dual) :: res

    iloc=maxloc(u%x)
    res=u(iloc(1))

end function maxval_d


!-----------------------------------------
! Obtain the min of 2 to 4 dual numbers
!----------------------------------------
elemental function min_dd(val1, val2, val3, val4) result(res)
    type(dual), intent(in) :: val1, val2
    type(dual), intent(in), optional :: val3, val4
    type(dual) :: res

    if (val1%x < val2%x) then
        res = val1
    else
        res = val2
    endif
    if(present(val3))then
       if(res%x > val3%x) res = val3
```

```
        endif
        if(present(val4))then
           if(res%x > val4%x) res = val4
        endif

    end function min_dd


    !----------------------------------------
    ! Obtain the min of a dual and a double
    !---------------------------------------
    elemental function min_dr(u, r) result(res)
        type(dual), intent(in) :: u
        real, intent(in) :: r
        type(dual) :: res

        if (u%x < r) then
            res = u
        else
            res = r
        endif

    end function min_dr


  !----------------------------------------
    ! Obtain the min value of vector u
    !---------------------------------------
    function minval_d(u) result(res)
        type(dual), intent(in) :: u(:)
        integer :: iloc(1)
        type(dual) :: res

        iloc=minloc(u%x)
        res=u(iloc(1))

    end function minval_d


    !------------------------------------------------------
    !Returns the nearest integer to u%x, ELEMENTAL
    !------------------------------------------------------
    elemental function nint_d(u) result(res)
        type(dual), intent(in) :: u
        integer :: res

        res=nint(u%x)

    end function nint_d


    !----------------------------------------------------------------
    ! SIGN(a,b) with two dual numbers as inputs,
    ! the result will be |a| if b%x>=0, -|a| if b%x<0,ELEMENTAL
    !----------------------------------------------------------------
    elemental function sign_dd(val1, val2) result(res)
        type(dual), intent(in) :: val1, val2
        type(dual) :: res

        if (val2%x < 0.0) then
            res = -abs(val1)
```

```
        else
            res =  abs(val1)
        endif

    end function sign_dd


   !--------------------------------------------------------------
   ! SIGN(a,b) with one real and one dual number as inputs,
   ! the result will be |a| if b%x>=0, -|a| if b%x<0,ELEMENTAL
   !--------------------------------------------------------------
   elemental function sign_rd(val1, val2) result(res)
       real, intent(in) :: val1
       type(dual), intent(in) :: val2
       type(dual) :: res

       if (val2%x < 0.0) then
           res = -abs(val1)
       else
           res = abs(val1)
       endif

    end function sign_rd


   !----------------------------------------
   ! SIN of dual numbers
   ! <res, dres> = sin(<u, du>) = <sin(u), cos(u) * du>
   !---------------------------------------
   elemental function sin_d(u) result(res)
       type(dual), intent(in) :: u
       type(dual) :: res

       res%x = sin(u%x)
       res%dx = cos(u%x) * u%dx

   end function sin_d


   !----------------------------------------
   ! TAN of dual numbers
   ! <res, dres> = tan(<u, du>) = <tan(u), du / cos(u)^2>
   !---------------------------------------
   elemental function tan_d(u) result(res)
       type(dual), intent(in) :: u
       type(dual) :: res

       res%x = tan(u%x)
       res%dx = u%dx / cos(u%x)**2

   end function tan_d


   !----------------------------------------
   ! SQRT of dual numbers
   ! <res, dres> = sqrt(<u, du>) = <sqrt(u), du / (2 * sqrt(u))>
   !---------------------------------------
   elemental function sqrt_d(u) result(res)
       type(dual), intent(in) :: u
       type(dual) :: res
       integer :: i
```

```
        res%x = sqrt(u%x)

        if (res%x .ne. 0.0) then
            res%dx = 0.5 * u%dx / res%x
        else
            do i = 1, ndv
                if (u%dx(i) .eq. 0.0) then
                    res%dx(i) = 0.0
                else
                    res%dx(i) = set_NaN()
                end if
            end do
        end if

    end function sqrt_d


    !-----------------------------------------
    ! Sum of a dual array
    !-----------------------------------------
    function sum_d(u) result(res)
        type(dual), intent(in) :: u(:)
        type(dual) :: res
        integer :: i

        res%x = sum(u%x)
        do i = 1, ndv
            res%dx(i) = sum(u%dx(i))
        end do

    end function sum_d


    !-----------------------------------------
    ! Find the location of the max value in an
    ! array of dual numbers
    !-----------------------------------------
    function maxloc_d(array) result(ind)
        type(dual), intent(in) :: array(:)
        integer :: ind(1)

        ind = maxloc(array%x)

    end function maxloc_d


    elemental function set_NaN() result(res)
        real :: res

        res = sqrt(negative_one)

    end function set_NaN

end module dnadmod
```

B ONE-DIMENSIONAL SCALAR TRANSPORT SOLVER IN FORTRAN

B.1 User Interface

```fortran
program main
    use adpsolver
    use adpio
    implicit none

    type(adpmodel) :: solver  ! adpmodel object

    integer :: nargs  ! Number of command line arguments
    character*80 :: arg  ! Command line argument
    character*80 :: opt  ! Option from command line argument
    character*80 :: val  ! Value from command line argument

    character*80 :: infile  ! Input file name
    character*80 :: outfile  ! Output file name

    integer :: err  ! Error flag

    ! Get command line arguments
    nargs = command_argument_count()
    if (nargs .gt. 0) then
        call get_command_argument(1, infile)
    else
        write(*, *) "An input file must be specified."
        call exit(1)
    end if

    ! Initialize solver
    write(*, *) "Initializing solver..."
    call solver%init()

    ! Read the input file
    write(*, *) "Reading input file '", trim(infile), "'..."
    err = readInput(solver, infile)
    if (err .ne. 0) then
        write(*, *) "Error reading input file, aborting!"
        call exit(2)
    end if

    ! Calculate a solution
    write(*, *) "Solving..."
    err = solver%solve()
    if (err .ne. 0) then
        write(*, *) "Solution error, aborting!"
        call exit(3)
    end if

    ! Determine the output file name
    if (nargs .gt. 1) then
        call get_command_argument(2, outfile)
    else
        outfile = trim(solver%jobname) // ".out"
    end if

    ! Write the output
    write(*, *) "Writing output to file '", trim(outfile), "'..."
```

```
    err = writeOutput(solver, outfile)
    if (err .ne. 0) then
        write(*, *) "Error generating output file!"
        call exit(4)
    end if

    call solver%dealloc()

end program main
```

B.2 Physics Module

```
module adpsolver
    implicit none

    type :: adpmodel
        character*80 :: jobname  ! Job name

        real :: phi0  ! Scalar value at x = 0
        real :: phi1  ! Scalar value at x = 1

        real :: u  ! Velocity
        real :: gamma  ! Diffusivity
        real :: c  ! Proportionality constant for scalar production

        integer :: npts  ! Number of grid points on discretized domain
                         ! (spaced uniformly from x = 0 to x = 1)
        real :: dx  ! Distance between points in x array

        real, allocatable, dimension(:) :: x  ! array of grid coordinates
        real, allocatable, dimension(:) :: phi  ! scalar field function

    contains
        procedure :: init => adpmodel_init
        procedure :: alloc => adpmodel_alloc
        procedure :: dealloc => adpmodel_dealloc
        procedure :: solve => adpmodel_solve
        procedure :: solveImplicit => adpmodel_solveImplicit

    end type adpmodel

contains
    !!! Initialize an adpmodel object
    !!!
    !!! Input:
    !!!     this = adpmodel object to initialize
    subroutine adpmodel_init(this)
        class(adpmodel), intent(inout) :: this

        this%jobname = "ADPSolution1"

        this%phi0 = 1.0d0
        this%phi1 = 0.0d0

        this%u = 1.0d0
        this%gamma = 0.1d0
        this%c = -1.0d0

        this%npts = 11
    end subroutine adpmodel_init
```

```fortran


    !!! Allocate dynamic memory
    !!!
    !!! Input:
    !!!     this = adpmodel object
    !!!
    !!! Return:
    !!!     stat = Error status flag (0 = success,
    !!!                               1 = Allocation error)
    integer function adpmodel_alloc(this) result(stat)
        class(adpmodel), intent(inout) :: this

        integer :: n
        integer :: err

        ! Initialize error flags
        err = 0
        stat = 0

        call this%dealloc()

        n = this%npts
        allocate(this%x(n), this%phi(n), stat = err)
        if (err .ne. 0) then
            write(*, *) "Error: Could not allocate arrays!"
            stat = 1
        end if
    end function adpmodel_alloc


    !!! Deallocate dynamic memory
    !!!
    !!! Input:
    !!!     this = adpmodel object
    subroutine adpmodel_dealloc(this)
        class(adpmodel), intent(inout) :: this

        if (allocated(this%phi)) deallocate(this%phi)
        if (allocated(this%x)) deallocate(this%x)
    end subroutine adpmodel_dealloc


    !!! Run the solution
    !!!
    !!! Input:
    !!!     this = adpmodel object
    !!!
    !!! Return:
    !!!     stat = Error status (0 = success,
    !!!                          1 = problem not overdamped,
    !!!                          2 = error allocating arrays,
    !!!                          3 = error executing solver)
    integer function adpmodel_solve(this) result(stat)
        class(adpmodel), intent(inout) :: this

        integer :: err  ! Temporary error flag
        integer :: i  ! Loop control variable

        ! Initialize the error flags
        err = 0
```

```fortran
        stat = 0

        ! Make sure we are in the overdamped regime
        if (this%u**2 .le. 4.0d0 * this%c * this%gamma) then
            write(*, *) "Error: The specified properties do not meet ", &
                "the overdamping requirement for this analysis."
            stat = 1
            return
        end if

        ! Allocate solution arrays
        err = this%alloc()
        if (err .ne. 0) then
            stat = 2
            return
        end if

        ! Populate x array for npts
        this%dx = 1.0d0 / (this%nPts - 1)
        do i = 1, this%npts
            this%x(i) = (i - 1) * this%dx
        end do

        ! Solve the problem
        err = this%solveImplicit()

        ! Check solution status
        if (err .ne. 0) then
            stat = 3
            return
        end if

    end function adpmodel_solve


    !!! Solve the problem implicitly
    !!!
    !!! Inputs:
    !!!     this = AdvectionSolver object
    !!!
    !!! Return:
    !!!     stat = Error status (0 = success)
    integer function adpmodel_solveImplicit(this) result(stat)
        class(adpmodel), intent(inout) :: this

        integer :: i, j  ! Loop control variables
        integer :: n  ! Shortcut to npts
        real, dimension(:), allocatable :: ld  ! Lower diagonal vector
        real, dimension(:), allocatable :: md  ! Middle diagonal vector
        real, dimension(:), allocatable :: ud  ! Uppder diagonal vector
        real, dimension(:), allocatable :: bc  ! BC vector

        real :: aE, aW, aP  ! Matrix coefficients
        real :: w  ! temporary variable

        ! Initialize error flag
        stat = 0

        ! Allocate the solution matrix and BC vector
        n = this%npts
        allocate(bc(n), ld(n), md(n), ud(n))
```

```
        bc = 0.0d0
        ld = 0.0d0
        md = 0.0d0
        ud = 0.0d0

        ! Calculate the matrix coefficients (1st-order upwinding)
     aW = -this%gamma / this%dx**2; aE = aW
     if (this%u > 0.0d0) then
         aW = aW - this%u / this%dx
     else
         aE = aE + this%u / this%dx
     end if

        aP = -aE - aW - this%c

        ! Populate the diagonal and BC vectors
        bc(1) = this%phi0
        md(1) = 1.0d0
        do i = 2, n - 1
            bc(i) = 0.0d0
            ld(i) = aW
            md(i) = aP
            ud(i) = aE
        end do
        bc(n) = this%phi1
        md(n) = 1.0d0

        do i = 2, n
            w = ld(i) / md(i - 1)
            md(i) = md(i) - w * ud(i - 1)
            bc(i) = bc(i) - w * bc(i - 1)
        end do

        this%phi(n) = bc(n) / md(n)
        do i = n - 1, 1, -1
            this%phi(i) = (bc(i) - ud(i) * this%phi(i + 1)) / md(i)
        end do

        deallocate(bc, ld, md, ud)

    end function adpmodel_solveImplicit

end module adpsolver
```

B.3 I/O Module

```
module adpio
    use adpsolver
    implicit none

    integer, parameter :: ioUnit = 10  ! IO unit for files

contains
    !!! Read and parse the input file
    !!!
    !!! Input:
    !!!     model = adpmodel object
    !!!     filename = Name of input file to read/parse
    !!!
    !!! Return:
```

```
    !!!     stat = Error status (0 = success,
    !!!                          1 = error opening input file,
    !!!                          2 = error parsing card(s))
    integer function readInput(model, filename) result(stat)
        class(adpmodel), intent(inout) :: model
        character*80, intent(in) :: filename  ! Name of input file

        character*80 :: card
        character*80 :: id

        integer :: err  ! I/O error flag

        ! Initialize the error flags
        err = 0
        stat = 0

        ! Open the file for reading
        open(unit = ioUnit, file = filename, status = 'old', &
            &  action = 'read', iostat = err)
        if (err .ne. 0) then
            write(*, '(A)') "ERROR: The input file could not be read."
            write(*, '(A, A)') "    Filename = ", trim(filename)
            write(*, '(A, I3)') "    I/O Error = ", err
            close(ioUnit)
            stat = 1
            return
        end if

        ! Begin processing input cards
        read(ioUnit, '(A)', iostat = err) card
        do while (.not. is_iostat_end(err))
            ! Remove all whitespace from card
            card = removeWhitespace(card)

            if (card(1:1) .eq. '*') then
                write(*, '(A, 1X, A)') "Found card:", trim(card)
                err = parseCard(model, card)
                if (err .ne. 0) then
                    stat = 2
                end if
            else if (len(trim(card)) .gt. 1) then
                write(*, '(A, 1X, A)') "Found comment:", trim(card)
            end if

            ! Read the next card
            read(ioUnit, '(A)', iostat = err) card
        end do

        if (stat .eq. 2) then
            write(*, *) "Error: Some input cards were not processed."
        end if

        close(unit = ioUnit)

    end function readInput


    !!! Parse an input card and extract all parameter data
    !!!
    !!! Input:
    !!!     model = adpmodel object
```

```
!!!     card = Input card to parse and extract data from
!!!
!!! Return:
!!!     stat = Error status flag (0 = success,
!!!                               1 = I/O error parsing card,
!!!                               2 = error parsing words on card,
!!!                               3 = invalid word parameter name,
!!!                               4 = invalid card ID)
integer function parseCard(model, card) result(stat)
    class(adpmodel), intent(inout) :: model
    character*80, intent(in) :: card  ! Card string

    character*80 :: id  ! Card ID (first word on card)
    integer :: nWords  ! Number of words on card (excludes ID)
    character*80 :: word  ! List of words on card (excludes ID)

    integer :: i  ! Loop control variable
    integer :: ind  ! End index of last word
    integer :: err  ! Error flag

    character*80 :: paramName  ! Parameter name
    character*80 :: paramValue  ! Parameter value

    ! Initialize the error flags
    err = 0
    stat = 0

    ! Get the card ID (first word on card)
    read(card, *, iostat = err) id
    if (err .ne. 0) then
        write(*, *) "Unknown error parsing card ID:", trim(card)
        stat = 1
        return
    end if

    ! Get the number of remaining words
    nwords = count(transfer(card, 'A', len(trim(card))) == ",")

    ! Get the words
    ind = index(card, trim(id)) + len(trim(id)) + 1  ! Add 1 for comma
    do i = 1, nWords
        ! Read the next word from the card
        read(card(ind:80), *, iostat = err) word
        if (err .ne. 0) then
            write(*, '(A, I2, A, A)') "Unknown error parsing word ", &
                &   i, ":", trim(card)
            stat = 2
            return
        end if

        ! Get the starting index of the next word (add 1 for comma)
        ind = index(card, trim(word)) + len(trim(word)) + 1

        ! Parse the word to extract the parameter name and value
        err = parseArg(word, paramName, paramValue)
        if (err .ne. 0) then
            write(*, '(A, I2, A, A)') "Invalid string on word ", &
                &   i, ":", trim(card)
            stat = 2
            cycle
        end if
```

```
        ! Extract the data based on card type
        select case(id)
            case ("*job")
                err = setJobField(model, paramName, paramValue)
            case ("*bcs")
                err = setBCField(model, paramName, paramValue)
            case ("*props")
                err = setPropsField(model, paramName, paramValue)
            case ("*grid")
                err = setGridField(model, paramName, paramValue)
            case ("*solver")
                err = setSolverField(model, paramName, paramValue)
            case default
                write(*, '(2X, A, A, A)') "Unrecognized card ID: ", &
                    &   trim(id), ". Card skipped!"
                stat = 4
                return
        end select

        ! Check for an error setting fields
        if (err .ne. 0) then
            stat = 3
        end if
    end do

end function parseCard


!!! Set a field from the job card
!!!
!!! Input:
!!!     this = adpmodel object
!!!     paramName = Name of field to set
!!!     paramValue = String representation of parameter value
!!!
!!! Return:
!!!     stat = Error status flag (0 = success,
!!!                              1 = Unrecognized parameter name)
integer function setJobField(model, paramName, paramValue) result (stat)
    class(adpmodel), intent(inout) :: model
    character*80, intent(in) :: paramName  ! Parameter name
    character*80, intent(in) :: paramValue  ! Parameter value

    ! Initialize the error flag
    stat = 0

    select case(paramName)
        case ("name")
            model%jobName = paramValue
        case default
            write(*, '(6X, A)') &
                &   "Unrecognized parameter name on job card:", &
                &    trim(paramName)
            stat = 1
    end select

end function setJobField


!!! Set a field from the bcs card
```

```fortran
!!!
!!! Input:
!!!     model = adpmodel object
!!!     paramName = Name of field to set
!!!     paramValue = String representation of parameter value
!!!
!!! Return:
!!!     stat = Error status flag (0 = success,
!!!                                1 = Unrecognized parameter name)
integer function setBCField(model, paramName, paramValue) result(stat)
    class(adpmodel), intent(inout) :: model
    character*80, intent(in) :: paramName  ! Parameter name
    character*80, intent(in) :: paramValue  ! Parameter value

    ! Initialize the error flag
    stat = 0

    select case(paramName)
        case ("phi0")
            model%phi0 = stringToReal(paramValue)
        case ("phi1")
            model%phi1 = stringToReal(paramValue)
        case default
            write(*, '(6X, A, 1X, A)') &
                &   "Unrecognized parameter name on bcs card:", &
                &    trim(paramName)
            stat = 1
    end select
end function setBCField


!!! Set a field from the props card
!!!
!!! Input:
!!!     model = adpmodel object
!!!     paramName = Name of field to set
!!!     paramValue = String representation of parameter value
!!!
!!! Return:
!!!     stat = Error status flag (0 = success,
!!!                                1 = Unrecognized parameter name)
integer function setPropsField(model, paramName, paramValue) result(stat)
    class(adpmodel), intent(inout) :: model
    character*80, intent(in) :: paramName  ! Parameter name
    character*80, intent(in) :: paramValue  ! Parameter value

    ! Initialize the error flag
    stat = 0

    select case(paramName)
        case ("u")
            model%u = stringToReal(paramValue)
        case ("gamma")
            model%gamma = stringToReal(paramValue)
        case ("c")
            model%c = stringToReal(paramValue)
        case default
            write(*, '(6X, A, 1X, A)') &
                &   "Unrecognized parameter name on props card:", &
                &    trim(paramName)
            stat = 1
```

```
            end select
    end function setPropsField


    !!! Set a field from the grid card
    !!!
    !!! Input:
    !!!     model = adpmodel object
    !!!     paramName = Name of field to set
    !!!     paramValue = String representation of parameter value
    !!!
    !!! Return:
    !!!     stat = Error status flag (0 = success,
    !!!                               1 = Unrecognized parameter name)
    integer function setGridField(model, paramName, paramValue) result(stat)
        class(adpmodel), intent(inout) :: model
        character*80, intent(in) :: paramName  ! Parameter name
        character*80, intent(in) :: paramValue  ! Parameter value

        ! Initialize the error flag
        stat = 0

        select case(paramName)
            case ("npts")
                model%npts = stringToInt(paramValue)
            case default
                write(*, '(6X, A, 1X, A)') &
                    &   "Unrecognized parameter name on grid card:", &
                    &    trim(paramName)
                stat = 1
            end select
    end function setGridField


    !!! Set a field from the model card
    !!!
    !!! Input:
    !!!     model = adpsolver object
    !!!     paramName = Name of field to set
    !!!     paramValue = String representation of parameter value
    !!!
    !!! Return:
    !!!     stat = Error status flag (0 = success,
    !!!                               1 = Unrecognized parameter name)
    integer function setSolverField(model, paramName, paramValue) result(stat)
        class(adpmodel), intent(inout) :: model
        character*80, intent(in) :: paramName  ! Parameter name
        character*80, intent(in) :: paramValue  ! Parameter value

        ! Initialize the error flag
        stat = 0

        select case(paramName)
            case ("method")
                select case(paramValue)
                    case ("analytical")
                        model%method = analytical
                    case ("implicit")
                        model%method = implicit
                    case default
                        write(*, '(6X, A, 1X, A)') &
```

```
                        &   "Unrecognized solution method:", paramValue
            end select
        case ("derivatives")
            select case (paramValue)
                case ("yes")
                    model%computeDerivatives = .true.
                case ("no")
                    model%computeDerivatives = .false.
                case default
                    write(*, '(A)') "Derivatives option must be ", &
                        &   "yes/no. Defaulting to no."
            end select
        case default
            write(*, '(6X, A, 1X, A)') &
                &   "Unrecognized parameter name on solver card:", &
                &    trim(paramName)
            stat = 1
        end select

end function setSolverField


REAL function stringToReal(str) result (val)
    character*80, intent(in) :: str  ! String to convert

    integer :: err  ! I/O error status

    read(str, *, iostat = err) val
    if (err .ne. 0) then
        write(*, '(6X, A, A)') "Error converting string to real: ", str
    end if
end function stringToReal


integer function stringToInt(str) result (val)
    character*80, intent(in) :: str  ! String to convert

    integer :: err  ! I/O error status

    read(str, *, iostat = err) val
    if (err .ne. 0) then
        write(*, '(6X, A, A)') "Error converting string to int: ", str
    end if
end function stringToInt


!!! Write the results to an output file
!!!
!!! Inputs:
!!!     model = adpmodel object
!!!     filename_opt = Optional output filename (defaults to jobName.out)
!!!
!!! Return:
!!!     stat = Error status (0 = success,
!!!                          1 = Error opening output file)
integer function writeOutput(model, filename_opt) result(stat)
    class(adpmodel), intent(inout) :: model  ! adpmodel object
    character*80, intent(in), optional :: filename_opt  ! Optional filename

    character*80 :: filename  ! Output file name
    integer :: err  ! Error status flag
```

```
        stat = 0

        ! Get the filename to write to
        if (present(filename_opt)) then
            filename = filename_opt
        else
            filename = trim(model%jobname) // ".out"
        end if

        ! Open the output file for writing
        open(unit = ioUnit, file = filename, action = 'write', iostat = err)
        if (err .ne. 0) then
            write(*, *) "Error: Unable to open output file for writing."
            stat = 1
            return
        end if

        ! Write the header and results
        call writeData(model)

        ! Close the output file
        close(unit = ioUnit)

    end function writeOutput


    subroutine writeData(model)
        class(adpmodel), intent(inout) :: model  ! adpmodel object

        integer :: i, j  ! Loop control variable
        character*80 :: fmtString

        if (model%method .eq. analytical .and. model%computeDerivatives) then
            ! Header
            write(ioUnit, '(A)') "x,phi,dPhi/dX,dPhi/dU,dPhi/dGamma,dPhi/dC"

            ! Results
            write(fmtString, *) '(5(ES23.15, ", "), ES23.15)'
            do i=1, model%npts
                write(ioUnit, fmtString) model%x(i), model%phi(i), &
                    &   model%dPhi_dU(i), model%dPhi_dGamma(i), model%dPhi_dC(i)
            end do

        else
            ! Header
            write(ioUnit, '(A)') "x,phi"

            ! Results
            write(fmtString, *) '(ES23.15, ", ", ES23.15)'
            do i=1, model%npts
                write(ioUnit, fmtString) model%x(i), model%phi(i)
            end do
        end if
    end subroutine writeData


    character*80 function removeWhitespace(str) result(mstr)
        character*80, intent(in) :: str

        integer :: i  ! Loop control variable
```

```
        integer :: l  ! Length of modified string

        character, parameter :: tab = achar(9)  ! Tab character

        ! Initialize length of modified string to 0
        l = 0
        mstr = char(0)

        ! Loop over each character in unmodified string
        do i = 1, len(str)
            ! Only consider non-space, non-tab characters
            if (str(i:i) .ne. ' ' .and. str(i:i) .ne. tab) then
                l = l + 1  ! Increment the count
                mstr(l:l) = str(i:i)  ! Add the non-whitespace character
            end if
        end do
    end function removeWhitespace


    !!! Parse an argument to determine the name and value
    !!!
    !!! Input:
    !!!     arg = Argument string to parse
    !!!
    !!! Output:
    !!!     argName = name extracted from argument string
    !!!     argValue = value extracted from argument string
    !!!
    !!! Return:
    !!!     stat = Error status (0 = success,
    !!!                          1 = argument value missing,
    !!!                          2 = argument name and value missing)
    integer function parseArg(arg, argName, argValue) result(stat)
        character*80, intent(in) :: arg  ! Argument string to parse
        character*80, intent(out) :: argName  ! Name of argument
        character*80, intent(out) :: argValue  ! Value of argument

        character*80 :: trimmedArg
        integer :: ind  ! Index of equal sign in argument

        argName = ""
        argValue = ""

        trimmedArg = trim(adjustl(arg))
        ind = index(trimmedArg, "=")
        if (len(trimmedArg) .eq. 0) then
            stat = 2
            return
        else if (ind .eq. 0) then
            argName = trimmedArg
            stat = 1
        else
            argName = trimmedArg(1 : ind - 1)
            argValue = trimmedArg(ind + 1 :)
            stat = 0
        end if
    end function parseArg

end module adpio
```

C SUMMARY OF CODE CHANGES TO APPENDIX B FOR DNAD INTEGRATION

C.1 Changes to `adpsolver` Module

Original: lines 1-2

```
1    module adpsolver
2        implicit none
```

Modified: lines 1-6

```
1    module adpsolver
2    #ifdef dnad
3        use dnadmod
4    #define real type(dual)
5    #endif
6        implicit none
```

C.2 Changes to `adpio` Module

Original: lines 1-2

```
1    module adpio
2        implicit none
```

Modified: lines 1-6

```
1    module adpio
2    #ifdef dnad
3        use dnadmod
4    #define real type(dual)
5    #endif
6        use adpsolver
```

Original: lines 5-7

```
5        integer, parameter :: ioUnit = 10  ! IO unit for files
6
7    contains
```

Modified: lines 9-16

```
9        integer, parameter :: ioUnit = 10  ! IO unit for files
10   #ifdef dnad
11       integer, parameter :: maxdvs = 3  ! Maximum number of derivatives supported
12       integer :: dvcount = 0  ! Count of partial derivatives requested
13       character*80, dimension(maxdvs) :: dvnames = ""  ! List of derivative names
14   #endif
15
16   Contains
```

Original: lines 149-150

```
149                     err = setSolverField(model, paramName, paramValue)
150                 case default
```

Modified: lines 158-163

```
158                     err = setSolverField(model, paramName, paramValue)
159 #ifdef dnad
160                 case ("*dnad")
161                     err = setDNADField(model, paramName, paramValue)
162 #endif
163                 case default
```

Original: lines 339-342

```
339     end function setSolverField
340
341
342     real function stringToReal(str) result (val)
```

Modified: lines 352-414

```
352     end function setSolverField
353
354
355 #ifdef dnad
356     !!! Set a field from the dnad card
357     !!!
358     !!! Input:
359     !!!     model = adpmodel object
360     !!!     paramName = Name of field to set
361     !!!     paramValue = String representation of parameter value
362     !!!
363     !!! Return:
364     !!!     stat = Error status flag (0 = success,
365     !!!                                  1 = Unrecognized parameter name,
366     !!!                                  2 = Number of allowed dvs exceeded)
367     integer function setDNADField(model, paramName, paramValue) result(stat)
368         class(adpmodel), intent(inout) :: model
369         character(len=*), intent(in) :: paramName  ! Parameter name
370         character(len=*), intent(in) :: paramValue  ! Parameter value
371
372         ! Initialize the error flag
373         stat = 0
374
375         select case(paramName)
376             case ("dv")
377                 ! Check number of design variables
378                 if (dvcount .ge. ndv) then
379                     write(*, '(6X, A, I0, A, /, 6X, A, A)') &
380                         &   "The maximum number of design variables (", &
381                         &   ndv, ") has been reached.", paramValue, &
382                         &   " will not be included as a design variable."
383                 else
384                     select case(paramValue)
385                         case ("u")
386                             dvcount = dvcount + 1
387                             model%u%dx(dvcount) = 1.0
388                             dvnames(dvcount) = "dPhi/dU"
389                         case ("gamma")
390                             dvcount = dvcount + 1
391                             model%gamma%dx(dvcount) = 1.0
392                             dvnames(dvcount) = "dPhi/dGamma"
```

```
393                         case ("c")
394                             dvcount = dvcount + 1
395                             model%c%dx(dvcount) = 1.0
396                             dvnames(dvcount) = "dPhi/dC"
397                         case default
398                             write(*, '(6X, A, 1X, A)') &
399                                 &   "Unrecognized design variable:", &
400                                 &   paramValue
401                     end select
402                 end if
403             case default
404                 write(*, '(6X, A, 1X, A)') &
405                     &   "Unrecognized parameter name on dnad card:", &
406                     &    trim(paramName)
407                 stat = 1
408             end select
409
410     end function setDNADField
411 #endif
412
413
414     REAL function stringToReal(str) result (val)
```

Original: lines 405-408

```
405     end function writeOutput
406
407
408     subroutine writeData(model)
```

Modified: lines 477-481

```
477     end function writeOutput
478
479
480 #ifndef dnad
481     subroutine writeData(model)
```

Original: lines 435-438

```
435     end subroutine writeData
436
437
438     character*80 function removeWhitespace(str) result(mstr)
```

Modified: lines 508-546

```
508     end subroutine writeData
509 #else
510     subroutine writeData(model)
511         class(adpmodel), intent(inout) :: model   ! adpmodel object
512
513         integer :: i, j  ! Loop control variable
514         character*80 :: fmtString
515
516         if (model%method .eq. analytical .and. model%computeDerivatives) then
517             ! Header
518             write(fmtString, *) '(', 4 + dvcount, '(A, ","), A)'
```

```
            write(ioUnit, fmtString) "x", "phi", "dPhi/dU", "dPhi/dGamma", &
                &   "dPhi/dC", (trim(dvnames(j))//"_AD", j=1, dvcount)

            ! Results
            write(fmtString, *) '(', 4 + dvcount, '(ES23.15, ", "), ES23.15)'
            do i=1, model%npts
                write(ioUnit, fmtString) model%x(i)%x, model%phi(i)%x, &
                    &   model%dPhi_dU(i)%x, model%dPhi_dGamma(i)%x, &
                    &   model%dPhi_dC(i)%x, (model%phi(i)%dx(j), j=1, dvcount)
            end do
        else
            ! Header
            write(fmtString, *) '(', 1 + dvcount, '(A, ","), A)'
            write(ioUnit, fmtString) "x", "phi", &
                &   (trim(dvnames(j))//"_AD", j=1, dvcount)

            ! Results
            write(fmtString, *) '(', 1 + dvcount, '(ES23.15, ", "), ES23.15)'
            do i=1, model%npts
                write(ioUnit, fmtString) model%x(i)%x, model%phi(i)%x, &
                    &   (model%phi(i)%dx(j), j=1, dvcount)
            end do
        end if
    end subroutine writeData
#endif


    character*80 function removeWhitespace(str) result(mstr)
```

D SUMMARY OF CODE CHANGES TO MACHUP FOR DNAD INTEGRATION

D.1 Changes to `loads_m` Module

Original: lines 1-2

```
1    module loads_m
2        use plane_m
```

Modified: lines 1-6

```
1    module loads_m
2    #ifdef dnad
3        use dnadmod
4    #define real type(dual)
5    #endif
6        use plane_m
```

Original: lines 58-61

```
58      real :: ans(7),P(3),percent,span,chord
59      120 Format(A15, 100ES25.13)
60
61      !Get filename if specified
```

Modified: lines 62-68

```
62      real :: ans(7),P(3),percent,span,chord
63      real :: zero
64      120 Format(A15, 100ES25.13)
65
66      zero = 0.0
67
68      !Get filename if specified
```

D.2 Changes to `plane_m` Module

Original: lines 1-2

```
1    module plane_m
2        use myjson_m
```

Modified: lines 1-6

```
1    module plane_m
2    #ifdef dnad
3        use dnadmod
4    #define real type(dual)
5    #endif
6        use myjson_m
```

Original: lines 150-174

```
150     call t%json%get('plane.name', cval); call json_check(); t%name = trim(cval)
151     t%CG(1) = json_file_required_real(t%json,'plane.CGx')
152     t%CG(2) = json_file_required_real(t%json,'plane.CGy')
153     t%CG(3) = json_file_required_real(t%json,'plane.CGz')
154
155     t%Sr = json_file_required_real(t%json,'reference.area')
156     t%long_r = json_file_required_real(t%json,'reference.longitudinal_length')
157     t%lat_r = json_file_required_real(t%json,'reference.lateral_length')
158
159     t%alpha = json_file_required_real(t%json,'condition.alpha'); t%alpha = t%alpha*pi/180.0
160     t%beta = json_file_optional_real(t%json,'condition.beta',0.0); t%beta = t%beta*pi/180.0
161
162     t%omega(1) = json_file_optional_real(t%json,'condition.omega.roll',0.0)
163     t%omega(2) = json_file_optional_real(t%json,'condition.omega.pitch',0.0)
164     t%omega(3) = json_file_optional_real(t%json,'condition.omega.yaw',0.0)
165
166     t%hag = json_file_optional_real(t%json,'condition.ground',0.0)
167     if(t%hag.gt.0.0) t%groundplane = 1
168
169     call t%json%get('solver.type', cval); call json_check(); solver = trim(cval)
170     jacobian_converged = json_file_optional_real(t%json,'solver.convergence', 1.0e-6)
171     jacobian_omega = json_file_optional_real(t%json,'solver.relaxation',0.9)
172     nonlinear_maxiter = json_file_optional_integer(t%json,'solver.maxiter',100)
173
174     call t%json%get('airfoil_DB', cval); call json_check(); DB_Airfoil = trim(cval)
```

Modified: lines 154-178

```
154     call t%json%get('plane.name', cval); call json_check(); t%name = trim(cval)
155     call myjson_get(t%json,'plane.CGx', t%CG(1))
156     call myjson_get(t%json,'plane.CGy', t%CG(2))
157     call myjson_get(t%json,'plane.CGz', t%CG(3))
158
159     call myjson_get(t%json,'reference.area', t%Sr)
160     call myjson_get(t%json,'reference.longitudinal_length', t%long_r)
161     call myjson_get(t%json,'reference.lateral_length', t%lat_r)
162
163     call myjson_get(t%json,'condition.alpha',t%alpha); t%alpha=t%alpha*pi/180.0
164     call myjson_get(t%json,'condition.beta',t%beta,0.0); t%beta=t%beta*pi/180.0
165
166     call myjson_get(t%json,'condition.omega.roll', t%omega(1), 0.0)
167     call myjson_get(t%json,'condition.omega.pitch', t%omega(2), 0.0)
168     call myjson_get(t%json,'condition.omega.yaw', t%omega(3), 0.0)
169
170     call myjson_get(t%json,'condition.ground', t%hag, 0.0)
171     if(t%hag.gt.0.0) t%groundplane = 1
172
173     call t%json%get('solver.type', cval); call json_check(); solver = trim(cval)
174     call myjson_get(t%json,'solver.convergence', jacobian_converged, 1.0e-6)
175     call myjson_get(t%json,'solver.relaxation', jacobian_omega, 0.9)
176     call myjson_get(t%json,'solver.maxiter', nonlinear_maxiter, 100)
177
178     call t%json%get('airfoil_DB', cval); call json_check(); DB_Airfoil = trim(cval)
```

Original: lines 193-196

```
193             call json_check();
194             call t%json%get('controls.'//trim(j_cont%name)//'.deflection', t%controls(icontrol)%deflection);
195             call json_check()
196             t%controls(icontrol)%deflection = pi/180.0*t%controls(icontrol)%deflect
```

Modified: lines 197-199

```
197             call json_check();
198             call myjson_get(t%json, 'controls.'//trim(j_cont%name)//'.deflection', t%controls(icontrol)%deflection)
199             t%controls(icontrol)%deflection = pi/180.0*t%controls(icontrol)%deflect
```

Original: lines 232-251

```
232         call t%json%get('wings.'//trim(j_wing%name)//'.ID', t%wings(iwing)%ID); call json_check()
233         call t%json%get('wings.'//trim(j_wing%name)//'.side', cval); call json_check();
234         t%wings(iwing)%orig_side = trim(cval)
235         call t%json%get('wings.'//trim(j_wing%name)//'.connect.ID', t%wings(iwing)%connectid); call json_check()
236         call t%json%get('wings.'//trim(j_wing%name)//'.connect.location', cval); call json_check();
237         t%wings(iwing)%connectend = trim(cval)
238         call t%json%get('wings.'//trim(j_wing%name)//'.connect.dx', t%wings(iwing)%doffset(1)); call json_check()
239         call t%json%get('wings.'//trim(j_wing%name)//'.connect.dy', t%wings(iwing)%doffset(2)); call json_check()
240         call t%json%get('wings.'//trim(j_wing%name)//'.connect.dz', t%wings(iwing)%doffset(3)); call json_check()
241         call t%json%get('wings.'//trim(j_wing%name)//'.connect.yoffset', t%wings(iwing)%dy); call json_check()
242         call t%json%get('wings.'//trim(j_wing%name)//'.span', t%wings(iwing)%span); call json_check()
243         call t%json%get('wings.'//trim(j_wing%name)//'.sweep', sweep); call json_check()
244         call t%json%get('wings.'//trim(j_wing%name)//'.dihedral', dihedral); call json_check()
245         call t%json%get('wings.'//trim(j_wing%name)//'.mounting_angle', mount); call json_check()
246         call t%json%get('wings.'//trim(j_wing%name)//'.washout', washout); call json_check()
247         call t%json%get('wings.'//trim(j_wing%name)//'.root_chord', t%wings(iwing)%chord_1); call json_check()
248         call t%json%get('wings.'//trim(j_wing%name)//'.tip_chord', t%wings(iwing)%chord_2); call json_check()
249
250         call t%json%get('wings.'//trim(j_wing%name)//'.sweep_definition', t%wings(iwing)%sweep_definition);
251         if(json_failed()) t%wings(iwing)%sweep_definition=1
```

Modified: lines 235-254

```
235         call t%json%get('wings.'//trim(j_wing%name)//'.ID', t%wings(iwing)%ID); call json_check()
236         call t%json%get('wings.'//trim(j_wing%name)//'.side', cval); call json_check();
237         t%wings(iwing)%orig_side = trim(cval)
238         call t%json%get('wings.'//trim(j_wing%name)//'.connect.ID', t%wings(iwing)%connectid); call json_check()
239         call t%json%get('wings.'//trim(j_wing%name)//'.connect.location', cval); call json_check();
240         t%wings(iwing)%connectend = trim(cval)
241         call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.connect.dx', t%wings(iwing)%doffset(1))
242         call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.connect.dy', t%wings(iwing)%doffset(2))
243         call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.connect.dz', t%wings(iwing)%doffset(3))
244         call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.connect.yoffset', t%wings(iwing)%dy)
245         call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.span', t%wings(iwing)%span)
246         call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.sweep', sweep)
247         call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.dihedral', dihedral)
248         call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.mounting_angle', mount)
249         call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.washout',washout)
250         call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.root_chord', t%wings(iwing)%chord_1)
251         call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.tip_chord', t%wings(iwing)%chord_2)
252
253         call t%json%get('wings.'//trim(j_wing%name)//'.sweep_definition', t%wings(iwing)%sweep_definition);
254         if(json_failed()) t%wings(iwing)%sweep_definition=1
```

Original: lines 286-303

```
286         !control surface defs
287         call t%json%get('wings.'//trim(j_wing%name)//'.control.span_root', t%wings(iwing)%control_span_root);
288         if(json_failed()) then
289             call json_clear_exceptions()
290             t%wings(iwing)%has_control_surface = 0
291         else
292             t%wings(iwing)%has_control_surface = 1
293             call t%json%get('wings.'//trim(j_wing%name)//'.control.span_root', t%wings(iwing)%control_span_root);
294             call json_check()
295             call t%json%get('wings.'//trim(j_wing%name)//'.control.span_tip', t%wings(iwing)%control_span_tip );
296             call json_check()
```

```
297             call t%json%get('wings.'//trim(j_wing%name)//'.control.chord_root', t%wings(iwing)%control_chord_root);
298             call json_check()
299             call t%json%get('wings.'//trim(j_wing%name)//'.control.chord_tip', t%wings(iwing)%control_chord_tip );
300             call json_check()
301             call t%json%get('wings.'//trim(j_wing%name)//'.control.is_sealed', t%wings(iwing)%control_is_sealed);
302             call json_check()
303         end if
```

Modified: lines 289-302

```
289         !control surface defs
290         call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.control.span_root', t%wings(iwing)%control_span_root, -1.0)
291
292         if(t%wings(iwing)%control_span_root < 0.0) then
293             call json_clear_exceptions()
294             t%wings(iwing)%has_control_surface = 0
295         else
296             t%wings(iwing)%has_control_surface = 1
297             call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.control.span_tip', t%wings(iwing)%control_span_tip)
298             call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.control.chord_root', t%wings(iwing)%control_chord_root)
299             call myjson_get(t%json, 'wings.'//trim(j_wing%name)//'.control.chord_tip', t%wings(iwing)%control_chord_tip)
300             call t%json%get('wings.'//trim(j_wing%name)//'.control.is_sealed', t%wings(iwing)%control_is_sealed);
301             call json_check()
302         end if
```

Original: lines 377-384

```
377         case ('linear')
378             airfoils(i)%aL0 = json_file_required_real(f_json,trim(prefix)//'properties.alpha_L0');
379             airfoils(i)%CLa = json_file_required_real(f_json,trim(prefix)//'properties.CL_alpha');
380             airfoils(i)%CmL0 = json_file_required_real(f_json,trim(prefix)//'properties.Cm_L0');
381             airfoils(i)%Cma = json_file_required_real(f_json,trim(prefix)//'properties.Cm_alpha');
382             airfoils(i)%CD0 = json_file_required_real(f_json,trim(prefix)//'properties.CD_min');
383             airfoils(i)%CLmax = json_file_optional_real(f_json,trim(prefix)//'properties.CL_max',-1.0);
384             airfoils(i)%has_data_file = 0
```

Modified: lines 376-383

```
376         case ('linear')
377             call myjson_get(f_json, trim(prefix)//'properties.alpha_L0', airfoils(i)%aL0);
378             call myjson_get(f_json, trim(prefix)//'properties.CL_alpha', airfoils(i)%CLa);
379             call myjson_get(f_json, trim(prefix)//'properties.Cm_L0', airfoils(i)%CmL0);
```

```
380             call myjson_get(f_json, trim(prefix)//'properties.Cm_alpha',
    airfoils(i)%Cma);
381             call myjson_get(f_json, trim(prefix)//'properties.CD_min',
    airfoils(i)%CD0);
382             call myjson_get(f_json, trim(prefix)//'properties.CL_max',
    airfoils(i)%CLmax, -1.0);
383             airfoils(i)%has_data_file = 0
```

Original: lines 604-607

```
604                     if(trim(j_mix%name).eq.trim(t%controls(icontrol)%name)) then
605                         call t%json%get('wings.'//trim(t%wings(iwing)%name)
    //'.control.mix.'//trim(j_mix%name), ratio);
606                         call json_check()
607                         if(t%controls(icontrol)%is_symmetric.eq.1) then
```

Modified: lines 603-605

```
603                     if(trim(j_mix%name).eq.trim(t%controls(icontrol)%name)) then
604                         call myjson_get(t%json, 'wings.'// trim(t%wings(iwing)%
    name) //'.control.mix.'//trim(j_mix%name), ratio)
605                         if(t%controls(icontrol)%is_symmetric.eq.1) then
```

Original: lines 720-722

```
720     integer :: i
721     real :: time1,time2
722     call cpu_time(time1)
```

Modified: lines 718-720

```
718     integer :: i
719     REAL :: time1,time2
720     call cpu_time(time1)
```

Original: lines 1133-1136

```
1133    real :: A(3,3),Atemp(3,3),X(3),B(3),span,span1,span2,span3,F0(3),F1(3),P(3),
    percent
1134    120 Format(A15, 100ES25.13)
1135
1136    filename = trim(adjustl(t%master_filename))//'_loads.txt'
```

Modified: lines 1131-1137

```
1131    real :: A(3,3),Atemp(3,3),X(3),B(3),span,span1,span2,span3,F0(3),F1(3),P(3),
    percent
1132    real :: zero
1133    120 Format(A15, 100ES25.13)
1134
1135    zero = 0.0
1136
1137    filename = trim(adjustl(t%master_filename))//'_loads.txt'
```

Original: lines 1153-1155

```
1153    call ds_create_from_data(dist,30,2,rawdata)
1154    call ds_cubic_setup(dist,1,2,0.0,2,0.0)
1155    call ds_print_data(dist)
```

Modified: lines 1154-1156

```
1153    call ds_create_from_data(dist,30,2,rawdata)
1154    call ds_cubic_setup(dist, 1, 2, zero, 2, zero)
1155    call ds_print_data(dist)
```

D.3 Changes to `special_functions_m` Module

Original: lines 1-2

```
1   module special_functions_m
2       use plane_m
```

Modified: lines 1-6

```
1   module special_functions_m
2   #ifdef dnad
3       use dnadmod
4   #define real type(dual)
5   #endif
6       use plane_m
```

Original: lines 459-460

```
459     start_alpha = json_optional_real(json_command,'start_alpha',0.0)
460     start_alpha = (start_alpha+1.0)*pi/180.0
```

Modified: lines 463-464

```
463     call myjson_get(json_command, 'start_alpha', start_alpha, 0.0)
464     start_alpha = (start_alpha+1.0)*pi/180.0
```

Original: lines 469-471

```
469     do ialpha = -10, 300, 1
470         t%alpha = real(ialpha)/10.0*pi/180.0 + start_alpha
471         call plane_run_current(t)
```

Modified: lines 473-475

```
473     do ialpha = -10, 300, 1
474         t%alpha = REAL(ialpha)/10.0*pi/180.0 + start_alpha
475         call plane_run_current(t)
```

Original: lines 581-598

```
581      write(*,*) 'Using control surface: ',trim(controlname)
582
583      call json_get(json_command,'CL',CL_target,json_found)
584
585      if(json_failed()) then
586          call json_clear_exceptions()
587          trimType = 2
588
589          CW_target = json_required_real(json_command,'CW');
590          climb=json_optional_real(json_command,'climb',0.0); climb=climb*pi/180.0
591          thrust_x = json_optional_real(json_command,'thrust.x',0.0);
592          thrust_z = json_optional_real(json_command,'thrust.z',0.0);
593          thrust_a = json_optional_real(json_command,'thrust.angle',0.0);
594
595      else !trim using CL, Cm
596          trimType = 1
597          Cm_target = json_optional_real(json_command,'Cm',0.0);
598      end if
```

Modified: lines 585-602

```
585      write(*,*) 'Using control surface: ',trim(controlname)
586
587      call myjson_get(json_command, 'CL', CL_target, -1.0)
588
589      if(CL_target < 0.0) then
590          call json_clear_exceptions()
591          trimType = 2
592
593          call myjson_get(json_command, 'CW', CW_target)
594          call myjson_get(json_command,'climb',climb,0.0); climb=climb*pi/180.0
595          call myjson_get(json_command, 'thrust.x', thrust_x, 0.0);
596          call myjson_get(json_command, 'thrust.z', thrust_z, 0.0);
597          call myjson_get(json_command, 'thrust.angle', thrust_a, 0.0);
598
599      else !trim using CL, Cm
600          trimType = 1
601          call myjson_get(json_command, 'Cm', Cm_target, 0.0);
602      end if
```

Original: lines 606-611

```
606      de = t%controls(icontrol)%deflection !radians
607
608      delta  = json_optional_real(json_command,'delta',0.5);
609      maxres = json_optional_real(json_command,'convergence',1.0e-10);
610      relaxation = json_optional_real(json_command,'relaxation',1.0);
611      maxiter = json_optional_integer(json_command,'maxiter',50);
```

Modified: lines 610-615

```
610      de = t%controls(icontrol)%deflection !radians
611
612      call myjson_get(json_command, 'delta', delta, 0.5);
613      call myjson_get(json_command, 'convergence', maxres, 1.0e-10);
614      call myjson_get(json_command, 'relaxation', relaxation, 1.0);
615      call myjson_get(json_command, 'maxiter', maxiter, 50);
```

Original: lines 755-769

```
755     write(*,*) '---------- Finding alpha to target CL -----------'
756
757     CL_target = json_required_real(json_command,'CL');
758
759     !store alpha and de in case no solution is found
760     alpha_temp = t%alpha !radians
761
762     alpha = 0.0 !t%alpha !radians
763
764     delta  = json_optional_real(json_command,'delta',0.5);
765     maxres = json_optional_real(json_command,'convergence',1.0e-10);
766     relaxation = json_optional_real(json_command,'relaxation',1.0);
767     maxiter = json_optional_integer(json_command,'maxiter',50);
768
769     write(*,*)
```

Modified: lines 759-773

```
759     write(*,*) '---------- Finding alpha to target CL -----------'
760
761     call myjson_get(json_command, 'CL', CL_target);
762
763     !store alpha and de in case no solution is found
764     alpha_temp = t%alpha !radians
765
766     alpha = 0.0 !t%alpha !radians
767
768     call myjson_get(json_command, 'delta', delta, 0.5);
769     call myjson_get(json_command, 'convergence', maxres, 1.0e-10);
770     call myjson_get(json_command, 'relaxation', relaxation, 1.0);
771     call myjson_get(json_command, 'maxiter', maxiter, 50);
772
773     write(*,*)
```

Original: lines 843-855

```
843     type(plane_t) :: t
844     character(len=:),allocatable :: cval
845     character(100) :: target_var, change_var
846     real :: result,x0,x1,xnew,f0,f1, target_val, tolerance
847     integer :: ios
848
849     !Read json target info
850     call t%json%get('run.target.variable',        cval);   call json_check();
    target_var = trim(cval)
851     call t%json%get('run.target.value',      target_val);   call json_check()
852     call t%json%get('run.target.tolerance',   tolerance);   call json_check()
853     call t%json%get('run.target.change',          cval);   call json_check();
    change_var = trim(cval)
854
855     write(*,*) '     target variable = ',trim(target_var)
```

Modified: lines 847-857

```
847     type(plane_t) :: t
848     character(len=:), allocatable :: target_var, change_var
849     real :: result,x0,x1,xnew,f0,f1, target_val, tolerance
```

```
850
851      !Read json target info
852      call myjson_get(t%json, 'run.target.variable', target_var)
853      call myjson_get(t%json, 'run.target.value', target_val)
854      call myjson_get(t%json, 'run.target.tolerance', tolerance)
855      call myjson_get(t%json, 'run.target.change', change_var)
856
857      write(*,*) '    target variable = ',trim(target_var)
```

Original: lines 922-927

```
922          call json_get(json_command,trim(c_var%name)//'.file', cval,json_found); call json_check(); var_file = trim(cval)
923          save_file = json_optional_integer(json_command,trim(c_var%name)//'.save', 0)
924
925          call f_json%load_file(filename = var_file);       call json_check()
926          call f_json%get(trim(var_name),   var_value);     call json_check()
927          call json_value_add(p_root, trim(c_var%name), var_value)
```

Modified: lines 924-929

```
924          call json_get(json_command,trim(c_var%name)//'.file', cval,json_found); call json_check(); var_file = trim(cval)
925          call myjson_get(json_command,trim(c_var%name)//'.save', save_file, 0);
926
927          call f_json%load_file(filename = var_file);       call json_check()
928          call myjson_get(f_json, trim(var_name), var_value)
929          call json_value_add(p_root, trim(c_var%name), var_value)
```

Original: lines 961-964

```
961      call f_json%load_file(filename = fn);                    call json_check()
962      call f_json%get(trim(fitness_var),   fitness_val);      call json_check()
963
964      open(unit = 10, File = 'fitness.txt', action = 'write', iostat = ios)
```

Modified: lines 963-966

```
963      call f_json%load_file(filename = fn);                    call json_check()
964      call myjson_get(f_json, trim(fitness_var), fitness_val)
965
966      open(unit = 10, File = 'fitness.txt', action = 'write', iostat = ios)
```

Original: lines 984-989

```
984      call f_json%load_file(filename = fn);                    call json_check()
985      call f_json%get('total.MyAirplane.CD',   CD);      call json_check()
986      call f_json%get('total.Wing_1_left.Cl',   Cl);      call json_check()
987      call f_json%get('total.Wing_1_left.Cn',   Cn);      call json_check()
988
989      if(opt_type.eq.1) then
```

Modified: lines 986-991

```
986     call f_json%load_file(filename = fn);              call json_check()
987     call myjson_get(f_json, 'total.MyAirplane.CD', CD)
988     call myjson_get(f_json, 'total.MyAirplane.Cl', Cl)
989     call myjson_get(f_json, 'total.MyAirplane.Cn', Cn)
990
991     if(opt_type.eq.1) then
```

Original: lines 1028-1031

```
1028    call f_json%load_file(filename = fn);        call json_check()
1029    call f_json%get(trim(read_var),    result);    call json_check()
1030
1031 end subroutine sf_run_single
```

Modified: lines 1030-1033

```
1028    call f_json%load_file(filename = fn);        call json_check()
1029    call myjson_get(f_json, trim(read_var), result)
1030
1031 end subroutine sf_run_single
```

D.4 Changes to view_m Module

Original: lines 1-2

```
1    module view_m
2        use plane_m
```

Modified: lines 1-6

```
1    module view_m
2    #ifdef dnad
3        use dnadmod
4    #define real type(dual)
5    #endif
6        use plane_m
```

Original: line 124

```
124           delta = gpsize/real(gnum)
```

Modified: line 128

```
128           delta = gpsize/REAL(gnum)
```

Original: lines 132-133

```
132            P1(1) = P0(1) - gpsize*cos(t%alpha); P1(2) = P0(2) + real(i)*delta; P1(3) = P0(3) - gpsize*sin(t%alpha)
133            P2(1) = P0(1) + gpsize*cos(t%alpha); P2(2) = P0(2) + real(i)*delta; P2(3) = P0(3) + gpsize*sin(t%alpha)
```

Modified: lines 136-137

```
136            P1(1) = P0(1) - gpsize*cos(t%alpha); P1(2) = P0(2) + REAL(i)*delta; P1(3) = P0(3) - gpsize*sin(t%alpha)
137            P2(1) = P0(1) + gpsize*cos(t%alpha); P2(2) = P0(2) + REAL(i)*delta; P2(3) = P0(3) + gpsize*sin(t%alpha)
```

Original: lines 140-141

```
140            P1(1) = P0(1) - real(i)*delta*cos(t%alpha); P1(2) = P0(2) + gpsize; P1(3) = P0(3) - real(i)*delta*sin(t%alpha)
141            P2(1) = P0(1) - real(i)*delta*cos(t%alpha); P2(2) = P0(2) - gpsize; P2(3) = P0(3) - real(i)*delta*sin(t%alpha)
```

Modified: lines 144-145

```
144            P1(1) = P0(1) - REAL(i)*delta*cos(t%alpha); P1(2) = P0(2) + gpsize; P1(3) = P0(3) - REAL(i)*delta*sin(t%alpha)
145            P2(1) = P0(1) - REAL(i)*delta*cos(t%alpha); P2(2) = P0(2) - gpsize; P2(3) = P0(3) - REAL(i)*delta*sin(t%alpha)
```

D.5 Changes to `wing_m` Module

Original: lines 1-2

```
1  module wing_m
2      use section_m
```

Modified: lines 1-6

```
1  module wing_m
2  #ifdef dnad
3      use dnadmod
4  #define real type(dual)
5  #endif
6      use section_m
```

Original: lines 71-73

```
71     integer :: isec
72     real :: start(3),qvec(3),nvec(3),avec(3),fvec(3),dtheta,percent_1,percent_2,percent_c,chord_1,chord_2,RA,span
73     real :: my_sweep,my_dihedral,my_twist,temp
```

Modified: lines 75-78

```
75     integer :: isec
76     REAL :: dtheta
```

```
77      real :: start(3), qvec(3), nvec(3), avec(3), fvec(3), percent_1, percent_2, percent_c, chord_1, chord_2, RA, span
78      real :: my_sweep, my_dihedral, my_twist, temp
```

Original: lines 104-112

```
104     call wing_allocate(t)
105     dtheta = pi/real(t%nSec)
106     t%area = 0.0
107     span = 0.0
108     do isec=1,t%nSec
109         percent_1 = 0.5*(1.0-cos(dtheta*real(isec-1)))
110         percent_2 = 0.5*(1.0-cos(dtheta*real(isec)))
111         percent_c = 0.5*(1.0-cos(dtheta*(real(isec)-0.5)))
112         if(t%side.eq.'left') then !must handle differently for left wing
```

Modified: lines 109-117

```
109     call wing_allocate(t)
110     dtheta = pi/REAL(t%nSec)
111     t%area = 0.0
112     span = 0.0
113     do isec=1,t%nSec
114         percent_1 = 0.5*(1.0-cos(dtheta*REAL(isec-1)))
115         percent_2 = 0.5*(1.0-cos(dtheta*REAL(isec)))
116         percent_c = 0.5*(1.0-cos(dtheta*(REAL(isec)-0.5)))
117         if(t%side.eq.'left') then !must handle differently for left wing
```

Original: lines 274-276

```
274     integer :: isec
275     real :: A,B,C,D,P1(3),P2(3),P3(3),tempv(3),tempr,temp_percent
276     P1(1) = CG(1) - hag*sin(alpha) !offset from CG
```

Modified: lines 279-282

```
279     integer :: isec
280     real :: A,B,C,D,P1(3),P2(3),P3(3),tempv(3),tempr,temp_percent,zero
281     zero = 0.0
282     P1(1) = CG(1) - hag*sin(alpha) !offset from CG
```

Original: lines 295-302

```
295         t%sec(isec)%chord_2 = tempr
296
297         call math_reflect_point(A,B,C,0.0,t%sec(isec)%un,tempv)
298         t%sec(isec)%un = tempv
299         call math_reflect_point(A,B,C,0.0,t%sec(isec)%ua,tempv)
300         t%sec(isec)%ua = tempv
301         call math_reflect_point(A,B,C,0.0,t%sec(isec)%uf,tempv)
302         t%sec(isec)%uf = tempv
```

Modified: lines 301-308

```
301         t%sec(isec)%chord_2 = tempr
302
303         call math_reflect_point(A,B,C,zero,t%sec(isec)%un,tempv)
304         t%sec(isec)%un = tempv
305         call math_reflect_point(A,B,C,zero,t%sec(isec)%ua,tempv)
306         t%sec(isec)%ua = tempv
307         call math_reflect_point(A,B,C,zero,t%sec(isec)%uf,tempv)
308         t%sec(isec)%uf = tempv
```

D.6 Changes to `airfoil_m` Module

Original: lines 1-2

```
1    module airfoil_m
2        use dataset_m
```

Modified: lines 1-6

```
1    module airfoil_m
2    #ifdef dnad
3        use dnadmod
4    #define real type(dual)
5    #endif
6        use dataset_m
```

D.7 Changes to `atmosphere_m` Module

Original: lines 1-2

```
1    module atmosphere_m
2        use dataset_m
```

Modified: lines 1-6

```
1    module atmosphere_m
2    #ifdef dnad
3        use dnadmod
4    #define real type(dual)
5    #endif
6        use dataset_m
```

Original: line 12

```
12      real :: temp(36,7)
```

Modified: lines 17-20

```
17      real :: temp(36,7)
18      real :: zero
19
20      zero = 0.0
```

Original: lines 52-55

```
52      call ds_create_from_data(t%properties,36,7,temp(:,:))
53      call ds_cubic_setup(t%properties,1,2,0.0,2,0.0)
54
55  end subroutine atm_create
```

Modified: lines 60-63

```
60      call ds_create_from_data(t%properties,36,7,temp(:,:))
61      call ds_cubic_setup(t%properties,1,2,zero,2,zero)
62
63  end subroutine atm_create
```

D.8 Changes to `dataset_m` Module

Original: lines 1-2

```
1   module dataset_m
2       use math_m
```

Modified: lines 1-6

```
1   module dataset_m
2   #ifdef dnad
3       use dnadmod
4   #define real type(dual)
5   #endif
6       use math_m
```

D.9 Changes to `math_m` Module

Original: lines 1-4

```
1   module math_m
2       implicit none
3       real, parameter :: pi = 3.1415926535897932
4   contains
```

Modified: lines 1-9

```
1   module math_m
2   #ifdef dnad
3       use dnadmod
4   #define real type(dual)
5   #endif
6
7       implicit none
8       REAL, parameter :: pi = 3.1415926535897932
9   contains
```

Original: lines 225-229

```
225     INTEGER::n,D,info
226     REAL,DIMENSION(n)::B,X
227     REAL,DIMENSION(n,n)::A
```

```
228
229      INTEGER,allocatable,DIMENSION(:) :: INDX
```

Modified: lines 230-234

```
230      INTEGER::n,D,info
231      real,DIMENSION(n)::B,X
232      real,DIMENSION(n,n)::A
233
234      INTEGER,allocatable,DIMENSION(:) :: INDX
```

Original: lines 274-277

```
274      REAL, PARAMETER :: TINY=1.5D-16
275      REAL  AMAX,DUM, SUM, A(N,N)
276      REAL,ALLOCATABLE,DIMENSION(:) :: VV
277      INTEGER N, CODE, D, INDX(N)
```

Modified: lines 279-282

```
279      REAL, PARAMETER :: TINY=1.5D-16
280      real  AMAX,DUM, SUM, A(N,N)
281      real,ALLOCATABLE,DIMENSION(:) :: VV
282      INTEGER N, CODE, D, INDX(N)
```

Original: lines 356-357

```
356      integer :: N
357      REAL  SUM, A(N,N),B(N)
358      INTEGER INDX(N)
```

Modified: lines 361-363

```
361      integer :: N
362      real  SUM, A(N,N),B(N)
363      INTEGER INDX(N)
```

D.10 Changes to `section_m` Module

Original: lines 1-2

```
1    module section_m
2        use airfoil_m
```

Modified: lines 1-6

```
1    module section_m
2    #ifdef dnad
3        use dnadmod
4    #define real type(dual)
5    #endif
6        use airfoil_m
```

D.11 Changes to `myjson_m` Module

Original: lines 1-134

```
1   module myjson_m
2       use json_m
3       implicit none
4
5       logical :: json_found
6
7   contains
8
9   !-------------------------------------------------------------------------------
    -
10  real function json_required_real(json,name)
11      implicit none
12      type(json_value),intent(in),pointer :: json
13      character(len=*) :: name
14      real :: value
15
16      call json_get(json, name, value, json_found)
17      if(json_failed()) then
18          write(*,*) 'Error: Unable to read required value: ',name
19          STOP
20      end if
21
22      json_required_real = value
23  end function json_required_real
24
25  !-------------------------------------------------------------------------------
    -
26  real function json_optional_real(json,name,default_value)
27      implicit none
28      type(json_value),intent(in),pointer :: json
29      character(len=*) :: name
30      real :: value, default_value
31
32      call json_get(json, name, value, json_found)
33      if((.not.json_found) .or. json_failed()) then
34          write(*,*) trim(name),' set to ',default_value
35          value = default_value
36          call json_clear_exceptions()
37      end if
38
39      json_optional_real = value
40  end function json_optional_real
41
42  !-------------------------------------------------------------------------------
    -
43  integer function json_optional_integer(json,name,default_value)
44      implicit none
45      type(json_value),intent(in),pointer :: json
46      character(len=*) :: name
47      integer :: value, default_value
48
49      call json_get(json, name, value, json_found)
50      if((.not.json_found) .or. json_failed()) then
51          write(*,*) trim(name),' set to ',default_value
52          value = default_value
53          call json_clear_exceptions()
54      end if
```

```

    json_optional_integer = value
end function json_optional_integer

!----------------------------------------------------------------------------
-
real function json_file_required_real(json,name)
    implicit none
    type(json_file) :: json
    character(len=*) :: name
    real :: value

    call json%get(name, value)
    if(json_failed()) then
        write(*,*) 'Error: Unable to read required value: ',name
        STOP
    end if

    json_file_required_real = value
end function json_file_required_real

!----------------------------------------------------------------------------
-
real function json_file_optional_real(json,name,default_value)
    implicit none
    type(json_file) :: json
    character(len=*) :: name
    real :: value, default_value

    call json%get(name, value)
    if(json_failed()) then
        write(*,*) name,' set to ',default_value
        value = default_value
        call json_clear_exceptions()
    end if

    json_file_optional_real = value
end function json_file_optional_real

!----------------------------------------------------------------------------
-
integer function json_file_optional_integer(json,name,default_value)
    implicit none
    type(json_file) :: json
    character(len=*) :: name
    integer :: value, default_value

    call json%get(name, value)
    if(json_failed()) then
        write(*,*) trim(name),' set to ',default_value
        value = default_value
        call json_clear_exceptions()
    end if

    json_file_optional_integer = value
end function json_file_optional_integer

!----------------------------------------------------------------------------
-
subroutine json_check()
    if(json_failed()) then
```

```
        call print_json_error_message()
        STOP
    end if
end subroutine json_check
!--------------------------------------------------------------------------------
subroutine print_json_error_message()
    implicit none
    character(len=:),allocatable :: error_msg
    logical :: status_ok

    !get error message:
    call json_check_for_errors(status_ok, error_msg)

    !print it if there is one:
    if (.not. status_ok) then
        write(*,'(A)') error_msg
        deallocate(error_msg)
        call json_clear_exceptions()
    end if

end subroutine print_json_error_message

end module myjson_m
```

Modified: lines 1-305

```
1    module myjson_m
2    #ifdef dnad
3        use dnadmod
4    #ifndef ndv
5    #define ndv 1
6    #endif
7    #endif
8        use json_m
9        implicit none
10
11       logical :: json_found
12   #ifdef dnad
13       integer, save :: n_design_vars = 0
14   #endif
15
16       interface myjson_get
17           module procedure :: myjson_value_get_real, myjson_file_get_real
18   #ifdef dnad
19           module procedure :: myjson_value_get_dual, myjson_file_get_dual
20   #endif
21           module procedure :: myjson_value_get_integer, myjson_file_get_integer
22           module procedure :: myjson_file_get_string
23       end interface myjson_get
24
25   #ifdef dnad
26       interface json_value_add
27           module procedure :: myjson_value_add_dual, myjson_value_add_dual_vec
28       end interface
29   #endif
30
31   contains
32
33   !---------------------------------------------------------------------------
34   subroutine myjson_value_get_real(json, name, value, default_value)
```

```
    implicit none
    type(json_value),intent(in),pointer :: json
    character(len=*), intent(in) :: name
    real, intent(out) :: value
    real, intent(in), optional :: default_value

    call json_get(json, name, value, json_found)
    if(json_failed() .or. (.not. json_found)) then
        if (present(default_value)) then
            write(*,*) trim(name),' set to ',default_value
            value = default_value
            call json_clear_exceptions()
        else
            write(*,*) 'Error: Unable to read required value: ',name
            STOP
        end if
    end if
end subroutine myjson_value_get_real

#ifdef dnad
!----------------------------------------------------------------------------
subroutine myjson_value_get_dual(json, name, value, default_value)
    implicit none
    type(json_value),intent(in),pointer :: json
    character(len=*), intent(in) :: name
    type(dual), intent(out) :: value
    real, intent(in), optional :: default_value

    real, dimension(:), allocatable :: vec

    call json_get(json, name, value%x, json_found)
    if(json_failed() .or. (.not. json_found)) then
        call json_clear_exceptions()
        call json_get(json, name, vec, json_found)
        if(json_found .and. (.not. json_failed())) then
            value = vec(1)  ! This will initialize derivatives to zero
            if(vec(2) /= 0) then
                if(n_design_vars < ndv) then
                    n_design_vars = n_design_vars + 1
                    value%dx(n_design_vars) = vec(2)
                else
                    write(*,*) 'Error: The number of design variables ', &
                        &   'exceeds the compiled limit: ', ndv
                    write(*,*) '       Reduce the number of design ', &
                        &   'variables, or increase the limit by'
                    write(*,*) '       specifying -Dndv=<num> when compiling.'
                    STOP
                end if
            end if
        else
            if (present(default_value)) then
                value = default_value
                write(*,*) trim(name),' set to ', value
                call json_clear_exceptions()
            else
                write(*,*) 'Error: Unable to read required value: ',name
                STOP
            end if
        end if
    end if

```

```
end subroutine myjson_value_get_dual

#endif
!----------------------------------------------------------------------
subroutine myjson_value_get_integer(json, name, value, default_value)
    implicit none
    type(json_value),intent(in),pointer :: json
    character(len=*), intent(in) :: name
    integer, intent(out) :: value
    integer, intent(in), optional :: default_value

    call json_get(json, name, value, json_found)
    if((.not.json_found) .or. json_failed()) then
        if (present(default_value)) then
            write(*,*) trim(name),' set to ',default_value
            value = default_value
            call json_clear_exceptions()
        else
            write(*,*) 'Error: Unable to read required value: ', name
            STOP
        end if
    end if

end subroutine myjson_value_get_integer

!----------------------------------------------------------------------
subroutine myjson_file_get_real(json, name, value, default_value)
    implicit none
    type(json_file) :: json
    character(len=*), intent(in) :: name
    real, intent(out) :: value
    real, intent(in), optional :: default_value

    call json%get(name, value)
    if(json_failed()) then
        if (present(default_value)) then
            write(*,*) trim(name), ' set to ', default_value
            value = default_value
            call json_clear_exceptions()
        else
            write(*,*) 'Error: Unable to read required value: ', trim(name)
            STOP
        end if
    end if

end subroutine myjson_file_get_real

#ifdef dnad
!----------------------------------------------------------------------
subroutine myjson_file_get_dual(json, name, value, default_value)
    implicit none
    type(json_file) :: json
    character(len=*), intent(in) :: name
    type(dual), intent(out) :: value
    real, intent(in), optional :: default_value

    real :: temp
    real, dimension(:), allocatable :: vec

    call json%get(name, temp)
    if(.not. json_failed()) then
```

```
        value = temp  ! Derivatives not specified, initialize to zero
    else
        call json_clear_exceptions()
        call json%get(name, vec)
        if(.not. json_failed()) then
            value = vec(1)  ! This will initialize derivatives to zero
            if(vec(2) /= 0) then
                if(n_design_vars < ndv) then
                    n_design_vars = n_design_vars + 1
                    value%dx(n_design_vars) = vec(2)
                else
                    write(*,*) 'Error: The number of design variables ', &
                        &   'exceeds the compiled limit: ', ndv
                    write(*,*) '       Reduce the number of design ', &
                        &   'variables, or increase the limit by'
                    write(*,*) '       specifying -Dndv=<num> when compiling.'
                    STOP
                end if
            end if
        else
            if (present(default_value)) then
                value = default_value
                write(*,*) trim(name),' set to ', value
                call json_clear_exceptions()
            else
                write(*,*) 'Error: Unable to read required value: ', trim(name)
                STOP
            end if
        end if
    end if

end subroutine myjson_file_get_dual

#endif
!-----------------------------------------------------------------------------
subroutine myjson_file_get_integer(json, name, value, default_value)
    implicit none
    type(json_file) :: json
    character(len=*), intent(in) :: name
    integer, intent(out) :: value
    integer, intent(in), optional :: default_value

    call json%get(name, value)
    if(json_failed()) then
        if (present(default_value)) then
            write(*,*) trim(name),' set to ',default_value
            value = default_value
            call json_clear_exceptions()
        else
            write(*,*) 'Error: Unable to read required value: ',name
            STOP
        end if
    end if
end subroutine myjson_file_get_integer

!-----------------------------------------------------------------------------
subroutine myjson_file_get_string(json, name, value)
    implicit none
    type(json_file) :: json
    character(len=*), intent(in) :: name
    character(:), allocatable, intent(out) :: value
```

```
    call json%get(name, value)
    if(json_failed()) then
        write(*,*) 'Error: Unable to read required value: ',name
        STOP
    end if

    value = trim(value)
end subroutine myjson_file_get_string

!---------------------------------------------------------------------------
subroutine json_check()
    if(json_failed()) then
        call print_json_error_message()
        STOP
    end if
end subroutine json_check
!---------------------------------------------------------------------------
subroutine print_json_error_message()
    implicit none
    character(len=:),allocatable :: error_msg
    logical :: status_ok

    !get error message:
    call json_check_for_errors(status_ok, error_msg)

    !print it if there is one:
    if (.not. status_ok) then
        write(*,'(A)') error_msg
        deallocate(error_msg)
        call json_clear_exceptions()
    end if

end subroutine print_json_error_message

#ifdef dnad
subroutine myjson_value_add_dual(me, name, val)

    implicit none

    type(json_value), pointer   :: me
    character(len=*),intent(in) :: name
    type(dual),intent(in)       :: val

    real,allocatable,dimension(:) :: vec
    integer :: vec_length

    ! Dual numbers are written to json as a vector: [u, du/dx, du/dy, ...]
    vec_length = size(val%dx) + 1
    allocate(vec(vec_length))
    vec(1) = val%x
    vec(2:) = val%dx

    call json_value_add(me, name, vec)

end subroutine myjson_value_add_dual

subroutine myjson_value_add_dual_vec(me, name, val)

    implicit none

```

```
    type(json_value), pointer          :: me
    character(len=*),intent(in)        :: name
    type(dual),dimension(:),intent(in)  :: val

    type(json_value),pointer :: var
    integer :: i

    !create the variable as an array:
    call json_value_create(var)
    call to_array(var,name)

    !populate the array:
    do i=1,size(val)
        call json_value_add(var, '', val(i))
    end do

    !add it:
    call json_value_add(me, var)

    !cleanup:
    nullify(var)

end subroutine myjson_value_add_dual_vec
!*****************************************************************************

#endif
end module myjson_m
```

E EXAMPLE MACHUP INPUT FILES

E.1 Example JSON-Formatted Input File

```
{
    "run": {
        "targetcl": {
            "CL": 0.5, "delta": 0.1, "relaxation": 1.0, "maxiter": 100,
            "convergence": 1.0E-12, "run": 1
        },
        "forces": {"run": 1},
        "distributions": {"run": 1},
        "stl": {"run": 1}
    },
    "solver": { "type": "nonlinear", "convergence": 1.0E-12, "relaxation": 0.9 },
    "plane": { "name": "MyAirplane", "CGx": 0, "CGy": 0, "CGz": 0 },
    "reference": { "area": 8.0, "longitudinal_length": 1.0, "lateral_length": 8.0 },
    "condition": { "alpha": 5.0, "beta": 0.0 },
    "airfoil_DB": "./AirfoilDB",
    "wings": {
        "wing_main": {
            "name": "wing_main", "ID": 1, "is_main": 1, "side": "both",
            "connect": {
                "ID": 0, "location": "tip", "dx": 0, "dy": 0, "dz": 0, "yoffset": 0
            },
            "span": 4, "mounting_angle": 0,
            "sweep": 0.0, "dihedral": 0.0, "washout": 0,
            "root_chord": 1.27323954473516, "tip_chord": -1,
            "airfoils": { "NACA_2412": "" },
            "grid": 100,
            "control": {}
        }
    }
}
```

E.2 Example Parameter File for a NACA 2412 Airfoil in Inviscid Incompressible Flow

```
{
    "NACA_2412": {
        "properties": {
            "type": "linear",
            "alpha_L0": -0.0380,
            "CL_alpha":  6.8583,
            "Cm_L0": 0.0,
            "Cm_alpha": 0.0,
            "CD0": 0.0,
            "CD0_L": 0.0,
            "CD0_L2": 0.0,
            "CL_max": 1.4,
            "Comments": "All angles in radians and slopes in 1/radians"
        }
    }
}
```

E.3 Example Profile File for a NACA 2412 Airfoil

```
      50
1.0000000000000000E+00  -7.1367152187917782E-09
9.9585604667663574E-01  -3.1594577012583613E-04
9.8349648714065552E-01  -1.2519687879830599E-03
9.6313685178756714E-01  -2.7746756095439196E-03
9.3513005971908569E-01  -4.8345020040869713E-03
8.9995884895324707E-01  -7.3724603280425072E-03
8.5822510719299316E-01  -1.0325845330953598E-02
8.1063729524612427E-01  -1.3630562461912632E-02
7.5799691677093506E-01  -1.7218593508005142E-02
7.0118451118469238E-01  -2.1010775119066238E-02
6.4114278554916382E-01  -2.4906730279326439E-02
5.7886141538619995E-01  -2.8774760663509369E-02
5.1535928249359131E-01  -3.2445460557937622E-02
4.5166760683059692E-01  -3.5712052136659622E-02
3.8890376687049866E-01  -3.8347791880369186E-02
3.2840368151664734E-01  -4.0438853204250336E-02
2.7069056034088135E-01  -4.1869580745697021E-02
2.1664057672023773E-01  -4.2349778115749359E-02
1.6707630455493927E-01  -4.1626252233982086E-02
1.2276169657707214E-01  -3.9503570646047592E-02
8.4396116435527802E-02  -3.5854429006576538E-02
5.2605386823415756E-02  -3.0618341639637947E-02
2.7930552139878273E-02  -2.3790119215846062E-02
1.0814117267727852E-02  -1.5401563607156277E-02
1.5862619038671255E-03  -5.5013755336403847E-03
4.6834559179842472E-04   5.7065724395215511E-03
7.6267276890575886E-03   1.7224393784999847E-02
2.3013709113001823E-02   2.8722338378429413E-02
4.6425748616456985E-02   3.9908505976200104E-02
7.7515803277492523E-02   5.0407152622938156E-02
1.1579234898090363E-01   5.9802222996950150E-02
1.6062285006046295E-01   6.7684493958950043E-02
2.1124278008937836E-01   7.3695354163646698E-02
2.6677113771438599E-01   7.7561683952808380E-02
3.2623127102851868E-01   7.9118162393569946E-02
3.8857534527778625E-01   7.8316092491149902E-02
4.5230948925018311E-01   7.5411744415760040E-02
5.1669228076934814E-01   7.0949688553810120E-02
5.8073848485946655E-01   6.5182760357856750E-02
6.4338487386703491E-01   5.8385424315929413E-02
7.0359897613525391E-01   5.0850689411163330E-02
7.6039564609527588E-01   4.2882818728685379E-02
8.1285268068313599E-01   3.4793458878993988E-02
8.6012434959411621E-01   2.6899019256234169E-02
9.0145480632781982E-01   1.9516089931130409E-02
9.3618875741958618E-01   1.2953279539942741E-02
9.6377998590469360E-01   7.4985213577747345E-03
9.8379832506179810E-01   3.4026026260107756E-03
9.9593400955200195E-01   8.6140306666493416E-04
1.0000000000000000E+00   7.1367152187917782E-09
```

F OPTIX SOURCE CODE

```python
import json
from myjson import myjson
from collections import OrderedDict, Iterable
import numpy as np
import os
import shutil
import time

import multiprocessing

np.set_printoptions(precision = 14)

zero = 1.0e-20


class objective_model(object):
    """Defines the evaluation model of an objective function

    This class defines a model consisting of an objective function that can be
    evaluated individually and with gradients. The class allows this function
    to be executed at multiple design points, either synchronously or
    asynchronously (using the Python multiprocessing module).

    Two methods for evaluating the function are used. The first method is
    required and evaluates only the function itself, while the second method
    is optional and evaluates the function and its gradient with respect to the
    design variables. If the second function is not provided, a second-order
    central differencing scheme is used to approximate gradients when needed.
    """
    def __init__(self,
                 objective_fcn,
                 objective_fcn_with_gradient = None,
                 max_processes = 1,
                 dx = 0.01
                ):
        """Constructor

        Constructor for the objective_model class

        Inputs
        ------
        objective_fcn:
            The function to evaluate through this model. The function should
            accept two arguments. The first argument is a list of design
            variable values specifying a fixed design point. The second
            argument is the case number assigned to the evaluation. The
            function should return a single result that is the value of the
            objective function at the specified design point.
        objective_fcn_with_gradient:
            The function to evaluate through this model when gradients are
            requested. The function should accept the same two arguments as
            objective_fcn. The function should return two results: the value
            of the objective function at the specified design point and the
            gradient of the objective function at the specified design point.
            Note that the first return value should be equal to the
            objective_fcn return value for a given design point.

            If objective_fcn_with_gradient is not specified (default), a
            second-order central difference approximation will be used with
```

```
            objective_fcn when gradients are needed.

        max_processes:
            The maximum number of simultaneous processes to use. If set to 1
            (default), all function evaluations will be executed sequentially.
            Otherwise, the Python multiprocessing module will be used to
            execute multiple function evaluations simultaneously.

        dx:
            The perturbation size to use if the second-order central difference
            approximation is used to estimate the gradient of the function. If
            objective_fcn_with_gradient is specified, dx is not used.
        """
        # Set the objective function
        self.obj_fcn = objective_fcn

        # Set the gradient function
        if objective_fcn_with_gradient is not None:
            # Use the user-specified function to calculate gradients
            self.obj_fcn_with_gradient = objective_fcn_with_gradient
        else:
            # Use central differencing scheme to approximate the gradient
            self.obj_fcn_with_gradient = self.central_difference
            self.dx = dx

        # Set the maximum number of simultaneous processes
        self.max_processes = max_processes

        # Initialize the number of function/gradient evaluations
        self.n_fcn_evals = 0
        self.n_grad_evals = 0


    def evaluate(self, design_points):
        """Evaluate the function at multiple design points

        This routine evaluates the objective function at multiple design
        points, each specified by a list of design variables.

        Inputs
        ------
        design_points = A list of design points. A design point is defined by
                        a list of design variables that are passed into the
                        objective function for a single evaluation. Therefore,
                        design_points is a list of lists.

        Outputs
        -------
        objective = A list of results from the objective function,
                    corresponding to the value of the objective function at
                    each design point specified.

        """
        objective = []
        if self.max_processes > 1:
            # Execute function at multiple design points in parallel
            with multiprocessing.Pool(processes = self.max_processes) as pool:
                args = [(design_points[i], i + 1)
                        for i in range(len(design_points))]
                objective = pool.map(self.obj_fcn, args)
        else:
```

```
            # Execute function at each design point sequentially
            for i in range(len(design_points)):
                objective.append(self.obj_fcn((design_points[i], i + 1)))

        # Increment the number of function evaluations
        self.n_fcn_evals += len(design_points)

        return objective


    def evaluate_gradient(self, design_point):
        """Evaluate the function and its gradient at a specified design point

        This routine evaluates the objective function and its gradient at a
        single design point.

        Inputs
        ------
        design_point = The design point at which to evaluate the function and
                       its gradient. The design point is defined as a list of
                       values, one value for each design variable required by
                       the objective function.

        Outputs
        -------
        objective = The value of the objective function at the specified design
                    point.

        gradient = The gradient of the objective function at the specified
                   design point.
        """
        objective, gradient = self.obj_fcn_with_gradient((design_point, 0))
        self.n_fcn_evals += 1
        self.n_grad_evals += 1

        return objective, gradient


    def central_difference(self, args):
        """Approximate the gradient of a function using central differencing

        This routine approximates the gradient of a specified function with
        respect to all design variables at a specified design point. The
        gradient is approximated using second-order central differencing.

        Inputs
        ------
        design_point = A list of design variables defining the design point at
                       which the objective function and its gradient will be
                       evaluated

        case_id = The case ID to use for the objective function evaluation. The
                  case IDs for gradient evaluations will be incremented
                  sequentially starting from (case_id + 1).
        """
        design_point = args[0]
        case_id = args[1]
        # Initialize a list of objective function arguments by perturbing each
        # variable by +/-dx
        n_design_vars = len(design_point)
        argslist = [(design_point[:], i) for i in range(case_id,
```

```
                case_id + 2 * n_design_vars + 1)]
        for i in range(1, n_design_vars + 1):
            argslist[i][0][i - 1] += self.dx
            argslist[i + n_design_vars][0][i - 1] -= self.dx

        if self.max_processes > 1:
            # Execute function at multiple design points in parallel
            with multiprocessing.Pool(processes = self.max_processes) as pool:
                results = pool.map(self.obj_fcn, argslist)
        else:
            # Execute function at each design point sequentially
            results = []
            for a in argslist:
                results.append(self.obj_fcn(a))

        # Get the objective function value at the specified design point
        objective = results[0]

        # Calculate the gradient of the objective function from results
        # at the perturbed design points
        gradient = []
        for i in range(1, n_design_vars + 1):
            gradient.append((results[i] - results[i + n_design_vars]) /
                    (2.0 * self.dx))

        return objective, gradient


class settings(object):
    """Defines the various settings used by the optimization algorithm
    """
    def __init__(self):
        self.opt_file = 'optimization.txt'
        self.grad_file = 'gradient.txt'
        self.verbose = False

        self.nvars = 0
        self.varnames = []
        self.varsinit = []
        self.opton = []

        self.default_alpha = 0.0
        self.stop_delta = 1.0E-12
        self.nsearch = 8
        self.line_search_type = 'quadratic'
        self.alpha_tol = 0.1
        self.max_refinements = 100
        self.rsq_tol = 0.99
        self.max_alpha_factor = 100

        self.wolfe_armijo = 1.0e-4
        self.wolfe_curv = 0.9

        self.nconstraints = 0
        self.constrainttype = []
        self.constraintnames = []
        self.constraintvalues = []
        self.penalty = []
        self.penalty_factor = []
```

```
    def load(settings_file):
        self = settings()
        input = myjson(settings_file)

        # Read settings from JSON file
        json_settings = input.get('settings', OrderedDict)
        self.default_alpha = json_settings.get('default_alpha', float)
        self.stop_delta = json_settings.get('stop_delta', float)
        self.nsearch = json_settings.get('n_search', int)
        self.line_search_type = json_settings.get('line_search_type', str,
                'quadratic')
        self.verbose = json_settings.get('verbose', bool, False)  # optional

        self.alpha_tol = json_settings.get('alpha_tol', float, self.alpha_tol)
        self.max_refinements = json_settings.get('max_refinements', int,
                self.max_refinements)
        self.rsq_tol = json_settings.get('rsq_tol', float, self.rsq_tol)
        self.max_alpha_factor = json_settings.get('max_alpha_factor', int,
                self.max_alpha_factor)

        self.wolfe_armijo = json_settings.get('wolfe_armijo', float,
                self.wolfe_armijo)
        self.wolfe_curv = json_settings.get('wolfe_curvature', float,
                self.wolfe_curv)

        # Read variables
        json_variables = input.get('variables', OrderedDict, OrderedDict())
        self.nvars = 0
        self.varnames = []
        self.varsinit = []
        self.opton = []
        for var_name in json_variables.data:
            self.add_variable(var_name,
                    json_variables.get(var_name +'.init', float),
                    json_variables.get(var_name + '.opt', str) == 'on')

        # Read constraints
        json_constraints = input.get('constraints', OrderedDict, OrderedDict())
        self.nconstraints = len(json_constraints.data)
        self.nconstraints = 0
        self.contrainttype = []
        self.constraintnames = []
        self.constraintvalues = []
        self.penalty = []
        self.penalty_factor = []
        valid_constraint_types = ['=', '<', '>']
        for const_name in json_constraints.data:
            json_constraint_data = json_constraints.get(const_name,
                    OrderedDict)
            const_type = json_constraint_data.get('type', str)
            if const_type not in valid_constraint_types:
                print('Unknown constraint type: {0}. Constraint {1} skipped.'
                    .format(const_type, const_name))
                print('Valid constraint types are {0}.'
                        .format(valid_constraint_types))
                continue

            self.constrainttype.append(const_type)
            self.constraintnames.append(const_name)
            self.constraintvalues.append(
                    json_constraint_data.get('value', float))
```

```
            self.penalty.append(
                    json_constraint_data.get('penalty', float))
            self.penalty_factor.append(
                    json_constraint_data.get('factor', float))

        return self


    def write(self, settings_file):
        data = OrderedDict()
        data['settings'] = OrderedDict()
        data['settings']['default_alpha'] = self.default_alpha
        data['settings']['stop_delta'] = self.stop_delta
        data['settings']['n_search'] = self.nsearch
        data['settings']['line_search_type'] = self.line_search_type
        data['settings']['verbose'] = self.verbose

        data['settings']['alpha_tol'] = self.alpha_tol
        data['settings']['max_refinements'] = self.max_refinements
        data['settings']['rsq_tol'] = self.rsq_tol
        data['settings']['max_alpha_factor'] = self.max_alpha_factor

        data['settings']['wolfe_armijo'] = self.wolfe_armijo
        data['settings']['wolfe_curvature'] = self.wolfe_curv

        data['variables'] = OrderedDict()
        for i in range(self.nvars):
            data['variables'][self.varnames[i]] = OrderedDict()
            data['variables'][self.varnames[i]]['init'] = self.varsinit[i]
            data['variables'][self.varnames[i]]['opt'] = self.opton[i]

        with open(settings_file, 'w') as settings:
            json.dump(data, settings, indent = 4)


    def add_variable(self, varname, varinit, opton = True):
        if varname in self.varnames:
            i = self.varnames.index(varname)
            self.varsinit[i] = varinit
            self.opton[i] = opton
        else:
            self.varnames.append(varname)
            self.varsinit.append(varinit)
            self.opton.append(opton)
            self.nvars += 1


def optimize(obj_model, settings):
    """
    """
    header = ('{0:>4}, {1:>5}, {2:>5}, {3:>20}, {4:>20}, {5:>20}'
        .format('iter', 'outer', 'inner', 'fitness', 'alpha', 'mag(dx)'))
    for name in settings.varnames: header += ', {0:>20}'.format(name)
    with open(settings.opt_file, 'w') as opt_file:
        opt_file.write(header + '\n')

    with open(settings.grad_file, 'w') as grad_file:
        grad_file.write(header + '\n')

    print('---------- Variables ----------')
    for i in range(settings.nvars):
```

```
        print('{0} = {1}'.format(settings.varnames[i], settings.varsinit[i]))
    print('')

    print('---------- Constraints ----------')
    for i in range(settings.nconstraints):
        print('{0}, {1}, {2}'.format(settings.constraintnames[i],
            settings.constrainttype[i], settings.constraintvalues[i]))
    print('')

    print('---------- Settings ----------')
    print('      default alpha: {0}'.format(settings.default_alpha))
    print('     stopping delta: {0}'.format(settings.stop_delta))
    print('')

    iter = 0
    o_iter = 0
    mag_dx = 1.0
    design_point = settings.varsinit[:]
    while mag_dx > settings.stop_delta:
        design_point_init = np.copy(design_point)
        i_iter = 0

        print('Constraint Penalties')
        for i in range(settings.nconstraints):
            print('{0} {1}'.format(settings.constraintnames[i],
                    settings.penalty[i]))

        print('Beginning new update matrix')
        print(header)

        alpha = 0.0
        while mag_dx > settings.stop_delta:
            obj_value, gradient = obj_model.evaluate_gradient(design_point)
            append_file(iter, o_iter, i_iter, obj_value, alpha, mag_dx,
                    design_point, gradient, settings)

            # Initialize N to the identity matrix
            if (i_iter == 0):
                N = np.eye(settings.nvars)  # n x n

            else:
                dx = np.matrix(design_point - design_point_prev)  # 1 x n
                gamma = np.matrix(gradient - gradient_prev)  # 1 x n
                NG = N * np.transpose(gamma)  # n x 1
                denom = dx * np.transpose(gamma)  # 1 x 1
                N += ((1.0 + np.dot(gamma, NG) / denom)[0,0] *
                        (np.transpose(dx) * dx) / denom
                      - ((np.transpose(dx) * (gamma * N)) + (NG * dx)) / denom
                     )

                # Calculate the second Wolfe condition for the previous
                # iteration. The curvature condition ensures that the slope is
                # sufficiently large to contribute to a reduction in the
                # objective function. If this condition is not met, the inner
                # loop is stopped and the direction matrix is reset to the
                # direction of steepest descent.
                if np.dot(s, gradient) < settings.wolfe_curv *
                        np.dot(s, gradient_prev):
                    print("Wolfe condition (ii): curvature condition " +
                            "not satisified!")
                    break
```

```
            s = -np.dot(N, gradient)
            design_point_prev = np.copy(design_point)
            gradient_prev = np.copy(gradient)

            alpha, design_point = line_search(design_point[:], obj_value,
                    gradient, s, obj_model, settings)

            dx = design_point - design_point_prev
            mag_dx = np.linalg.norm(dx)
            i_iter += 1
            iter += 1

        dx = design_point - design_point_init
        mag_dx = np.linalg.norm(dx)
        append_file(iter, o_iter, i_iter, obj_value, alpha, mag_dx,
                design_point, gradient, settings)

        o_iter += 1
        for i in range(settings.nconstraints):
            settings.penalty[i] = (settings.penalty[i] *
                    settings.penalty_factor[i])

    # Run the final case
    obj_value = obj_model.obj_fcn((design_point, -1))
    append_file(iter, o_iter, i_iter, obj_value, 0.0, mag_dx, design_point,
            gradient, settings)
    return (obj_value, design_point)


def line_search(design_point, obj_value, gradient, s, obj_model, settings):
    if settings.line_search_type == 'quadratic':
        return line_search_quad(design_point, obj_value, gradient, s,
                obj_model, settings)
    else:
        return line_search_lin(design_point, obj_value, s, obj_model, settings)


def line_search_lin(design_point, obj_value, s, obj_model, settings):
    if settings.verbose:
        print('line search ----------------------------------------------')

    s_norm = np.linalg.norm(s)
    alpha = max(settings.default_alpha, 1.1 * settings.stop_delta / s_norm)
    alpha_mult = settings.nsearch / 2.0

    found_min = False
    while not found_min:
        xval, yval = run_mult_cases(settings.nsearch, alpha, s, design_point,
                obj_value, obj_model)
        if settings.verbose:
            for i in range(settings.nsearch + 1):
                print('{0:5d}, {1:15.7E}, {2:15.7E}'
                        .format(i, xval[i], yval[i]))

        mincoord = yval.index(min(yval))
        if yval[1] > yval[0]:
            if (alpha * s_norm) < settings.stop_delta:
                print('Line search within stopping tolerance: alpha = {0}'
                        .format(alpha))
                return alpha, design_point
```

```
            elif mincoord == 0:
                if settings.verbose: print('Too big of a step. Reducing alpha')
                alpha /= alpha_mult
            else:
                if mincoord < settings.nsearch: found_min = True
                else: alpha *= alpha_mult
        else:
            if settings.verbose: print('mincoord = {0}'.format(mincoord))
            if mincoord == 0: return alpha, design_point
            elif mincoord < settings.nsearch: found_min = True
            else: alpha *= alpha_mult

    a1 = xval[mincoord - 1]
    a2 = xval[mincoord]
    a3 = xval[mincoord + 1]
    f1 = yval[mincoord - 1]
    f2 = yval[mincoord]
    f3 = yval[mincoord + 1]

    da = a2 - a1
    alpha = a1 + da * (4.0 * f2 - f3 - 3.0 * f1) / (2.0 * (2.0 * f2 - f3 - f1))
    if alpha > a3 or alpha < a1:
        if f2 > f1: alpha = a1
        else: alpha = a2

    for i in range(len(design_point)):
        design_point[i] += alpha * s[i]

    if settings.verbose: print('Final alpha = {0}'.format(alpha))
    return alpha, design_point


def line_search_quad(design_point, obj_value, gradient, s, obj_model, settings):
    """Perform a quadratic line search to minimize the objective function

    This subroutine evaluates the objective function multiple times in the
    direction of s and fits a parabola to the results using a least-squares
    algorithm to identify the minimum value for the objective function in
    the current direction.

    Inputs
    ------
    design_point = A list of design variables defining the design point at
                   which to begin the line search

    obj_value = The value of the objective function at the specified design
                point.
    gradient = The gradient of the objective function at the specified design
               point.

    s = The direction matrix defining the direction in which to conduct the
        line search.

    obj_model = The objective model object

    settings = The optimization settings object
    Outputs
    -------
    alpha_min = The alpha corresponding to the minimum value of the objective
                function in the current direction

```

```
    design_point = The design point corresponding to the minimum value of the
                   objective function in the current direction
    """
    if settings.verbose:
        print('Performing quadratic line search...')

    # Determine the initial step size to use in the direction of s
    stop_delta = settings.stop_delta / np.linalg.norm(s)
    alpha = max(settings.default_alpha, 1.1 * stop_delta)

    found_min = False
    line_search_min = (0.0, obj_value)
    alpha_history = []

    # Determine the maximum number of adjustments in alpha to attempt.
    nadjust = int(np.ceil(-np.log10(stop_delta) /
        np.log10(np.ceil(settings.nsearch / 2))))

    for i in range(nadjust):
        # Compute the objective function multiple times in the direction of s
        alphas, obj_vals = run_mult_cases(settings.nsearch, alpha, s,
                design_point, obj_value, obj_model)
        alpha_history.append(alpha)

        # Save the minimum data point for later comparisons
        ind = obj_vals.index(min(obj_vals))
        if obj_vals[ind] < line_search_min[1]:
            line_search_min = (alphas[ind], obj_vals[ind])
        alpha_min_est = line_search_min[0]

        if settings.verbose:
            for j in range(settings.nsearch + 1):
                print('{:5d}, {:23.15E}, {:23.15E}'.format(
                    j, alphas[j], obj_vals[j]))

        # Check for invalid results
        if np.isnan(obj_vals).any():
            print('Found NaN')
            break

        # Check for plateau
        if min(obj_vals) == max(obj_vals):
            print('Objective function has plateaued')
            break

        # Check stopping criteria
        if alpha <= stop_delta and ind < settings.nsearch - 1:
            print('stopping criteria met')
            break

        # Fit a quadratic through the data and find the resulting minimum
        q = quadratic(np.asarray(alphas), np.asarray(obj_vals))
        (alpha_min_est, obj_value_est) = q.vertex()

        if (alpha_min_est is None or alpha_min_est < 0 or not q.convex() or
                q.rsq < settings.rsq_tol):
            # Can't find a better minimum by curve fitting all data points.
            # Try a quadratic through minimum and two closest neighbors.
            left = min(max(ind - 1, 0), len(alphas) - 3)
            right = left + 3
            q = quadratic(np.asarray(alphas[left:right]),
```

```
                    np.asarray(obj_vals[left:right]))
            (alpha_min_est, obj_value_est) = q.vertex()

            if (alpha_min_est is None or alpha_min_est < 0 or not q.convex()):
                if ind == settings.nsearch:
                    # If minimum is at the end, try increasing alpha
                    alpha_min_est = alpha * 4
                elif ind == 0:
                    # If minimum is at beginning, try reducing alpha
                    alpha_min_est = alpha / 2
                else:
                    # Can't find a better minimum by curve fitting,
                    # so just use the current minimum.
                    break

        # Set alpha for next iteration
        alpha = max(alpha / settings.max_alpha_factor,
                min(alpha * settings.max_alpha_factor,
                alpha_min_est / np.ceil(settings.nsearch / 2.0)))
        print('alpha for next iteration = ', alpha)

        # Check to see if we've already tried close to this alpha
        alpha_close = min(alpha_history, key=lambda a: abs(a - alpha) / alpha)
        delta = abs(alpha_close - alpha) / alpha
        if delta <= settings.alpha_tol:
            break

    # Update design point based on alpha that minimized objective function
    alpha_min = line_search_min[0]
    design_point[:] += alpha_min * s[:]

    # Calculate the first Wolfe condition. This is a measure of how much the
    # step length (alpha) decreases the objective function, but has no effect
    # on the behavior of the quadratic line search.
    armijo = obj_value + settings.wolfe_armijo * alpha_min * np.dot(s, gradient)
    if line_search_min[1] > armijo:
        print("Wolfe condition (i): Armijo rule not satisfied.")

    if settings.verbose: print('Line search minimized at alpha = {0}'
            .format(alpha_min))
    return alpha_min, design_point


def run_mult_cases(nevals, alpha, s, dp0, obj_fcn0, obj_model):
    # Calculate linearly distributed alphas for the line search
    alphas = [(i * alpha) for i in range(nevals + 1)]

    # Set up the design points in the direction of the line search
    design_points = []
    for i in range(nevals):
        design_points.append([(dp0[j] + alphas[i + 1] * s[j])
                for j in range(len(dp0))])

    # Evaluate the function at each design point
    obj_fcn_values = [obj_fcn0] + obj_model.evaluate(design_points)

    return alphas, obj_fcn_values


def append_file(iter, o_iter, i_iter, obj_fcn_value, alpha, mag_dx,
        design_point, gradient, settings):
```

```
    msg = ('{0:4d}, {1:5d}, {2:5d}, {3: 20.13E}, {4: 20.13E}, {5: 20.13E}'
        .format(iter, o_iter, i_iter, obj_fcn_value, alpha, mag_dx))
    values_msg = msg
    for value in design_point:
        values_msg = ('{0}, {1: 20.13E}'.format(values_msg, value))
    print(values_msg)
    with open(settings.opt_file, 'a') as opt_file:
        print(values_msg, file = opt_file)

    grad_msg = msg
    for grad in gradient:
        grad_msg = ('{0}, {1: 20.13E}'.format(grad_msg, grad))
    with open(settings.grad_file, 'a') as grad_file:
        print(grad_msg, file = grad_file)


class quadratic(object):
    """Class for fitting, evaluating, and interrogating quadratic functions
    This class is used for fitting a quadratic function to a data set
    evaluating the function at specific points, and determining the
    characteristics of the function.
    """
    def __init__(self, x, y):
        """
        Construct a quadratic object from tabulated data.
        Quadratic is of the form f(x) = ax^2 + bx + c
        Inputs
        ------
        x = List of independent values
        y = List of dependent values
        """
        super().__init__()

        # Calculate the quadratic coefficients
        x_sq = [xx**2 for xx in x]
        A = np.vstack([x_sq, x, np.ones(len(x))]).T
        self.a, self.b, self.c = np.linalg.lstsq(A, y)[0]

        # Calculate the coefficient of determination
        f = [self.f(xx) for xx in x]
        ssres = ((f - y)**2).sum()
        sstot = ((y - y.mean())**2).sum()

        if abs(sstot) < zero:
            # Data points actually formed a horizontal line
            self.rsq = 0.0
        else:
            self.rsq = 1 - ssres / sstot


    def convex(self):
        """
        Test to see if the quadratic is convex (opens up).
        """
        # Convex has positive curvature (2nd derivative)
        # f"(x) = 2a, so a > 0 corresponds to convex
        return (self.a > 0)


    def vertex(self):
        """
```

```
        Find the coordinates of the vertex
        """
        if self.a != 0.0:
            # Find x where f'(x) = 2ax + b = 0
            x = -0.5 * self.b / self.a
            return (x, self.f(x))
        else:
            # Quadratic is actually a line, no minimum!
            return (None, None)


    def f(self, x):
        """
        Evaluate the quadratic function at x
        """
        if x is not None: return self.a * x**2 + self.b * x + self.c
        else: return None


class myjson:
    def __init__(self, filename=None, parent=None, data=None, path=None):
        self.file = ''
        self.data = OrderedDict()
        self.path = ''

        if filename is not None: self.load(filename)
        elif parent is not None:
            self.file = parent.file
            self.data = data
            self.path = parent.path + path


    def load(self, filename):
        if not os.path.isfile(filename):
            print('Error: Cannot find file "{0}". Make sure'.format(filename))
            print('       the path is correct and the file is accessible.')
            raise IOError(filename)

        self.file = filename
        with open(self.file) as file:
            self.data = json.load(file, object_pairs_hook = OrderedDict)


    def get(self, value_path, value_type, default_value = None):
        abs_path = self.path + '.' + value_path

        json_data = self.data
        for path in value_path.split('.'):
            try:
                json_data = json_data[path]
            except KeyError as exc:
                if default_value is None:
                    print('Error: required JSON path not found. Op aborted.')
                    print('       Missing path is "{0}"'.format(abs_path))
                    raise
                else:
                    json_data = default_value

        if not isinstance(value_type, Iterable): value_type = [value_type]
        if type(json_data) not in value_type:
            print('Error: JSON value is of an incorrect type.')
```

```
            print('       Expected {0} but found {1}'
                .format(value_type, type(json_data)))
            print('       Invalid path is "{0}"'.format(abs_path))
            raise KeyError(value_path)

        if type(json_data) is OrderedDict:
            return myjson(parent = self, data = json_data, path = value_path)
        else:
            return json_data


def load(filename):
    return myjson(filename)
```

G INPUT FILES AND PYTHON SCRIPTS FOR WING SHAPE OPTIMIZATION

The sections below list the Python scripts and main input file required to run the optimization analyses presented in Secs. 3.3 and 3.4. The objective functions (`evaluate` and `evaluate_with_gradient`) and the `get_list_of_vars` helper function are contained in a single Python script file called `obj_fcn.py`. Required MachUp executables, the main input file (`input.json`), and all other supporting input files referenced in the main input file must be contained in a folder called `OrigFiles` that must reside within the current working directory. An airfoil database must also exist within the current working directory that contains all the necessary airfoils referenced by the input files. Note that the files in this database will be different between inviscid and viscous analyses.

G.1 Main Optimization Execution Script

```
import optix
import argparse
import obj_fcn
from timeit import default_timer as timer

parser = argparse.ArgumentParser()
parser.add_argument('-nchord',
        help = '-nchord N (Number of chord control points)',
        type = int, required = False, default = 0)
parser.add_argument('-ntwist',
        help = '-ntwist N (Number of twist control points)',
        type = int, required = False, default = 0)
parser.add_argument('-ncamber',
        help = '-ncamber N (Number of camber control points)',
        type = int, required = False, default = 0)
args = parser.parse_args()
nchord = args.nchord
ntwist = args.ntwist
ncamber = args.ncamber

# Create a settings object
settings = optix.settings()
settings.default_alpha = 1.0
settings.stop_delta = 1e-10
settings.nsearch = 4
settings.line_search_type = 'quadratic'
settings.verbose = True
settings.alpha_tol = 0.1
settings.max_refinements = 100
settings.rsq_tol = 0.99
settings.max_alpha_factor = 100
settings.wolfe_armijo = 0.0001
settings.wolfe_curvature = 0.9

# Set up the control point parameters
for i in range(nchord):
    settings.add_variable('chord{}'.format(i), 1.0, True)

```

```
for i in range(1, ntwist):
    settings.add_variable('twist{}'.format(i), 0.0, True)

for i in range(ncamber):
    settings.add_variable('camber{}'.format(i), 1.0, True)

settings.write('settings.json')

# Number of design variables
ndv = nchord + ntwist + ncamber + 1  # Add 1 for angle of attack
if nchord > 0: ndv += 2  # Add 1 for area and 1 for longitudinal length
if ntwist > 0: ndv -= 1  # Subtract 1 for root twist

# Create the objective model
model = optix.objective_model(obj_fcn.evaluate,
        obj_fcn.evaluate_with_gradient, max_processes = 4)

# Begin the optimization
start = timer()
optix.optimize(model, settings)

end = timer()
print("Optimization time: {0} seconds.".format(end - start))
print("Number of function evaluations: {0}".format(model.n_fcn_evals))
print("Number of gradient evaluations: {0}".format(model.n_grad_evals))
```

G.2 Objective Function (`obj_fcn.evaluate`)

```
import os
import shutil
import json
import numpy as np
from collections import OrderedDict

def evaluate(args):

    # Get the list of variables
    with open('settings.json', 'r') as settings_file:
        settings_data = json.load(settings_file,
                object_pairs_hook = OrderedDict)

    chord = get_list_of_vars(settings_data, 'chord', 0.01, 3.0, args[0])
    twist = get_list_of_vars(settings_data, 'twist', -90.0, 90.0, args[0])
    camber = get_list_of_vars(settings_data, 'camber', 0.0, 5.0, args[0])

    case_id = args[1]

    # Get the current working directory
    work_dir = os.getcwd()

    # Get the current working directory and case directory names
    orig_dir = work_dir + '/' + 'OrigFiles'
    case_dir = work_dir + '/' + str(case_id)

    # Remove existing folder with same case ID
    if os.path.exists(case_dir): shutil.rmtree(case_dir)

    # Copy original files into case directory
    shutil.copytree(orig_dir, case_dir)

```

```
    # Make the temporary directory current
    os.chdir(case_dir)

    # Generate chord input file
    if len(chord) > 0:
        # Get the desired area, wingspan, and average chord
        with open('input.json', 'r') as input_file:
            input_data = json.load(input_file, object_pairs_hook = OrderedDict)
        area = input_data['reference']['area']
        b = input_data['reference']['lateral_length']
        cavg = area / b

        # Calculate the chord length at the wing tip
        c_tip = 2.0 * len(chord) * cavg - chord[0] - 2.0 * sum(chord[1:])
        chord += [c_tip]

        # Write the chord files
        y_chord = np.linspace(0.0, 1.0, len(chord))
        design_vars = {}
        for i in range(len(chord)):
            row = 'r' + str(i + 1)
            design_vars[row] = {}
            design_vars[row]['c1'] = y_chord[i]
            design_vars[row]['c2'] = chord[i]
        with open('chord_left.json', 'w') as data_file:
            json.dump(design_vars, data_file, sort_keys = False, indent = 4)
        with open('chord_right.json', 'w') as data_file:
            json.dump(design_vars, data_file, sort_keys = False, indent = 4)

    # Generate twist input file
    if len(twist) > 0:
        twist = [0.0] + twist
        y_twist = np.linspace(0.0, 1.0, len(twist))
        design_vars = {}
        for i in range(len(twist)):
            row = 'r' + str(i + 1)
            design_vars[row] = {}
            design_vars[row]['c1'] = y_twist[i]
            design_vars[row]['c2'] = twist[i]
        with open('twist_left.json', 'w') as data_file:
            json.dump(design_vars, data_file, sort_keys = False, indent = 4)
        with open('twist_right.json', 'w') as data_file:
            json.dump(design_vars, data_file, sort_keys = False, indent = 4)

    # Generate camber input file
    if len(camber) > 0:
        y_camber = np.linspace(0.0, 1.0, len(camber))
        design_vars = {}
        for i in range(len(camber)):
            row = 'r' + str(i + 1)
            design_vars[row] = {}
            design_vars[row]['c1'] = y_camber[i]
            design_vars[row]['c2'] = camber[i]
        with open('af_ratio_left.json', 'w') as data_file:
            json.dump(design_vars, data_file, sort_keys = False, indent = 4)
        with open('af_ratio_right.json', 'w') as data_file:
            json.dump(design_vars, data_file, sort_keys = False, indent = 4)

    # Execute MachUp with DNAD integration
    os.system('./MachUp_DNAD1.out input.json > out.txt')

```

```
    # Extract cost function from MachUp results
    with open('input_forces.json') as forces_file:
        forces_data = json.load(forces_file)

    # Get the drag and DNAD derivatives
    cd = forces_data['total']['MyAirplane']['CD'][0]

    # Move to the original work directory
    os.chdir(work_dir)

    return cd
```

G.3 Objective Function with Gradient Calculations (`obj_fcn.evaluate_with_gradient`)

```
import os
import shutil
import json
import numpy as np
from collections import OrderedDict

def evaluate_gradient(args):
    # Get the list of variables
    with open('settings.json', 'r') as settings_file:
        settings_data = json.load(settings_file, object_pairs_hook = OrderedDict)

    chord = get_list_of_vars(settings_data, 'chord', 0.1, 10.0, args[0])
    twist = get_list_of_vars(settings_data, 'twist', -90.0, 90.0, args[0])
    camber = get_list_of_vars(settings_data, 'camber', 0.0, 4.0, args[0])
    ndv = len(chord) + len(twist) + len(camber) + 1  # Add 1 for alpha
    if len(chord) > 0: ndv += 1  # Add 1 for chord at wingtip

    case_id = args[1]

    # Get the current working directory
    work_dir = os.getcwd()

    # Get the current working directory and case directory names
    orig_dir = work_dir + '/' + 'OrigFiles'
    case_dir = work_dir + '/' + str(case_id)

    # Remove existing folder with same case ID
    if os.path.exists(case_dir): shutil.rmtree(case_dir)

    # Copy original files into case directory
    shutil.copytree(orig_dir, case_dir)

    # Make the temporary directory current
    os.chdir(case_dir)



    # Generate chord input file
    if len(chord) > 0:
        # Get the desired area, wingspan, and average chord
        with open('input.json', 'r') as input_file:
            input_data = json.load(input_file, object_pairs_hook = OrderedDict)
        area = input_data['reference']['area']
        b = input_data['reference']['lateral_length']
        cavg = area / b
```

```

        # Calculate the chord length at the wing tip
        c_tip = 2.0 * len(chord) * cavg - chord[0] - 2.0 * sum(chord[1:])
        chord += [c_tip]

        # Write the chord files
        y_chord = np.linspace(0.0, 1.0, len(chord))
        design_vars = {}
        for i in range(len(chord)):
            row = 'r' + str(i + 1)
            design_vars[row] = {}
            design_vars[row]['c1'] = y_chord[i]
            design_vars[row]['c2'] = [chord[i], 1.0]
        with open('chord_left.json', 'w') as data_file:
            json.dump(design_vars, data_file, sort_keys = False, indent = 4)
        design_vars = {}
        for i in range(len(chord)):
            row = 'r' + str(i + 1)
            design_vars[row] = {}
            design_vars[row]['c1'] = y_chord[i]
            design_vars[row]['c2'] = chord[i]
        with open('chord_right.json', 'w') as data_file:
            json.dump(design_vars, data_file, sort_keys = False, indent = 4)

    # Generate twist input file
    if len(twist) > 0:
        twist = [0.0] + twist
        y_twist = np.linspace(0.0, 1.0, len(twist))
        design_vars = {}
        for i in range(len(twist)):
            row = 'r' + str(i + 1)
            design_vars[row] = {}
            design_vars[row]['c1'] = y_twist[i]
            design_vars[row]['c2'] = [twist[i], 1.0] if i > 0 else twist[i]
        with open('twist_left.json', 'w') as data_file:
            json.dump(design_vars, data_file, sort_keys = False, indent = 4)

        for i in range(len(twist)):
            row = 'r' + str(i + 1)
            design_vars[row] = {}
            design_vars[row]['c1'] = y_twist[i]
            design_vars[row]['c2'] = twist[i]
        with open('twist_right.json', 'w') as data_file:
            json.dump(design_vars, data_file, sort_keys = False, indent = 4)

    # Generate camber input file
    if len(camber) > 0:
        y_camber = np.linspace(0.0, 1.0, len(camber))
        design_vars = {}
        for i in range(len(camber)):
            row = 'r' + str(i + 1)
            design_vars[row] = {}
            design_vars[row]['c1'] = y_camber[i]
            design_vars[row]['c2'] = [camber[i], 1.0]
        with open('af_ratio_left.json', 'w') as data_file:
            json.dump(design_vars, data_file, sort_keys = False, indent = 4)

        for i in range(len(camber)):
            row = 'r' + str(i + 1)
            design_vars[row] = {}
            design_vars[row]['c1'] = y_camber[i]
```

```
            design_vars[row]['c2'] = camber[i]
        with open('af_ratio_right.json', 'w') as data_file:
            json.dump(design_vars, data_file, sort_keys = False, indent = 4)

    # Execute MachUp with DNAD integration
    os.system('./MachUp_DNAD{}.out input.json > out.txt'.format(ndv))

    # Extract cost function from MachUp results
    with open('input_forces.json') as forces_file:
        forces_data = json.load(forces_file)

    # Get the induced drag and DNAD derivatives
    cd = forces_data['total']['MyAirplane']['CD'][0]

    # Calculate the gradient
    grad_left = forces_data['total']['MyAirplane']['CD'][2:]
    grad = [2 * gl for gl in grad_left]
    if len(chord) > 0:
        # Get d(CD)/d(c_tip)
        dCD_dctip = grad[len(chord) - 1]

        # Remove derivative for wingtip chord from gradient
        grad = grad[:len(chord) - 1] + grad[len(chord):]

        # Apply the partial derivative factor for the wingtip chord
        grad[0] += -dCD_dctip
        for i in range(1, len(chord) - 1):
            grad[i] += -2.0 * dCD_dctip

    # Move to the original work directory
    os.chdir(work_dir)

    return cd, grad
```

G.4 Helper Function for Extracting Variable Names (get_list_of_vars)

```
def get_list_of_vars(settings_data, prefix, minval, maxval, allvalues = None):
    # Get a mask identifying indices for variables with matching prefix
    mask = [var.find(prefix) == 0 for var in settings_data['variables']]

    # If allvalues is not specified, get initial values from settings file
    if allvalues is None:
        allvalues = [settings_data['variables'][var]['init']
                for var in settings_data['variables']]

    # Unpack variables with matching prefix into a list
    values = [min(max(allvalues[i], minval), maxval)
            for i in range(len(allvalues)) if mask[i]]

    # Return the resulting list
    return values
```

G.5 Main MachUp Input File

```
{
    "run": {
        "targetcl": {
            "CL": 0.5,
```

```
            "delta": 0.1,
            "relaxation": 1.0,
            "maxiter": 100,
            "convergence": 1.000001E-12,
            "run": 1
        },
        "forces": {"run": 1},
        "distributions": {"run": 0},
        "stl": {"run": 0}
    },
    "solver": {
        "type": "nonlinear",
        "convergence": 1.000001E-12,
        "relaxation": 0.9
    },
    "plane": {
        "name": "MyAirplane",
        "CGx": 0,
        "CGy": 0,
        "CGz": 0
    },
    "reference": {
        "area": 8.0,
        "longitudinal_length": 1.0,
        "lateral_length": 8.0
    },
    "condition": {
        "alpha": 3.0,
        "beta": 0.0
    },
    "airfoil_DB": "../AirfoilDatabase",
    "wings": {
        "Wing_left": {
            "name": "Wing_1",
            "ID": 1,
            "is_main": 1,
            "side": "left",
            "connect": {
                "ID": 0,
                "location": "tip",
                "dx": 0,
                "dy": 0,
                "dz": 0,
                "yoffset": 0
            },
            "span": 4,
            "sweep": 0.0,
            "dihedral": 0.0,
            "mounting_angle": 0,
            "washout": 0,
            "washout_file": "twist_left.json",
            "root_chord": 1,
            "tip_chord": 1,
            "chord_file": "chord_left.json",
            "airfoils": {
                "NACA_n2412": "",
                "NACA_0012": "",
                "NACA_2412": "",
                "NACA_4412": "",
                "NACA_6412": "",
                "NACA_8412": ""
```

```
                },
                "af_ratio_file": "af_ratio_left.json",
                "grid": 100,
                "control": {}
            },
            "Wing_right": {
                "name": "Wing_right",
                "ID": 1,
                "is_main": 1,
                "side": "right",
                "connect": {
                    "ID": 0,
                    "location": "tip",
                    "dx": 0,
                    "dy": 0,
                    "dz": 0,
                    "yoffset": 0
                },
                "span": 4,
                "sweep": 0.0,
                "dihedral": 0.0,
                "mounting_angle": 0,
                "washout": 0,
                "washout_file": "twist_right.json",
                "root_chord": 1,
                "tip_chord": 1,
                "chord_file": "chord_right.json",
                "airfoils": {
                    "NACA_n2412": "",
                    "NACA_0012": "",
                    "NACA_2412": "",
                    "NACA_4412": "",
                    "NACA_6412": "",
                    "NACA_8412": ""
                },
                "af_ratio_file": "af_ratio_right.json",
                "grid": 100,
                "control": {}
            }
        }
    }
```

H AERODYNAMIC CALCULATIONS USING PANAIR

Panair is an open-source Fortran code that implements a high-order panel method for performing potential flow analysis of arbitrary geometries. It was originally developed by Boeing Military Airplane Development under contract to NASA in the latter part of the twentieth century, and has seen wide use in industry and academia since that time. Source code,[*] documentation,[†,‡,§] and a list of research publications[**] related to Panair can be found online. Because of its established reliability in performing aerodynamic calculations, Panair has been used in the present research to perform baseline calculations against which the accuracy of various formulations of lifting line theory can be measured.

In order to rapidly process finite wing models of different configurations with Panair, several functions were added to the MachUp source code for converting wing geometry data into properly-formatted Panair input files. These functions are listed in Appendix I. For each wing configuration analyzed, a MachUp input file was generated with the planform shape, airfoil geometry, and number of spanwise sections defined. MachUp was then executed to generate a Panair input file for this geometry. The Panair input file was then processed by Panair to calculate pressure distributions over the wing surface. Upon successful execution, Panair produces an output file that defines the predicted pressure coefficient at each node specified in the input file. A suite of Python scripts were used to integrate the pressures calculated by Panair to determine the spanwise section lift distribution and the total lift generated by the wing.

A wing representation in the Panair input file is divided into two panel networks – one representing the upper surface of the wing and another representing the lower surface. For rectangular and tapered wings, where the tip chord is nonzero, a third panel network is needed to define an endcap to close the volume of the wing. This endcap network is automatically added to the Panair input file by MachUp. Since all wings considered in the present work are symmetric about the root plane, a symmetry boundary condition is defined at the root so that only one semispan of the wing is modeled. A representative Panair input file is included in Appendix J. This model represents an elliptic wing with an aspect ratio of $A = 4$, an average chord length of

---

[*] http://www.pdas.com/panairdownload.html
[†] https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19840020672.pdf
[‡] https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19920013405.pdf
[§] https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19920013622.pdf
[**] http://www.pdas.com/panairrefs.html

$c_{avg} = 1\text{m}$, and a symmetric Joukowski airfoil with maximum thickness $\bar{t}_{max} = 4\%$. The wing is operating at a geometric angle of attack of $\alpha = 1\deg$ in incompressible flow with freestream density $\rho_\infty = 1\text{kg}/\text{m}^3$ and velocity $V_\infty = 1\text{m}/\text{s}$. There are 11 chordwise nodes and 21 spanwise nodes each for the upper and lower wing surface networks, defining a combined 20×20 mesh of panels for a total of 400 panels in the model. The spanwise lift distribution computed using this input file is shown in Figure H.1.

The physical scale of the model was found to have a noticeable effect on the pressure results, especially near the wing tip and for fine panel meshes. The source of this variation in the results based on physical scale is unknown since the pressure coefficients written to the Panair output file are nondimensional. It is theorized that data written to and read from intermediate solution files on disk during the solution process may be of limited precision so that, when read in, rounding error is introduced into the solution. If true, this rounding error could then be reduced by increasing the physical scale of the model so that more digits of precision are written to disk. This theory has not been tested, however, and a different source of error may be present in the code. In order to alleviate this problem, multiple Panair solutions were evaluated with increasing scales until a reasonably converged solution was achieved. An example of this using a mesh size of 80×80 panels for the elliptic wing described previously is shown in Figure H.2. For the wing configurations considered in this work, the results become independent of scale for $\bar{c} \geq 5\,\text{m}$.

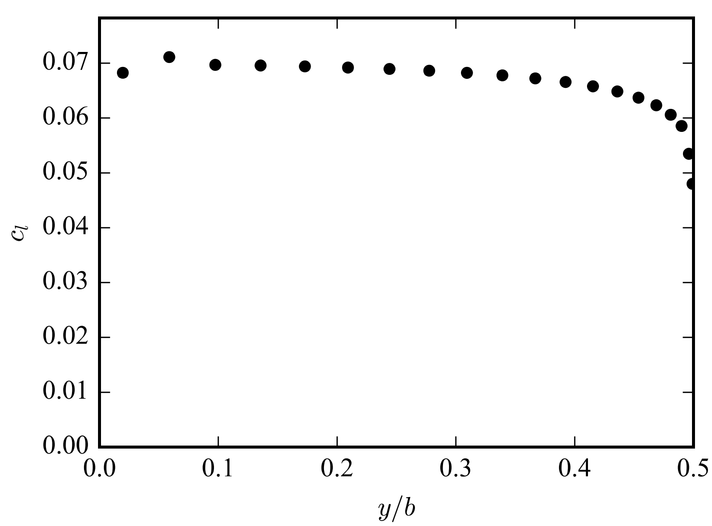


Figure H.1 Lift distribution with a 20×20 mesh for an elliptic wing with $A = 4$ and $\bar{t}_{max} = 4\%$.

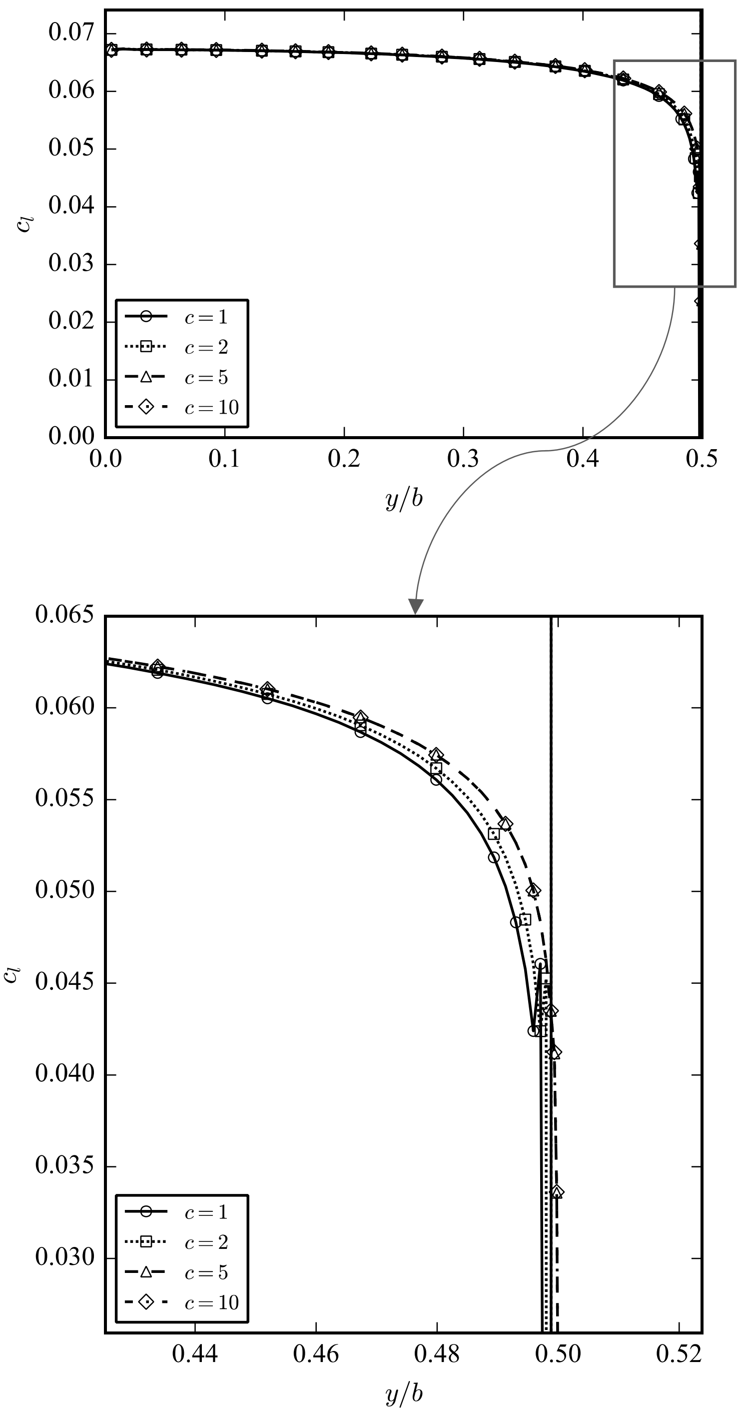


Figure H.2 Lift distributions as a function of average chord length with an 80×80 mesh for an elliptic wing with $A = 4$ and $\bar{t}_{max} = 4\%$.

The number of panels that can be included in a Panair model is extremely limited, driven by the state of computer hardware available at the time Panair was developed. In order to obtain the most accurate results possible, three models of successively increasing mesh sizes (20×20 , 40×40 , and 80×80) were used for each wing configuration, and the results were extrapolated using Richardson Extrapolation to estimate a fully-grid-resolved solution. A result set showing the extrapolated data for the same wing configuration as has already been described is given in Figure H.3. The extrapolated results are in good agreement with results from the 80×80 mesh.

The lifting line calculations presented in Chapters 4 and 5 represent wings having uniform, thin, symmetric sections with $a_0 = 2\pi$. Since the wing geometry provided to Panair must have a non-zero thickness, the thin sections were approximated by solving three result sets of successively decreasing airfoil thicknesses ($\bar{t}_{max} = 16\%$, 8% , and 4%) and extrapolating the results to a solution for $\bar{t}_{max} = 0\%$ using a linear least squares regression algorithm. A result set showing the thickness-extrapolation results for the elliptic wing described above is given in Figure H.4. Since the result sets for each airfoil thickness were themselves extrapolated from three Panair analyses of different grid sizes, a total of nine Panair analyses were needed to produce Figure H.4 and each of the Panair datasets used in the comparisons in Chapters 4 and 5.

Most of the low-aspect-ratio models discussed in Chapter 4 were developed for elliptic wings with straight quarter-chords, while others (namely Hauptman and Miloh [83] and Küchemann [73]) were developed for elliptic wings with straight mid-chords. In the interest of facilitating comparisons between these models, Panair was used to quantify the differences in wing lift coefficients and spanwise lift distributions between these two wing configurations. Figure H.5 shows wing lift coefficients computed using Panair for aspect ratios ranging from 1 to 8. Figure H.6 shows the spanwise lift distributions (scaled by the local chord ratio $c/\bar{c}$) computed using Panair for aspect ratios of 0.5, 2, and 8. From these results we conclude that the differences in lift are negligible between the two configurations, so that comparisons between the different low-aspect-ratio models discussed in Chapter 4 can be made without adjustment to the data.

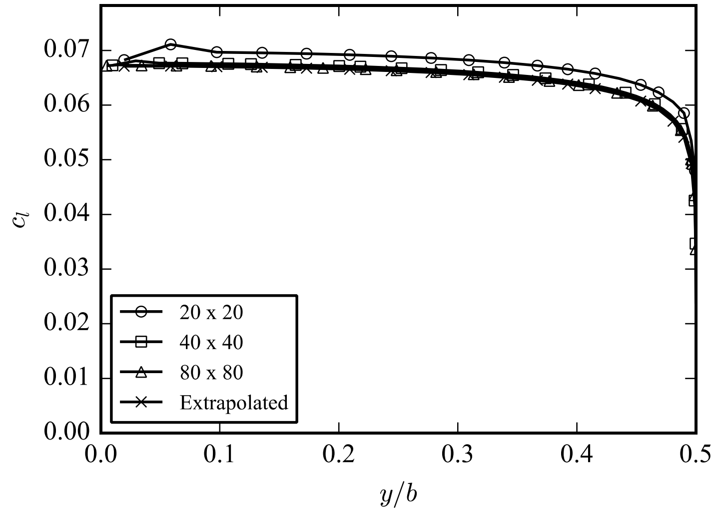


Figure H.3 Mesh refinement and extrapolated results computed for an elliptic wing with $A = 4$ and $\bar{t}_{\max} = 4\%$.

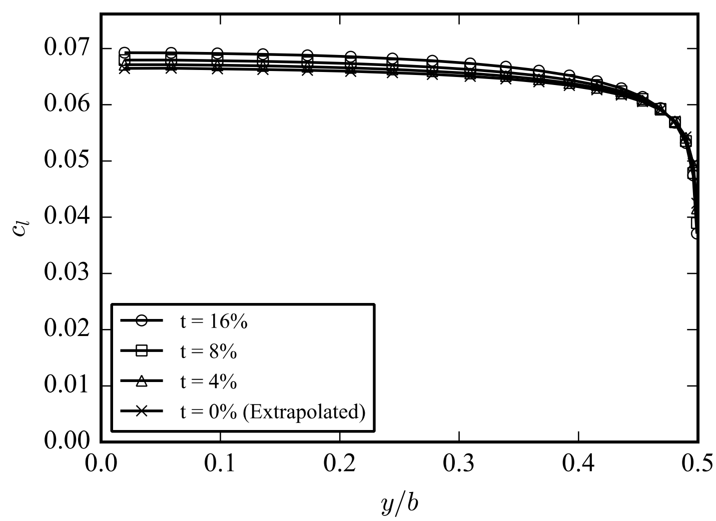


Figure H.4 Lift coefficient results computed for elliptic wings of different airfoil thicknesses with $A = 4$ and extrapolated results for $\bar{t}_{\max} = 0\%$.

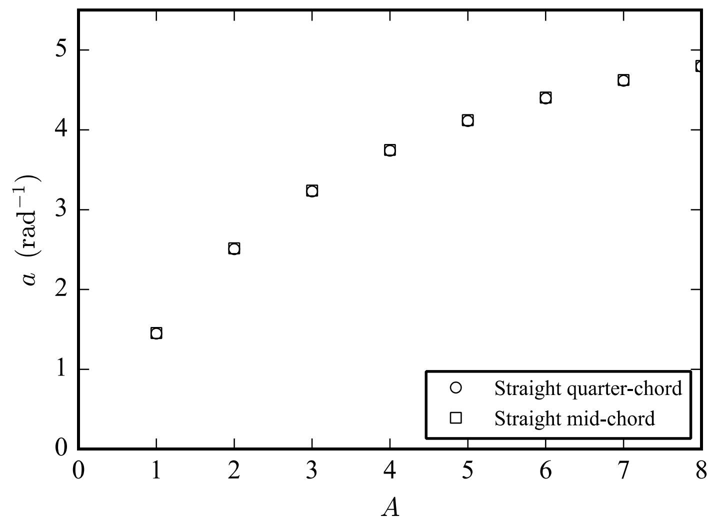


Figure H.5 Comparison of wing lift coefficients for wings with straight quarter-chord and straight mid-chord.

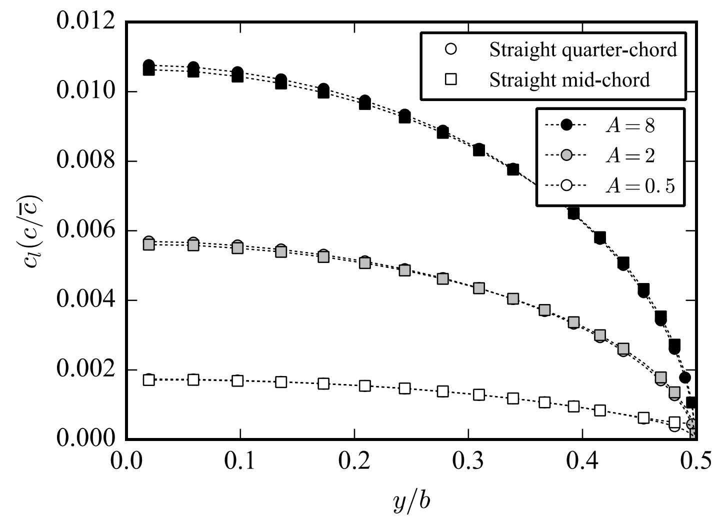


Figure H.6 Comparison of spanwise lift coefficients for wings with straight quarter-chord and straight mid-chord.

I MACHUP FUNCTIONS FOR WRITING PANAIR INPUT FILES

```
subroutine view_panair(t, json_command)
    type(plane_t), intent(in) :: t
    type(json_value), intent(in), pointer :: json_command

    real, allocatable, dimension(:,:) :: af_points
    character(100) :: filename, upper_network, lower_network, endcap_network
    integer :: i, iwing, af_datasize, symmetric
    integer :: ierror = 0

    integer :: endcap_npts
    real :: endcap_scale

    call myjson_get(json_command, 'endcap_npts', endcap_npts, 0)
    call myjson_get(json_command, 'endcap_scale', endcap_scale, 1.0)

    do i=1,size(airfoils)
        call af_create_geom_from_file(airfoils(i),DB_Airfoil)
    end do

    ! Open the file
    write(filename, '(A)') trim(adjustl(t%master_filename))//'_view.panair'
    open(unit = 10, File = trim(adjustl(filename)), action = 'write', iostat = ierror)

    ! Determine symmetry
    symmetric = 1
    do iwing = 1, t%nrealwings
        if(t%wings(iwing)%orig_side .eq. 'right' .or. t%wings(iwing)%orig_side .eq. 'left') then
            symmetric = 0
        end if
    end do

    ! Write Header information
    call view_write_panair_header(t, symmetric)

    do iwing=1,t%nrealwings !real wings
        ! If symmetric, only write out the right wings
        if(symmetric .eq. 1 .and. t%wings(iwing)%side .ne. 'right') then
            cycle
        end if

        ! Make sure all airfoils have the same number of points
        af_datasize = t%wings(iwing)%airfoils(1)%p%geom%datasize
        do i=1, t%wings(iwing)%nairfoils
            if(t%wings(iwing)%airfoils(i)%p%geom%datasize .ne. af_datasize) then
                write(*,*) 'All airfoils for wing ',t%wings(iwing)%name,' must have same number of nodes.'
                stop
            end if
        end do

        ! Allocate space for the points on an airfoil
        if (mod(af_datasize, 2) .eq. 0) af_datasize = af_datasize + 1  ! Must be odd number of points for Panair!
        allocate(af_points(af_datasize,3))

        ! Write the network header information
        call view_write_panair_network_header(t%wings(iwing))
```

```fortran

        ! Write the upper network
        write(upper_network, '(A, I0)') 'upper_', t%wings(iwing)%ID
        call view_write_panair_network(t%wings(iwing), af_points, trim(adjustl(upper_network)), af_datasize, af_datasize / 2 + 1)

        ! Write the lower network
        write(lower_network, '(A, I0)') 'lower_', t%wings(iwing)%ID
        call view_write_panair_network(t%wings(iwing), af_points, trim(adjustl(lower_network)), af_datasize / 2 + 1, 1)

        ! Write the endcap network
        if(t%wings(iwing)%chord_2 >= 0.0) then
            write(endcap_network, '(A, I0)') 'endcap_', t%wings(iwing)%ID
            call view_write_panair_endcap(t%wings(iwing), af_points, trim(adjustl(endcap_network)), endcap_npts, endcap_scale)
        end if

        ! Attach a wake to the trailing edge of the upper network
        call view_write_panair_wake(trim(adjustl(upper_network)))
        deallocate(af_points)
    end do

    write(10, "(A)") "$END"
    close(10)

end subroutine view_panair


subroutine view_write_panair_header(plane, symmetric)
    type(plane_t), intent(in) :: plane
    integer, intent(in) :: symmetric

    write(10, '(A)') '$TITLE'
    write(10, '(A)') plane%name
    write(10, '(A)') 'Generated by MachUp'
    write(10, '(A)') '$DATACHECK'
    write(10, '(A)') '=ndtchk'
    write(10, '(A)') '0.0'
    write(10, '(A)') '$SYMMETRIC'
    write(10, '(A, T11, A)') '=xzpln', 'xypln'
    write(10, '(I0, A, T11, A)') symmetric, '.0', '0.0'
    write(10, '(A)') '$MACH NUMBER'
    write(10, '(A)') '=amach'
    write(10, '(F10.6)') 0.0
    write(10, '(A)') '$CASES'
    write(10, '(A)') '=nacase'
    write(10, '(A)') '1.0'
    write(10, '(A)') '$ANGLES OF ATTACK'
    write(10, '(A)') '=alpc'
    write(10, '(F10.6)') 0.0
    write(10, '(A)') '=alpha(0)'
    write(10, '(F10.6)') plane%alpha * 180.0 / pi
    write(10, '(A)') '$YAW ANGLE'
    write(10, '(A)') '=betc'
    write(10, '(F10.6)') 0.0
    write(10, '(A)') '=beta(0)'
    write(10, '(F10.6)') plane%beta * 180.0 / pi
    write(10, '(A)') '$REFERENCE DATA'
    write(10, '(A, T11, A, T21, A)') '=xref', 'yref', 'zref'
    write(10, '(3F10.6)') 0.0, 0.0, 0.0
```

```
    write(10, '(A, T11, A, T21, A, T31, A)') '=sref', 'bref', 'cref', 'dref'
    write(10, '(4F10.6)') plane%Sr, plane%lat_r, plane%long_r, plane%lat_r
    write(10, '(A)') '$PRINTOUT CONTROL'
    write(10, '(A, T11, A, T21, A, T31, A, T41, A, T51, A)') '=isings', 'igeomp', 'isingp', 'icontp', 'ibconp', 'iedgep'
    write(10, '(A, T11, A, T21, A, T31, A, T41, A, T51, A)') '0.0', '0.0', '0.0', '0.0', '0.0', '0.0'
    write(10, '(A, T11, A, T21, A, T31, A, T41, A)') '=ipraic', 'nexdgn', 'ioutpr', 'ifmcpr', 'icostp'
    write(10, '(A, T11, A, T21, A, T31, A, T41, A)') '0.0', '0.0', '1.0', '0.0', '0.0'
end subroutine view_write_panair_header


subroutine view_write_panair_network_header(wi)
    type(wing_t), intent(in) :: wi

    integer :: nnetworks

    if(wi%chord_2 >= 0.0) then
        nnetworks = 3
    else
        nnetworks = 2
    end if

    ! Write header info
    write(10, "(A, I0)") "$POINTS for wing ", wi%ID
    write(10, "(A)") "=kn"  ! Number of networks in $POINTS block
    write(10, "(I0, T2, A)") nnetworks, ".0"
    write(10, "(A)") "=kt"  ! Boundary condition (1 = solid surface)
    write(10, "(A)") "1.0"

end subroutine view_write_panair_network_header


subroutine view_write_panair_network(wi, af_points, network, istart, iend)
    type(wing_t), intent(in) :: wi
    real, allocatable, dimension(:,:), intent(inout) :: af_points
    character(len=*), intent(in) :: network
    integer, intent(in) :: istart, iend

    integer :: isec, isec_start, isec_end, isec_inc
    type(section_t), pointer :: si

    write(10, "(A, T11, A)") "=nm", "nn"
    write(10, "(I0, T11, I0, T71, A)") istart - iend + 1, wi%nSec + 1, network

    if(wi%side .eq. "left") then
        isec_start = wi%nSec
        isec_end = 1
        isec_inc = -1
    else
        isec_start = 1
        isec_end = wi%nSec
        isec_inc = 1
    end if

    do isec = isec_start, isec_end, isec_inc
        si => wi%sec(isec)
        call view_create_local_airfoil_panair(wi, si, 1, af_points)
        call view_write_panair_points(istart, iend, af_points)
```

```fortran
    end do

    call view_create_local_airfoil_panair(wi, si, 2, af_points)
    call view_write_panair_points(istart, iend, af_points)

end subroutine view_write_panair_network


subroutine view_create_local_airfoil_panair(wi, si, sec_side, af_points)
    type(wing_t), intent(in) :: wi
    type(section_t), pointer, intent(in) :: si
    integer, intent(in) :: sec_side
    real, allocatable, dimension(:,:), intent(inout) :: af_points

    real :: percent, chord, RA
    integer :: i, mid
    integer :: af_datasize

    real :: a, b, c
    real, dimension(3) :: midpoint

    af_datasize = size(af_points, 1)
    if (mod(af_datasize, 2) .eq. 0) then
        write(*,*) "Allocated size of af_points must always be odd for Panair interface!"
        write(*,*) "If the number of points on an airfoil is even, allocate af_points to"
        write(*,*) "one more than this."
        stop
    end if

    if(sec_side .eq. 1) then
        percent = si%percent_1
    else if(sec_side .eq. 2) then
        percent = si%percent_2
    else
        percent = si%percent_c
    end if

    if(wi%chord_2 >= 0.0) then
        chord = wi%chord_1 + percent*(wi%chord_2 - wi%chord_1)
    else
        RA = 8.0 * wi%span / pi / wi%chord_1
        chord = 8.0 * wi%span / pi / RA * sqrt(1.0 - percent**2)
    end if

    if(sec_side .eq. 1) then
        call view_create_local_airfoil(si%af1_a, si%af1_b, wi%side, si%af_weight_1, chord, &
                & si%twist1, si%dihedral1, si%P1, af_points)
    else if(sec_side .eq. 2) then
        call view_create_local_airfoil(si%af2_a, si%af2_b, wi%side, si%af_weight_2, chord, &
                & si%twist2, si%dihedral2, si%P2, af_points)
    else
        call view_create_local_airfoil(si%afc_a, si%afc_b, wi%side, si%af_weight_c, chord, &
                & si%twist, si%dihedral, si%PC, af_points)
    end if

```

```fortran
    ! If the number of points on an airfoil is even, add an additional point at the leading edge
    if (mod(si%af1_a%geom%datasize, 2) .eq. 0) then
        ! Find the index of the second leading-edge point
        mid = si%af1_a%geom%datasize / 2 + 1

        ! Calculate the new midpoint between the first and second leading-edge points
        midpoint(:) = 0.5 * (af_points(mid - 1, :) + af_points(mid, :))

        ! Fit a parabola through three points at the leading edge
        call quadratic_fit(af_points(mid - 1 : mid + 1, 3:1:-2), a, b, c)
        if (.not. isnan(a) .and. .not. isnan(b) .and. .not. isnan(c)) then
            midpoint(1) = a * midpoint(3)**2 + b * midpoint(3) + c
        end if

        ! Shift the last half of the point array to the end
        af_points(si%af1_a%geom%datasize + 1 : mid + 1 : -1, :) = af_points(si%af1_a%geom%datasize : mid : -1, :)

        ! Place the new midpoint between the two leading-edge points
        af_points(mid, :) = midpoint(:)

    end if

    do i = 1, af_datasize
        af_points(i, 1) = -af_points(i, 1)
        af_points(i, 3) = -af_points(i, 3)
    end do

end subroutine view_create_local_airfoil_panair


subroutine view_write_panair_points(istart, iend, af_points)
    integer, intent(in) :: istart, iend
    real, dimension(:, :), intent(in) :: af_points

    integer :: ipt

    do ipt = istart, iend + 1, -2
        write(10, "(6F10.6)") af_points(ipt, :), af_points(ipt - 1, :)
    end do
    if (ipt == iend) then
        write(10, "(3F10.6)") af_points(ipt, :)
    end if
end subroutine view_write_panair_points


subroutine view_write_panair_endcap(wi, af_points, network, npts, rscale)
    type(wing_t), intent(in) :: wi
    real, allocatable, dimension(:, :), intent(inout) :: af_points
    character(len=*), intent(in) :: network
    integer, intent(in) :: npts
    real, intent(in) :: rscale

    real, allocatable, dimension(:, :) :: af_points_scaled, c
    real :: r
    real :: theta, theta_start, theta_end, dtheta
    integer :: i, j, af_mid, af_end

    ! Calculate the midpoint index
```

```
    af_end = size(af_points, 1)
    af_mid = (af_end + 1) / 2

    ! Write the network header info
    write(10, "(A, T11, A)") "=nm", "nn"
    write(10, "(I0, T11, I0, T71, A)") af_mid, npts + 2, network

    ! Get the airfoil points
    call view_create_local_airfoil_panair(wi, wi%sec(wi%nSec), 2, af_points)
    allocate(af_points_scaled(af_mid, 3))
    allocate(c(af_mid, 3))

    ! Set up theta
    if(wi%side .eq. 'left') then
        theta_start = 0.0
        theta_end = pi
    else
        theta_start = pi
        theta_end = 0.0
    end if
    dtheta = (theta_end - theta_start) / REAL(npts + 1)

    ! calculate the centerline of the airfoil (not quite camber line...)
    c(:, :) = 0.5 * (af_points(1:af_mid, :) + af_points(af_end:af_mid:-1, :))

    theta = theta_start
    do i = 1, npts + 2
        ! Scale the airfoil points
        do j = 1, af_mid
            r = 0.5 * (af_points(af_end - j + 1, 3) - af_points(j, 3))
            af_points_scaled(j, 1) = c(j, 1)
            af_points_scaled(j, 2) = c(j, 2) + rscale * r * sin(theta)
            af_points_scaled(j, 3) = c(j, 3) + r * cos(theta)
        end do

        call view_write_panair_points(af_mid, 1, af_points_scaled)

        theta = theta + dtheta
    end do

end subroutine view_write_panair_endcap


subroutine view_write_panair_wake(network)
    character(*), intent(in) :: network
    write(10, "(A)") "$TRAILING matchw=0"
    write(10, "(A)") "=kn"
    write(10, "(A)") "1.0"
    write(10, "(A, T11, A)") "=kt", "matchw"
    write(10, "(A, T11, A)") "18.0", "0.0"
    write(10, "(A, T11, A, T21, A, T31, A)") "=inat", "insd", "xwake", "twake"
    write(10, "(A, T11, A, T21, A, T31, A, T71, A)") network, "1.0", "10.0",
"0.0", "wake"
end subroutine view_write_panair_wake
```

J EXAMPLE PANAIR INPUT FILE

```
$TITLE
myairplane
Generated by MachUp
$DATACHECK
=ndtchk
0.0
$SYMMETRIC
=xzpln    xypln
1.0       0.0
$MACH NUMBER
=amach
  0.000000
$CASES
=nacase
1.0
$ANGLES OF ATTACK
=alpc
  0.000000
=alpha(0)
  1.000000
$YAW ANGLE
=betc
  0.000000
=beta(0)
  0.000000
$REFERENCE DATA
=xref     yref      zref
  0.000000  0.000000  0.000000
=sref     bref      cref      dref
  4.000000  1.000000  4.000000  1.000000
$PRINTOUT CONTROL
=isings   igeomp    isingp    icontp    ibconp    iedgep
0.0       0.0       0.0       0.0       0.0       0.0
=ipraic   nexdgn    ioutpr    ifmcpr    icostp
0.0       0.0       1.0       0.0       0.0
$POINTS for wing 1
=kn
2.0
=kt
1.0
=nm        nn
11         21                                                              upper_1
  0.954930  0.000000  0.000000  0.918311  0.000000  0.000377
  0.813166  0.000000  0.002748  0.652742  0.000000  0.007923
  0.456480  0.000000  0.014992  0.246982  0.000000  0.021670
  0.047048  0.000000  0.025288 -0.122724  0.000000  0.023915
 -0.245660  0.000000  0.017131 -0.310112  0.000000  0.006222
 -0.319908  0.000000 -0.000000
  0.951986  0.156918  0.000000  0.915480  0.156918  0.000376
  0.810659  0.156918  0.002739  0.650729  0.156918  0.007899
  0.455073  0.156918  0.014946  0.246220  0.156918  0.021603
  0.046903  0.156918  0.025210 -0.122346  0.156918  0.023841
 -0.244902  0.156918  0.017078 -0.309156  0.156918  0.006203
 -0.318922  0.156918 -0.000000
  0.943173  0.312869  0.000000  0.907005  0.312869  0.000372
  0.803155  0.312869  0.002714  0.644705  0.312869  0.007826
  0.450860  0.312869  0.014807  0.243941  0.312869  0.021403
  0.046469  0.312869  0.024977 -0.121213  0.312869  0.023620
 -0.242635  0.312869  0.016920 -0.306294  0.312869  0.006146
 -0.315969  0.312869 -0.000000
  0.928545  0.466891  0.000000  0.892938  0.466891  0.000367
```

0.790698	0.466891	0.002672	0.634706	0.466891	0.007704
0.443867	0.466891	0.014578	0.240158	0.466891	0.021071
0.045748	0.466891	0.024589	-0.119333	0.466891	0.023254
-0.238872	0.466891	0.016658	-0.301544	0.466891	0.006051
-0.311069	0.466891	-0.000000			
0.908192	0.618034	0.000000	0.873366	0.618034	0.000359
0.773367	0.618034	0.002613	0.620794	0.618034	0.007535
0.434138	0.618034	0.014258	0.234894	0.618034	0.020609
0.044745	0.618034	0.024050	-0.116718	0.618034	0.022744
-0.233636	0.618034	0.016293	-0.294934	0.618034	0.005918
-0.304250	0.618034	-0.000000			
0.882240	0.765367	0.000000	0.848409	0.765367	0.000348
0.751268	0.765367	0.002539	0.603055	0.765367	0.007320
0.421732	0.765367	0.013851	0.228181	0.765367	0.020020
0.043467	0.765367	0.023363	-0.113383	0.765367	0.022094
-0.226960	0.765367	0.015827	-0.286506	0.765367	0.005749
-0.295556	0.765367	-0.000000			
0.850849	0.907981	0.000000	0.818221	0.907981	0.000336
0.724536	0.907981	0.002448	0.581597	0.907981	0.007060
0.406727	0.907981	0.013358	0.220062	0.907981	0.019308
0.041920	0.907981	0.022532	-0.109348	0.907981	0.021308
-0.218884	0.907981	0.015264	-0.276312	0.907981	0.005544
-0.285040	0.907981	-0.000000			
0.814211	1.044997	0.000000	0.782989	1.044997	0.000322
0.693338	1.044997	0.002343	0.556554	1.044997	0.006756
0.389213	1.044997	0.012783	0.210587	1.044997	0.018476
0.040115	1.044997	0.021562	-0.104640	1.044997	0.020391
-0.209459	1.044997	0.014607	-0.264414	1.044997	0.005306
-0.272766	1.044997	-0.000000			
0.772554	1.175571	0.000000	0.742929	1.175571	0.000305
0.657865	1.175571	0.002223	0.528079	1.175571	0.006410
0.369300	1.175571	0.012129	0.199812	1.175571	0.017531
0.038063	1.175571	0.020459	-0.099286	1.175571	0.019348
-0.198743	1.175571	0.013859	-0.250886	1.175571	0.005034
-0.258811	1.175571	-0.000000			
0.726134	1.298896	0.000000	0.698289	1.298896	0.000287
0.618336	1.298896	0.002090	0.496349	1.298896	0.006025
0.347110	1.298896	0.011400	0.187806	1.298896	0.016478
0.035776	1.298896	0.019229	-0.093320	1.298896	0.018185
-0.186801	1.298896	0.013027	-0.235811	1.298896	0.004732
-0.243260	1.298896	-0.000000			
0.675237	1.414214	0.000000	0.649344	1.414214	0.000267
0.574995	1.414214	0.001943	0.461558	1.414214	0.005603
0.322780	1.414214	0.010601	0.174642	1.414214	0.015323
0.033268	1.414214	0.017881	-0.086779	1.414214	0.016910
-0.173708	1.414214	0.012114	-0.219282	1.414214	0.004400
-0.226209	1.414214	-0.000000			
0.620177	1.520812	0.000000	0.596395	1.520812	0.000245
0.528109	1.520812	0.001785	0.423922	1.520812	0.005146
0.296460	1.520812	0.009736	0.160402	1.520812	0.014073
0.030555	1.520812	0.016423	-0.079703	1.520812	0.015531
-0.159543	1.520812	0.011126	-0.201402	1.520812	0.004041
-0.207764	1.520812	-0.000000			
0.561294	1.618034	0.000000	0.539770	1.618034	0.000222
0.477967	1.618034	0.001615	0.383672	1.618034	0.004657
0.268312	1.618034	0.008812	0.145172	1.618034	0.012737
0.027654	1.618034	0.014864	-0.072136	1.618034	0.014057
-0.144395	1.618034	0.010069	-0.182279	1.618034	0.003657
-0.188037	1.618034	-0.000000			
0.498949	1.705280	0.000000	0.479816	1.705280	0.000197
0.424878	1.705280	0.001436	0.341057	1.705280	0.004140

```
  0.238510  1.705280  0.007833  0.129048  1.705280  0.011322
  0.024583  1.705280  0.013213 -0.064123  1.705280  0.012495
 -0.128357  1.705280  0.008951 -0.162033  1.705280  0.003251
 -0.167151  1.705280 -0.000000
  0.433529  1.782013  0.000000  0.416904  1.782013  0.000171
  0.369170  1.782013  0.001248  0.296339  1.782013  0.003597
  0.207238  1.782013  0.006806  0.112127  1.782013  0.009838
  0.021359  1.782013  0.011481 -0.055716  1.782013  0.010857
 -0.111527  1.782013  0.007777 -0.140788  1.782013  0.002825
 -0.145235  1.782013 -0.000000
  0.365436  1.847759  0.000000  0.351422  1.847759  0.000144
  0.311185  1.847759  0.001052  0.249793  1.847759  0.003032
  0.174687  1.847759  0.005737  0.094516  1.847759  0.008293
  0.018005  1.847759  0.009677 -0.046965  1.847759  0.009152
 -0.094010  1.847759  0.006556 -0.118675  1.847759  0.002381
 -0.122423  1.847759 -0.000000
  0.295089  1.902113  0.000000  0.283774  1.902113  0.000117
  0.251282  1.902113  0.000849  0.201708  1.902113  0.002448
  0.141060  1.902113  0.004633  0.076322  1.902113  0.006696
  0.014539  1.902113  0.007814 -0.037924  1.902113  0.007390
 -0.075913  1.902113  0.005294 -0.095830  1.902113  0.001923
 -0.098857  1.902113 -0.000000
  0.222924  1.944740  0.000000  0.214375  1.944740  0.000088
  0.189830  1.944740  0.000641  0.152380  1.944740  0.001850
  0.106563  1.944740  0.003500  0.057657  1.944740  0.005059
  0.010983  1.944740  0.005903 -0.028649  1.944740  0.005583
 -0.057348  1.944740  0.003999 -0.072394  1.944740  0.001453
 -0.074681  1.944740 -0.000000
  0.149384  1.975377  0.000000  0.143655  1.975377  0.000059
  0.127207  1.975377  0.000430  0.102111  1.975377  0.001239
  0.071409  1.975377  0.002345  0.038636  1.975377  0.003390
  0.007360  1.975377  0.003956 -0.019198  1.975377  0.003741
 -0.038430  1.975377  0.002680 -0.048512  1.975377  0.000973
 -0.050045  1.975377 -0.000000
  0.074923  1.993835  0.000000  0.072050  1.993835  0.000030
  0.063800  1.993835  0.000216  0.051214  1.993835  0.000622
  0.035815  1.993835  0.001176  0.019378  1.993835  0.001700
  0.003691  1.993835  0.001984 -0.009629  1.993835  0.001876
 -0.019274  1.993835  0.001344 -0.024331  1.993835  0.000488
 -0.025100  1.993835 -0.000000
 -0.000000  2.000000 -0.000000 -0.000000  2.000000 -0.000000
 -0.000000  2.000000 -0.000000 -0.000000  2.000000 -0.000000
 -0.000000  2.000000 -0.000000 -0.000000  2.000000 -0.000000
 -0.000000  2.000000 -0.000000 -0.000000  2.000000 -0.000000
 -0.000000  2.000000 -0.000000 -0.000000  2.000000 -0.000000
 -0.000000  2.000000 -0.000000
=nm        nn
11         21                                                       lower_1
 -0.319908  0.000000 -0.000000 -0.310112  0.000000 -0.006223
 -0.245660  0.000000 -0.017131 -0.122724  0.000000 -0.023915
  0.047048  0.000000 -0.025288  0.246982  0.000000 -0.021670
  0.456480  0.000000 -0.014992  0.652742  0.000000 -0.007923
  0.813166  0.000000 -0.002748  0.918311  0.000000 -0.000377
  0.954930  0.000000 -0.000000
 -0.318922  0.156918 -0.000000 -0.309156  0.156918 -0.006203
 -0.244902  0.156918 -0.017078 -0.122346  0.156918 -0.023841
  0.046903  0.156918 -0.025210  0.246220  0.156918 -0.021603
  0.455073  0.156918 -0.014946  0.650730  0.156918 -0.007899
  0.810659  0.156918 -0.002739  0.915480  0.156918 -0.000376
  0.951986  0.156918 -0.000000
 -0.315969  0.312869 -0.000000 -0.306294  0.312869 -0.006146
```

-0.242635	0.312869	-0.016920	-0.121213	0.312869	-0.023620
0.046469	0.312869	-0.024977	0.243941	0.312869	-0.021403
0.450860	0.312869	-0.014807	0.644705	0.312869	-0.007826
0.803155	0.312869	-0.002714	0.907005	0.312869	-0.000372
0.943173	0.312869	-0.000000			
-0.311069	0.466891	-0.000000	-0.301544	0.466891	-0.006051
-0.238872	0.466891	-0.016658	-0.119333	0.466891	-0.023254
0.045748	0.466891	-0.024589	0.240158	0.466891	-0.021071
0.443867	0.466891	-0.014578	0.634706	0.466891	-0.007704
0.790698	0.466891	-0.002672	0.892938	0.466891	-0.000367
0.928545	0.466891	-0.000000			
-0.304250	0.618034	-0.000000	-0.294934	0.618034	-0.005918
-0.233636	0.618034	-0.016293	-0.116718	0.618034	-0.022744
0.044745	0.618034	-0.024050	0.234894	0.618034	-0.020609
0.434138	0.618034	-0.014258	0.620794	0.618034	-0.007535
0.773367	0.618034	-0.002613	0.873366	0.618034	-0.000359
0.908192	0.618034	-0.000000			
-0.295556	0.765367	-0.000000	-0.286506	0.765367	-0.005749
-0.226960	0.765367	-0.015827	-0.113382	0.765367	-0.022094
0.043467	0.765367	-0.023363	0.228181	0.765367	-0.020020
0.421732	0.765367	-0.013851	0.603055	0.765367	-0.007320
0.751268	0.765367	-0.002539	0.848409	0.765367	-0.000348
0.882240	0.765367	-0.000000			
-0.285040	0.907981	-0.000000	-0.276312	0.907981	-0.005544
-0.218884	0.907981	-0.015264	-0.109348	0.907981	-0.021308
0.041920	0.907981	-0.022532	0.220062	0.907981	-0.019308
0.406727	0.907981	-0.013358	0.581597	0.907981	-0.007060
0.724536	0.907981	-0.002448	0.818221	0.907981	-0.000336
0.850849	0.907981	-0.000000			
-0.272766	1.044997	-0.000000	-0.264414	1.044997	-0.005306
-0.209459	1.044997	-0.014607	-0.104640	1.044997	-0.020391
0.040115	1.044997	-0.021562	0.210587	1.044997	-0.018476
0.389213	1.044997	-0.012783	0.556554	1.044997	-0.006756
0.693338	1.044997	-0.002343	0.782989	1.044997	-0.000322
0.814211	1.044997	-0.000000			
-0.258811	1.175571	-0.000000	-0.250886	1.175571	-0.005034
-0.198743	1.175571	-0.013859	-0.099286	1.175571	-0.019348
0.038063	1.175571	-0.020459	0.199812	1.175571	-0.017531
0.369300	1.175571	-0.012129	0.528079	1.175571	-0.006410
0.657865	1.175571	-0.002223	0.742929	1.175571	-0.000305
0.772554	1.175571	-0.000000			
-0.243260	1.298896	-0.000000	-0.235811	1.298896	-0.004732
-0.186801	1.298896	-0.013027	-0.093320	1.298896	-0.018185
0.035776	1.298896	-0.019229	0.187806	1.298896	-0.016478
0.347110	1.298896	-0.011400	0.496349	1.298896	-0.006025
0.618336	1.298896	-0.002090	0.698289	1.298896	-0.000287
0.726134	1.298896	-0.000000			
-0.226209	1.414214	-0.000000	-0.219282	1.414214	-0.004400
-0.173708	1.414214	-0.012114	-0.086779	1.414214	-0.016910
0.033268	1.414214	-0.017881	0.174643	1.414214	-0.015323
0.322780	1.414214	-0.010601	0.461558	1.414214	-0.005603
0.574995	1.414214	-0.001943	0.649344	1.414214	-0.000267
0.675237	1.414214	-0.000000			
-0.207764	1.520812	-0.000000	-0.201402	1.520812	-0.004041
-0.159543	1.520812	-0.011126	-0.079703	1.520812	-0.015531
0.030555	1.520812	-0.016423	0.160402	1.520812	-0.014073
0.296460	1.520812	-0.009736	0.423922	1.520812	-0.005146
0.528109	1.520812	-0.001785	0.596395	1.520812	-0.000245
0.620177	1.520812	-0.000000			
-0.188037	1.618034	-0.000000	-0.182279	1.618034	-0.003657
-0.144395	1.618034	-0.010069	-0.072136	1.618034	-0.014057

```
      0.027654  1.618034 -0.014864  0.145172  1.618034 -0.012737
      0.268312  1.618034 -0.008812  0.383672  1.618034 -0.004657
      0.477967  1.618034 -0.001615  0.539770  1.618034 -0.000222
      0.561294  1.618034 -0.000000
     -0.167151  1.705280 -0.000000 -0.162033  1.705280 -0.003251
     -0.128357  1.705280 -0.008951 -0.064123  1.705280 -0.012495
      0.024583  1.705280 -0.013213  0.129048  1.705280 -0.011322
      0.238510  1.705280 -0.007833  0.341057  1.705280 -0.004140
      0.424878  1.705280 -0.001436  0.479816  1.705280 -0.000197
      0.498949  1.705280 -0.000000
     -0.145235  1.782013 -0.000000 -0.140788  1.782013 -0.002825
     -0.111527  1.782013 -0.007777 -0.055716  1.782013 -0.010857
      0.021359  1.782013 -0.011481  0.112127  1.782013 -0.009838
      0.207238  1.782013 -0.006806  0.296339  1.782013 -0.003597
      0.369170  1.782013 -0.001248  0.416904  1.782013 -0.000171
      0.433529  1.782013 -0.000000
     -0.122423  1.847759 -0.000000 -0.118675  1.847759 -0.002381
     -0.094010  1.847759 -0.006556 -0.046965  1.847759 -0.009152
      0.018005  1.847759 -0.009677  0.094516  1.847759 -0.008293
      0.174687  1.847759 -0.005737  0.249793  1.847759 -0.003032
      0.311185  1.847759 -0.001052  0.351422  1.847759 -0.000144
      0.365436  1.847759 -0.000000
     -0.098857  1.902113 -0.000000 -0.095830  1.902113 -0.001923
     -0.075913  1.902113 -0.005294 -0.037924  1.902113 -0.007390
      0.014539  1.902113 -0.007814  0.076322  1.902113 -0.006696
      0.141060  1.902113 -0.004633  0.201708  1.902113 -0.002448
      0.251282  1.902113 -0.000849  0.283774  1.902113 -0.000117
      0.295089  1.902113 -0.000000
     -0.074681  1.944740 -0.000000 -0.072394  1.944740 -0.001453
     -0.057348  1.944740 -0.003999 -0.028649  1.944740 -0.005583
      0.010983  1.944740 -0.005903  0.057657  1.944740 -0.005059
      0.106563  1.944740 -0.003500  0.152380  1.944740 -0.001850
      0.189830  1.944740 -0.000641  0.214375  1.944740 -0.000088
      0.222924  1.944740 -0.000000
     -0.050045  1.975377 -0.000000 -0.048512  1.975377 -0.000973
     -0.038430  1.975377 -0.002680 -0.019198  1.975377 -0.003741
      0.007360  1.975377 -0.003956  0.038636  1.975377 -0.003390
      0.071409  1.975377 -0.002345  0.102111  1.975377 -0.001239
      0.127207  1.975377 -0.000430  0.143655  1.975377 -0.000059
      0.149384  1.975377 -0.000000
     -0.025100  1.993835 -0.000000 -0.024331  1.993835 -0.000488
     -0.019274  1.993835 -0.001344 -0.009629  1.993835 -0.001876
      0.003691  1.993835 -0.001984  0.019378  1.993835 -0.001700
      0.035815  1.993835 -0.001176  0.051214  1.993835 -0.000622
      0.063800  1.993835 -0.000216  0.072050  1.993835 -0.000030
      0.074923  1.993835 -0.000000
     -0.000000  2.000000 -0.000000 -0.000000  2.000000 -0.000000
     -0.000000  2.000000 -0.000000 -0.000000  2.000000 -0.000000
     -0.000000  2.000000 -0.000000 -0.000000  2.000000 -0.000000
     -0.000000  2.000000 -0.000000 -0.000000  2.000000 -0.000000
     -0.000000  2.000000 -0.000000 -0.000000  2.000000 -0.000000
     -0.000000  2.000000 -0.000000
    $TRAILING matchw=0
    =kn
    1.0
    =kt       matchw
    18.0      0.0
    =inat     insd      xwake     twake
    upper_1   1.0       10.0      0.0                                     wake_1
$END
```

K PRALINES SOURCE CODE

K.1 Pralines Main Program (`main.f90`)

```
program PrandtlsLiftingLine
    use LiftingLineInterface

    implicit none

    !Begin execution
    call BeginLiftingLineInterface()

end program
```

K.2 Interface Module (`liftinglineinterface.f90`)

```
module LiftingLineInterface
    use class_Planform
    use LiftingLineSetters
    use LiftingLineSolver
    use LiftingLineOutput
    use LiftingLineSolver_Test

    implicit none

contains
    subroutine BeginLiftingLineInterface()
        type(Planform) :: pf
        character*2 :: inp = 'A'

        call InitPlanform(pf)

        do while(inp /= 'Q')
            inp = PlanformParamters(pf)
            if (inp == 'A') then
                call ComputeCMatrixAndCoefficients(pf)
                call OutputPlanform(pf)
                do while(inp /= 'Q' .and. inp /= 'B')
                    inp = OperatingConditions(pf)
                    if (inp /= 'Q') then
                        call UpdateOperatingConditions(pf, inp)
                    end if
                end do
            else if (inp /= 'Q') then
                call UpdatePlanformParameters(pf, inp)
            end if
        end do
    end subroutine BeginLiftingLineInterface

    character*2 function PlanformParamters(pf) result(inp)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        ! Clear the screen and output the header
        call system('cls')
        call OutputHeader()

```

```
        ! Display options to user
        write(6, '(28x, a)') "Planform Design Menu"
        write(6, *)
        write(6, '(a)') "Select from the following menu options:"
        write(6, *)

        ! Wing parameters
        write(6, '(2x, a)') "Wing Parameters:"

        msg = "WT - Edit wing type"
        call DisplayMessageWithTextDefault(msg, GetWingType(pf), 4)

        msg = "N  - Edit number of nodes per semispan"
        call DisplayMessageWithIntegerDefault(msg, (pf%NNodes + 1) / 2, 4)

        msg = "RA - Edit aspect ratio"
        call DisplayMessageWithRealDefault(msg, pf%AspectRatio, 4)

        if (pf%WingType == Tapered) then
            msg = "RT - Edit taper ratio"
            call DisplayMessageWithRealDefault(msg, pf%TaperRatio, 4)
        end if

        msg = "S  - Edit section lift slope"
        call DisplayMessageWithRealDefault(msg, pf%SectionLiftSlope, 4)

        if (pf%WingType == Combination) then
            msg = "TZ - Edit z/b at the transition from tapered to elliptic"
            call DisplayMessageWithRealDefault(msg, pf%TransitionPoint, 4)

            msg = "TC - Edit c/croot at the transition from tapered to elliptic"
            call DisplayMessageWithRealDefault(msg, pf%TransitionChord, 4)
        end if

        if (pf%WingType /= Elliptic) then
            msg = "WD - Toggle washout distribution type"
            call DisplayMessageWithTextDefault(msg, GetWashoutDistributionType(pf), 4)
        end if

        msg = "LC - Edit low-aspect-ratio correction method"
        call DisplayMessageWithTextDefault(msg, GetLowAspectRatioMethod(pf), 4)

        ! Aileron parameters
        write(6, *)
        write(6, '(2x, a)') "Aileron Parameters:"

        msg = "ZR - Edit z/b of aileron root"
        call DisplayMessageWithRealDefault(msg, pf%AileronRoot, 4)

        msg = "ZT - Edit z/b of aileron tip"
        call DisplayMessageWithRealDefault(msg, pf%AileronTip, 4)

        msg = "PH - Make hinge line parallel with quarter-chord line?"
        call DisplayMessageWithLogicalDefault(msg, pf%ParallelHingeLine, 4)

        msg = "CR - Edit cf/c of aileron root"
        call DisplayMessageWithRealDefault(msg, pf%FlapFractionRoot, 4)

        msg = "CT - Edit cf/c of aileron tip"
        call DisplayMessageWithRealDefault(msg, pf%FlapFractionTip, 4)
```

```

        msg = "HE - Edit aileron hinge efficiency"
        call DisplayMessageWithRealDefault(msg, pf%HingeEfficiency, 4)

        ! Output and Plotting options
        write(6, *)
        write(6, '(2x, a)') "Output and Plotting Options:"
        msg = "C  - Output C matrix and Fourier Coefficients?"
        call DisplayMessageWithLogicalDefault(msg, pf%OutputMatrices, 4)

        msg = "F  - Edit output file name"
        call DisplayMessageWithTextDefault(msg, pf%FileName, 4)

        write(6, '(4x, a)') "PP - Plot planform in ES-Plot"

        ! Main Execution commands
        write(6, *)
        write(6, '(2x, a)') "A - Advance to Operating Conditions Menu"
        write(6, '(2x, a)') "T - Test solver against Problem 1.34b solution"
        write(6, '(2x, a)') "Q - Quit"

        write(6, *)
        write(6, '(a)') "Your selection: "

        inp = GetCharacterInput("  ")
        write(6, *)
    end function PlanformParamters

    character*2 function OperatingConditions(pf) result(inp)
        type(Planform), intent(inout) :: pf

        integer :: i
        character*80 :: msg

        ! Clear the screen and output the header
        call system('cls')
        call OutputHeader()

        ! Output the Planform summary
        call OutputPlanformSummary(6, pf)
        call OutputOperatingConditions(6, pf)
        call OutputFlightCoefficients(6, pf)
        write(6, '(80a)') ("*", i=1,80)

        ! Display options to user
        write(6, '(28x, a)') "Operating Conditions Menu"
        write(6, *)
        write(6, '(a)') "Select from the following menu options:"

        ! Operating Conditions
        write(6, *)
        write(6, '(2x, a)') "Operating Conditions:"

        msg = "AA - Edit root aerodynamic angle of attack"
        call DisplayMessageWithAngleDefault(msg, pf%AngleOfAttack, 4)

        msg = "CL - Edit coefficient of lift"
        call DisplayMessageWithRealDefault(msg, pf%LiftCoefficient, 4)

        msg = "OW - Use optimum total washout"
        call DisplayMessageWithLogicalDefault(msg, pf%UseOptimumWashout, 4)
```

```
        msg = "W  - Edit total amount of washout"
        call DisplayMessageWithAngleDefault(msg, pf%Washout, 4)

        msg = "AD - Edit aileron deflection"
        call DisplayMessageWithAngleDefault(msg, pf%AileronDeflection, 4)

        msg = "SR - Use steady dimensionless rolling rate"
        call DisplayMessageWithLogicalDefault(msg, pf%UseSteadyRollingRate, 4)

        msg = "R  - Edit dimensionless rolling rate"
        call DisplayMessageWithRealDefault(msg, pf%RollingRate, 4)

        ! Plotting options
        write(6, *)
        write(6, '(2x, a)') "Plotting Options:"
        write(6, '(4x, a)') "PP - Plot Planform in ES-Plot"
        write(6, '(4x, a)') "PW - Plot Dimensionless Washout Distribution in ES-Plot"
        write(6, '(4x, a)') "PL - Plot Section Lift Distribution in ES-Plot"
        write(6, '(4x, a)') "WL - Write Section Lift Distribution to 'liftdistribution.dat'"
        write(6, '(4x, a)') "PN - Plot Normalized Section Lift Coefficient in ES-Plot"
        write(6, '(4x, a)') "WN - Write Normalized Section Lift Coefficient in ES-Plot"

        ! Main Execution commands
        write(6, *)
        msg = "S - Save Flight coefficients to output file"
        call DisplayMessageWithTextDefault(msg, pf%FileName, 2)

        write(6, '(2x, a)') "B - Back to Planform Design Menu"
        write(6, '(2x, a)') "Q - Quit"

        write(6, *)
        write(6, '(a)') "Your selection: "

        inp = GetCharacterInput("  ")
        write(6, *)
    end function OperatingConditions

    subroutine UpdatePlanformParameters(pf, input)
        type(Planform), intent(inout) :: pf
        character*2, intent(in) :: input

        ! Process input command
        ! Wing parameters
        if (input == 'WT') then
            call EditWingType(pf)
        else if (input == 'N') then
            call EditNNodes(pf)
        else if (input == 'RA') then
            call EditAspectRatio(pf)
        else if (input == 'RT' .and. pf%WingType == Tapered) then
            call EditTaperRatio(pf)
        else if (input == 'S') then
            call EditLiftSlope(pf)
        else if (input == 'TZ' .and. pf%WingType == Combination) then
            call EditTransitionPoint(pf)
        else if (input == 'TC' .and. pf%WingType == Combination) then
```

```
            call EditTransitionChord(pf)
        else if (input == 'WD' .and. pf%WingType /= Elliptic) then
            call EditWashoutDistribution(pf)
        else if (input == 'LC') then
            call EditLowAspectRatioCorrectionMethod(pf)

        ! Aileron parameters
        else if (input == 'ZR') then
            call EditAileronRoot(pf)
        else if (input == 'ZT') then
            call EditAileronTip(pf)
        else if (input == 'PH') then
            call ToggleParallelHinge(pf)
        else if (input == 'CR') then
            call EditFlapFractionRoot(pf)
        else if (input == 'CT') then
            call EditFlapFractionTip(pf)
        else if (input == 'HE') then
            call EditHingeEfficiency(pf)

        ! Output options
        else if (input == 'C') then
            pf%OutputMatrices = .not. pf%OutputMatrices
        else if (input == 'F') then
            call EditFileName(pf)
        else if (input == 'PP') then
            call PlotPlanform(pf)

        ! Testing options
        else if (input == 'T') then
            call TestLiftingLineSolver()
        end if
    end subroutine UpdatePlanformParameters

    subroutine UpdateOperatingConditions(pf, input)
        type(Planform), intent(inout) :: pf
        character*2, intent(in) :: input

        ! Operating Conditions
        if (input == 'AA') then
            call EditAngleOfAttack(pf)
        else if (input == 'CL') then
            call EditLiftCoefficient(pf)
        else if (input == 'OW') then
            call ToggleUseOptimumWashout(pf)
        else if (input == 'W') then
            call EditWashout(pf)
        else if (input == 'AD') then
            call EditAileronDeflection(pf)
        else if (input == 'SR') then
            call ToggleUseSteadyRollingRate(pf)
        else if (input == 'R') then
            call EditRollingRate(pf)

        ! Output and Plotting options
        else if (input == 'PP') then
            call PlotPlanform(pf)
        else if (input == 'PW') then
            call PlotWashout(pf)
        else if (input == 'PL') then
            call PlotSectionLiftDistribution(pf)
```

```
        else if (input == 'WL') then
            call WriteSectionLiftDistribution(pf)
        else if (input == 'PN') then
            call PlotNormalizedLiftCoefficient(pf)
        else if (input == 'WN') then
            call WriteNormalizedLiftCoefficient(pf)
        else if (input == 'S') then
            call OutputFlightConditions(pf)
        end if

        call ComputeFlightConditions(pf)
    end subroutine UpdateOperatingConditions

    subroutine EditWingType(pf)
        type(Planform), intent(inout) :: pf

        logical :: cont
        character*2 :: inp

        write(6, *)
        write(6, '(a)') "Select from the following wing type options:"
        write(6, '(2x, a)') "T - Tapered"
        write(6, '(2x, a)') "E - Elliptic"
        write(6, '(2x, a)') "C - Combination (Tapered with elliptic tip)"
        write(6, *)
        write(6, '(a)') "Your selection: "

        cont = .true.
        do while(cont)
            inp = GetCharacterInput("  ")
            write(6, *)

            if (inp == "T") then
                call SetWingType(pf, Tapered)
                cont = .false.
            else if (inp == "E") then
                call SetWingType(pf, Elliptic)
                cont = .false.
            else if (inp == "C") then
                call SetWingType(pf, Combination)
                cont = .false.
            else
                write(6, '(a)') "Invalid input, please make a selection from the above menu."
            end if
        end do
    end subroutine EditWingType

    subroutine EditTransitionPoint(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg
        real*8 :: tp_old
        logical :: isValid

        tp_old = pf%TransitionPoint

        msg = "Enter z/b at the transition point from tapered to elliptic"
        call DisplayMessageWithRealDefault(msg, pf%TransitionPoint, 0)
        call SetTransitionPoint(pf, GetRealInput(0.0d0, 0.5d0, pf%TransitionPoint))
```

```fortran

        isValid = AreCombinationWingCoefficientsValid(pf)
        do while(.not. isValid)
            call SetTransitionPoint(pf, tp_old)
            write(6, *)
            write(6, '(a)') "The input provided results in invalid ellipse coefficients."
            write(6, '(a)') "Try a new value or press <ENTER> to accept default."

            call SetTransitionPoint(pf, GetRealInput(0.0d0, 0.5d0, pf%TransitionPoint))
            isValid = AreCombinationWingCoefficientsValid(pf)
        end do
    end subroutine EditTransitionPoint

    subroutine EditTransitionChord(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg
        real*8 :: tc_old
        logical :: isValid

        tc_old = pf%TransitionChord

        msg = "Enter c/croot at the transition point from tapered to elliptic"
        call DisplayMessageWithRealDefault(msg, pf%TransitionChord, 0)
        call SetTransitionChord(pf, GetRealInput(0.0d0, 10.0d0, pf%TransitionChord))

        isValid = AreCombinationWingCoefficientsValid(pf)
        do while(.not. isValid)
            call SetTransitionChord(pf, tc_old)
            write(6, *)
            write(6, '(a)') "The input provided results in invalid ellipse coefficients."
            write(6, '(a)') "Try a new value or press <ENTER> to accept default."

            call SetTransitionChord(pf, GetRealInput(0.0d0, 2.0d0, pf%TransitionChord))
            isValid = AreCombinationWingCoefficientsValid(pf)
        end do
    end subroutine EditTransitionChord

    subroutine EditWashoutDistribution(pf)
        type(Planform), intent(inout) :: pf

        if (pf%WashoutDistribution == Linear) then
            call SetWashoutDistribution(pf, Optimum)
        else
            call SetWashoutDistribution(pf, Linear)
        end if
    end subroutine EditWashoutDistribution

    subroutine EditLowAspectRatioCorrectionMethod(pf)
        type(Planform), intent(inout) :: pf

        logical :: cont
        character*2 :: inp

```

```
        write(6, *)
        write(6, '(a)') "Select from the following low-aspect-ratio correction methods:"
        write(6, '(2x, a)') "C - Classical Lifting Line Theory (no correction)"
        write(6, '(2x, a)') "H - Hodson"
        write(6, '(2x, a)') "M - Modified Slender Wing"
        write(6, '(2x, a)') "K - Kuchemann"
        write(6, *)
        write(6, '(a)') "Your selection: "

        cont = .true.
        do while(cont)
            inp = GetCharacterInput("  ")
            write(6, *)

            if (inp == "C") then
                call SetLowAspectRatioMethod(pf, Classical)
                cont = .false.
            else if (inp == "H") then
                call SetLowAspectRatioMethod(pf, Hodson)
                cont = .false.
            else if (inp == "M") then
                call SetLowAspectRatioMethod(pf, ModifiedSlender)
                cont = .false.
            else if (inp == "K") then
                call SetLowAspectRatioMethod(pf, Kuchemann)
                cont = .false.
            else
                write(6, '(a)') "Invalid input, please make a selection from the above menu."
            end if
        end do
    end subroutine

    subroutine EditNNodes(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg
        integer :: npss
        character*80 :: int_str

        npss = (pf%NNodes + 1) / 2

        msg = "Enter number of nodes per semispan or press <ENTER> to accept default"
        call DisplayMessageWithIntegerDefault(msg, npss, 0)
        call SetNNodes(pf, GetIntInput(4, 1000, npss))
    end subroutine EditNNodes

    subroutine EditAspectRatio(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)
        msg = "Enter new aspect ratio or press <ENTER> to accept default"
        call DisplayMessageWithRealDefault(msg, pf%AspectRatio, 0)
        call SetAspectRatio(pf, GetRealInput(1.0d-12, 100.0d0, pf%AspectRatio))
    end subroutine EditAspectRatio

    subroutine EditTaperRatio(pf)
```

```
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)
        msg = "Enter new taper ratio or press <ENTER> to accept default"
        call DisplayMessageWithRealDefault(msg, pf%TaperRatio, 0)
        call SetTaperRatio(pf, GetRealInput(0.0d0, 100.0d0, pf%TaperRatio))
    end subroutine EditTaperRatio

    subroutine EditLiftSlope(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)
        msg = "Enter new section lift slope or press <ENTER> to accept default"
        call DisplayMessageWithRealDefault(msg, pf%SectionLiftSlope, 0)
        call SetSectionLiftSlope(pf, GetRealInput(-100.0d0 * pi, 100.0d0 * pi, &
            & pf%SectionLiftSlope))
    end subroutine EditLiftSlope

    subroutine EditAileronRoot(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)
        msg = "Enter new z/b for aileron root or press <ENTER> to accept default"
        call DisplayMessageWithRealDefault(msg, pf%AileronRoot, 0)
        call SetAileronRoot(pf, GetRealInput(0.0d0, pf%AileronTip, pf%AileronRoot))
    end subroutine EditAileronRoot

    subroutine EditAileronTip(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)
        msg = "Enter new z/b for aileron tip or press <ENTER> to accept default"
        call DisplayMessageWithRealDefault(msg, pf%AileronTip, 0)
        call SetAileronTip(pf, GetRealInput(pf%AileronRoot, 0.5d0, pf%AileronTip))
    end subroutine EditAileronTip

    subroutine EditFlapFractionRoot(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)
        if (pf%ParallelHingeLine) then
            write(6, '(a)') "NOTE: Hinge is no longer constrained to be parallel with quarter-chord line."
        end if

        msg = "Enter new cf/c at aileron root or press <ENTER> to accept default"
        call DisplayMessageWithRealDefault(msg, pf%DesiredFlapFractionRoot, 0)
```

```
        call SetFlapFractionRoot(pf, GetRealInput(0.0d0, 1.0d0, pf%DesiredFlapFractionRoot))
    end subroutine EditFlapFractionRoot

    subroutine EditFlapFractionTip(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)
        msg = "Enter new cf/c at aileron tip or press <ENTER> to accept default"
        call DisplayMessageWithRealDefault(msg, pf%FlapFractionTip, 0)
        call SetFlapFractionTip(pf, GetRealInput(0.0d0, 1.0d0, pf%FlapFractionTip))
    end subroutine EditFlapFractionTip

    subroutine ToggleParallelHinge(pf)
        type(Planform), intent(inout) :: pf

        if (pf%ParallelHingeLine) then
            call SetFlapFractionRoot(pf, pf%DesiredFlapFractionRoot)
        else
            call SetParallelHingeLine(pf)
        end if
    end subroutine ToggleParallelHinge

    subroutine EditHingeEfficiency(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)
        msg = "Enter aileron hinge efficiency or press <ENTER> to accept default"
        call DisplayMessageWithRealDefault(msg, pf%HingeEfficiency, 0)
        call SetHingeEfficiency(pf, GetRealInput(0.0d0, 1.0d0, pf%HingeEfficiency))
    end subroutine EditHingeEfficiency

    subroutine EditDeflectionEfficiency(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)
        msg = "Enter deflection efficiency or press <ENTER> to accept default"
        call DisplayMessageWithRealDefault(msg, pf%DeflectionEfficiency, 0)
        call SetDeflectionEfficiency(pf, GetRealInput(0.0d0, 1.0d0, &
            & pf%DeflectionEfficiency))
    end subroutine EditDeflectionEfficiency

    subroutine EditFileName(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)
        msg = "Enter output file name or press <ENTER> to accept default"
        call DisplayMessageWithTextDefault(msg, pf%FileName, 0)
        call SetFileName(pf, GetStringInput(pf%FileName))
    end subroutine EditFileName
```

```fortran

    subroutine EditAngleOfAttack(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)
        write(6, '(a, a)') "NOTE: This operation will calculate a new lift ", &
            & "coefficient and optimum washout."

        msg = "Enter angle of attack or press <ENTER> to accept default"
        call DisplayMessageWithAngleDefault(msg, pf%DesiredAngleOfAttack, 0)
        call SetAngleOfAttack(pf, GetRealInput(-12.0d0, 12.0d0, &
            & pf%DesiredAngleOfAttack * 180.0d0 / pi))
    end subroutine EditAngleOfAttack

    subroutine EditLiftCoefficient(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg
        real*8 :: mn, mx, dflt

        mn = CL1(pf%CLa, -12.0d0 * pi / 180.0d0, pf%EW, pf%Washout)
        mx = CL1(pf%CLa,  12.0d0 * pi / 180.0d0, pf%EW, pf%Washout)
        if (pf%DesiredLiftCoefficient < mn) then
            dflt = mn
        else if (pf%DesiredLiftCoefficient > mx) then
            dflt = mx
        else
            dflt = pf%DesiredLiftCoefficient
        end if

        write(6, *)
        write(6, '(a, a)') "NOTE: This operation will calculate a new alpha ", &
            & "and optimum washout"
        msg = "Enter lift coefficient or press <ENTER> to accept default"
        call DisplayMessageWithRealDefault(msg, dflt, 0)
        call SetLiftCoefficient(pf, GetRealInput(mn, mx, dflt))
    end subroutine EditLiftCoefficient

    subroutine EditWashout(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)
        if (pf%UseOptimumWashout) then
            write(6, '(a)') "NOTE: Use of optimum total washout has been disabled."
        end if
        msg = "Enter total washout or press <ENTER> to accept default"
        call DisplayMessageWithAngleDefault(msg, pf%DesiredWashout, 0)
        call SetWashout(pf, GetRealInput(-12.0d0, 12.0d0, pf%DesiredWashout * 180.0d0 / pi))
    end subroutine EditWashout

    subroutine ToggleUseOptimumWashout(pf)
        type(Planform), intent(inout) :: pf

        if (pf%UseOptimumWashout) then
            call SetWashout(pf, pf%DesiredWashout * 180.0d0 / pi)
```

```
        else
            call SetOptimumWashout(pf)
        end if
    end subroutine ToggleUseOptimumWashout

    subroutine EditAileronDeflection(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)
        msg = "Enter aileron deflection or press <ENTER> to accept default"
        call DisplayMessageWithAngleDefault(msg, pf%AileronDeflection, 0)
        call SetAileronDeflection(pf, GetRealInput(-12.0d0, 12.0d0, &
            & pf%AileronDeflection * 180.0d0 / pi))
    end subroutine EditAileronDeflection

    subroutine ToggleUseSteadyRollingRate(pf)
        type(Planform), intent(inout) :: pf

        pf%UseSteadyRollingRate = .not. pf%UseSteadyRollingRate
        if (pf%UseSteadyRollingRate) then
            call SetSteadyRollingRate(pf)
        else
            call SetRollingRate(pf, pf%DesiredRollingRate)
        end if
    end subroutine ToggleUseSteadyRollingRate

    subroutine EditRollingRate(pf)
        type(Planform), intent(inout) :: pf

        character*80 :: msg

        write(6, *)

        if (pf%UseSteadyRollingRate) then
            write(6, '(a)') "NOTE: Use of steady rolling rate has been disabled."
        end if

        msg = "Enter dimensionless rolling rate or press <ENTER> to accept default"
        call DisplayMessageWithRealDefault(msg, pf%DesiredRollingRate, 0)
        call SetRollingRate(pf, GetRealInput(-100.0d0, 100.0d0, pf%DesiredRollingRate))
    end subroutine EditRollingRate

    subroutine DisplayMessageWithRealDefault(msg, dflt, tab)
        character*80, intent(in) :: msg ! Message to be displayed
        real*8, intent(in) :: dflt ! Default value to show in parenthesis
        integer, intent(in) :: tab ! Size of indentation to use

        call DisplayMessageWithTextDefault(msg, FormatReal(dflt, 5), tab)
    end subroutine DisplayMessageWithRealDefault

    subroutine DisplayMessageWithAngleDefault(msg, dflt, tab)
        character*80, intent(in) :: msg ! Message to be displayed
        real*8, intent(in) :: dflt ! Default value to show in parenthesis
        integer, intent(in) :: tab ! Size of indentation to use

        character*80 :: dflt_deg
```

```fortran

        write(dflt_deg, '(a, a)') trim(FormatReal(dflt * 180.0d0 / pi, 5)), " degrees"
        call DisplayMessageWithTextDefault(msg, dflt_deg, tab)
    end subroutine DisplayMessageWithAngleDefault

    subroutine DisplayMessageWithIntegerDefault(msg, dflt, tab)
        character*80, intent(in) :: msg ! Message to be displayed
        integer, intent(in) :: dflt ! Default value to show in parenthesis
        integer, intent(in) :: tab ! Size of indentation to use

        call DisplayMessageWithTextDefault(msg, FormatInteger(dflt), tab)
    end subroutine DisplayMessageWithIntegerDefault

    subroutine DisplayMessageWithTextDefault(msg, dflt, tab)
        character*80, intent(in) :: msg ! Message to be displayed
        character*80, intent(in) :: dflt ! Default value to show in parenthesis
        integer, intent(in) :: tab ! Size of indentation to use

        character*80 :: msg_fmt

        if (tab == 0) then
            msg_fmt = "(a, a, a, a)"
        else
            write(msg_fmt, '(a, i1, a)') "(", tab, "x, a, a, a, a)"
        end if

        write(6, msg_fmt) trim(msg), " ( ", trim(dflt), " )"
    end subroutine DisplayMessageWithTextDefault

    subroutine DisplayMessageWithLogicalDefault(msg, dflt, tab)
        character*80, intent(in) :: msg ! Message to be displayed
        logical, intent(in) :: dflt ! Default value to show in parenthesis
        integer, intent(in) :: tab ! Size of indentation to use

        character*80 :: tf

        if (dflt) then
            tf = "True"
        else
            tf = "False"
        end if
        call DisplayMessageWithTextDefault(msg, tf, tab)

    end subroutine DisplayMessageWithLogicalDefault

end module LiftingLineInterface
```

K.3 Planform Class Module (`class_Planform.f90`)

```fortran
module class_Planform
    use Utilities
    implicit none

    public :: Planform

    ! Supported wing types
    enum, bind(C)
        enumerator :: Tapered = 1, Elliptic = 2, Combination = 3
    end enum
```

```
    ! Supported washout distribution types
    enum, bind(C)
        enumerator :: Linear = 1, Optimum = 2
    end enum

    ! Supported low-aspect-ratio methods
    enum, bind(C)
        enumerator :: Classical=1, Hodson=2, ModifiedSlender=3, Kuchemann=4
    end enum

    type Planform
        ! Wing Parameters
        integer :: WingType = Tapered ! Wing type
        integer :: WashoutDistribution = Linear ! Washout distribution type
        integer :: NNodes = 99 ! Total number of nodes
        real*8 :: AspectRatio = 5.56d0 ! Aspect ratio
        real*8 :: TaperRatio = 1.0d0 ! Taper ratio (tapered wing only)
        real*8 :: TransitionPoint = 0.25d0 ! Transition point (Combination wing only)
        real*8 :: TransitionChord = 1.0d0 ! c/croot at transtion point (Combination wing only)
        real*8 :: SectionLiftSlope = 2.0d0 * pi ! Section lift slope
        real*8 :: AileronRoot = 0.253d0 ! Location of aileron root (z/b)
        real*8 :: AileronTip = 0.438d0 ! Location of aileron tip (z/b)
        logical :: ParallelHingeLine = .true. ! Is the hinge line parallel to the
                                              ! quarter-chord line? When true,
                                              ! FlapFractionTip will be calculated
        real*8 :: DesiredFlapFractionRoot = 0.28d0 ! Desired flap fraction at aileron root (cf/c)
        real*8 :: FlapFractionRoot = 0.28d0 ! Flap fraction at aileron root (cf/c)
        real*8 :: FlapFractionTip = 0.25d0 ! Flap fraction at aileron tip (cf/c)
        real*8 :: HingeEfficiency = 0.85d0 ! Aileron hinge efficiency
        real*8 :: DeflectionEfficiency = 1.0d0 ! Aileron deflection efficiency
        integer :: LowAspectRatioMethod = Classical ! Low-Aspect-Ratio correction method

        ! Coefficients for Tapered wing with elliptic tip
        real*8 :: C1 = 0.0d0 ! Represents transition point
        real*8 :: C2 = 0.0d0 ! Represents slope of tapered section
        real*8 :: C3 = 0.0d0 ! Represents secondary axis of ellipse
        real*8 :: C4 = 0.0d0 ! Represents ellipse center offset
        real*8 :: C5 = 0.0d0 ! Represents croot/b

        ! Output Options
        logical :: OutputMatrices = .true.  ! Write C Matrix and Fourier coefficients to output file?
        character*80 :: FileName = "planform.out" ! Name of output file

        ! Operating Conditions
        real*8 :: DesiredAngleOfAttack = pi / 36.0d0 ! Desired root aerodynamic angle of Attack
                                                     ! (alpha - alpha_L0), in radians
                                                     ! When specified, a new LiftCoefficient is calculated
        real*8 :: AngleOfAttack = pi / 36.0d0 ! Root Aerodynamic Angle of Attack
                                              ! (alpha - alpha_L0), in radians
```

```fortran
        real*8 :: DesiredLiftCoefficient = 0.4d0 ! Desired lift coefficient
                                                 ! When specified, a new AngleOfAttack is calculated
        real*8 :: LiftCoefficient = 0.4d0 ! Lift coefficient (user input, ignored if SpecifyAlpha == .true.)
        real*8 :: DesiredWashout = 0.0d0 ! Desired total washout, in radians
        real*8 :: OptimumWashout1 = 0.0d0 ! Optimum total washout, in radians (Eq. 1.8.37)
        real*8 :: OptimumWashout2 = 0.0d0 ! Optimum total washout, in radians (Eq. 1.8.42)
        real*8 :: Washout = 0.0d0 ! Total washout to use
        logical :: UseOptimumWashout = .true. ! Use the optimum total washout?
        real*8 :: AileronDeflection = 0.0d0 ! Aileron deflection, in radians
        real*8 :: DesiredRollingRate = 0.0d0 ! Desired dimensionless rolling rate (constant over wingspan)
        real*8 :: RollingRate = 0.0d0 ! Dimensionless rolling rate (constant over wingspan)
        logical :: SpecifyAlpha = .true. ! Was alpha specified?
                                         ! .true.  = Use desired alpha to calculate CL
                                         ! .false. = Use desired CL to calculate alpha
        logical :: UseSteadyRollingRate = .true. ! Use the steady dimensionless rolling rate?

        ! Planform Calculations
        real*8, allocatable, dimension(:,:) :: BigC, BigC_Inv
        real*8, allocatable, dimension(:) :: a, b, c, d, BigA
        real*8, allocatable, dimension(:) :: Omega
        logical :: IsAllocated = .false.

        ! Lift Coefficient Calculations
        real*8 :: KL  ! Lift slope factor
        real*8 :: EW  ! Washout effectiveness (epsilon omega)
        real*8 :: CLa ! Wing lift slope (derivative of CL with respect to alpha)
        real*8 :: CL1 ! Lift Coefficient (Eq. 1.8.24)
        real*8 :: CL2 ! Lift Coefficient (Eq. 1.8.5)

        ! Drag Coefficient Calculations
        real*8 :: KD   ! Induced drag factor
        real*8 :: KDL  ! Lift-washout contribution to induced drag
        real*8 :: KDW  ! Washout contribution to induced drag
        real*8 :: ES   ! Span efficiency factor
        real*8 :: CDi1 ! Induced drag coefficient (Eq. 1.8.25)
        real*8 :: CDi2 ! Induced drag coefficient (Eq. 1.8.6)
        real*8 :: CDi3 ! Induced drag coefficient (Eq. 32, Wing Flapping paper)

        ! Roll/yaw calculations
        real*8 :: CRM_da   ! Change in rolling moment coefficient with respect to alpha
        real*8 :: CRM_pbar ! Change in rolling moment coefficient with respect to rolling rate
        real*8 :: CRM      ! Rolling moment coefficient
        real*8 :: CYM      ! Yawing moment coefficient

    end type Planform

    contains
        character*80 function GetWingType(pf) result(name)
            type(Planform), intent(in) :: pf

```

```
        if (pf%WingType .eq. Tapered) then
            name = "Tapered"
        else if (pf%WingType .eq. Elliptic) then
            name = "Elliptic"
        else if (pf%WingType .eq. Combination) then
            name = "Tapered with elliptic tip"
        else
            name = "Unknown"
        end if
    end function GetWingType

    character*80 function GetWashoutDistributionType(pf) result(name)
        type(Planform), intent(in) :: pf

        if (pf%WashoutDistribution .eq. Linear) then
            name = "Linear"
        else if (pf%WashoutDistribution .eq. Optimum) then
            name = "Optimum"
        else
            name = "Unknown"
        end if
    end function GetWashoutDistributionType

    character*80 function GetLowAspectRatioMethod(pf) result(name)
        type(Planform), intent(in) :: pf

        if (pf%LowAspectRatioMethod .eq. Classical) then
            name = "Classical"
        else if (pf%LowAspectRatioMethod .eq. Hodson) then
            name = "Hodson"
        else if (pf%LowAspectRatioMethod .eq. ModifiedSlender) then
            name = "Modified Slender Wing"
        else if (pf%LowAspectRatioMethod .eq. Kuchemann) then
            name = "Kuchemann"
        else
            name = "Unknown"
        end if
    end function GetLowAspectRatioMethod

    real*8 function theta_i(i, nnodes) result(theta)
        integer, intent(in) :: i
        integer, intent(in) :: nnodes

        if (i < 1 .or. i > nnodes) then
            write(6, '(a, i3)') "ERROR: Function theta_i called with i = ", i
            if (i < 1) then
                theta = 0.0d0
            else
                theta = pi
            end if
        else
            theta = real(i-1, 8) / real(nnodes - 1, 8) * pi
        end if
    end function theta_i

    real*8 function theta_zb(zb) result(theta)
        real*8, intent(in) :: zb  ! z/b

        if (zb < -0.5d0 .or. zb > 0.5d0) then
```

```
                write(6, '(a, f7.4)') "ERROR: Function theta_d called with z/b = ", zb
                if (zb < -0.5d0) then
                    theta = 0.0d0
                else
                    theta = pi
                end if
            else
                theta = acos(-2.0d0 * zb)
            end if
        end function theta_zb

        real*8 function c_over_b_i(pf, i) result(cb)
            type(Planform), intent(in) :: pf
            integer, intent(in) :: i

            real*8 :: theta

            theta = theta_i(i, pf%NNodes)
            cb = c_over_b(pf, theta)
        end function c_over_b_i

        real*8 function c_over_b_zb(pf, zb) result(cb)
            type(Planform), intent(in) :: pf
            real*8, intent(in) :: zb  ! z/b

            real*8 :: theta

            theta = theta_zb(zb)
            cb = c_over_b(pf, theta)
        end function c_over_b_zb

        real*8 function c_over_b(pf, theta) result(cb)
            type(Planform), intent(in) :: pf
            real*8, intent(in) :: theta

            real*8 :: zb, u

            if (pf%WingType == Tapered) then
                ! Calculate c/b for tapered wing
                cb = (2.0d0 * (1.0d0 - (1.0d0 - pf%TaperRatio) * &
                    & dabs(cos(theta)))) / (pf%AspectRatio * (1.0d0 + pf%TaperRatio))
            else if (pf%WingType == Elliptic) then
                ! Calculate c/b for elliptic wing
                cb = (4.0d0 * sin(theta)) / &
                    & (pi * pf%AspectRatio)
            else if (pf%WingType == Combination) then
                ! Calculate c/b for combination wing
                zb = abs(z_over_b(theta))
                if (zb <= pf%TransitionPoint) then
                    cb = pf%C5 * (1.0d0 - pf%C2 * zb)
                else
                    u = (zb - pf%C4) / (0.5d0 - pf%C4)
                    cb = pf%C5 * pf%C3 * sqrt(1.0d0 - u**2)
                end if
            else
                ! Unknown wing type!
                stop "*** Unknown Wing Type ***"
            end if
        end function c_over_b
```

```
        real*8 function z_over_b_i(i, nnodes) result(zb)
            integer, intent(in) :: i
            integer, intent(in) :: nnodes

            zb = z_over_b(theta_i(i, nnodes))
        end function z_over_b_i

        real*8 function z_over_b(theta) result(zb)
            real*8, intent(in) :: theta

            zb = -0.5d0 * cos(theta)
        end function z_over_b

        real*8 function cf_over_c_i(pf, i) result(cfc)
            type(Planform), intent(in) :: pf
            integer, intent(in) :: i

            real*8 :: zbi

            zbi = z_over_b_i(i, pf%NNodes)
            if (Compare(dabs(zbi), pf%AileronRoot, zero) == -1 .or. &
                & Compare(dabs(zbi), pf%AileronTip, zero) == 1) then
                cfc = 0.0d0
            else
                cfc = 0.75d0 - y_i(pf, i) / c_over_b_i(pf, i)
            end if
        end function cf_over_c_i

        real*8 function y_i(pf, i) result(y)
            type(Planform), intent(in) :: pf
            integer, intent(in) :: i

            real*8 :: zb_i, cb_i
            real*8 :: zb_root, cfc_root, theta_root, cb_root, y_root
            real*8 :: zb_tip, cfc_tip, theta_tip, cb_tip, y_tip
            real*8 :: slope, offst

            zb_root = pf%AileronRoot
            cfc_root = pf%FlapFractionRoot
            theta_root = theta_zb(zb_root)
            cb_root = c_over_b(pf, theta_root)
            y_root = (0.75d0 - cfc_root) * cb_root

            zb_tip = pf%AileronTip
            cfc_tip = pf%FlapFractionTip
            theta_tip = theta_zb(zb_tip)
            cb_tip = c_over_b(pf, theta_tip)
            y_tip = (0.75d0 - cfc_tip) * cb_tip

            slope = (y_tip - y_root) / (zb_tip - zb_root)
            offst = y_root - slope * zb_root

            zb_i = z_over_b_i(i, pf%NNodes)
            y = slope * dabs(zb_i) + offst
        end function y_i

        real*8 function FlapEffectiveness(pf, i) result(eps_f)
            type(Planform), intent(in) :: pf
            integer, intent(in) :: i

```

```
            real*8 :: theta_f, eps_fi

            theta_f = acos(2.0d0 * cf_over_c_i(pf, i) - 1.0d0)
            eps_fi = 1.0d0 - (theta_f - sin(theta_f)) / pi
            eps_f = eps_fi * pf%HingeEfficiency * pf%DeflectionEfficiency
        end function FlapEffectiveness

        subroutine DeallocateArrays(pf)
            type(Planform), intent(inout) :: pf

            if (pf%IsAllocated) then
                deallocate(pf%BigC)
                deallocate(pf%BigC_Inv)
                deallocate(pf%a)
                deallocate(pf%b)
                deallocate(pf%c)
                deallocate(pf%d)
                deallocate(pf%BigA)
                deallocate(pf%Omega)

                pf%IsAllocated = .false.
            end if
        end subroutine DeallocateArrays

        subroutine AllocateArrays(pf)
            type(Planform), intent(inout) :: pf

            if (pf%IsAllocated) call DeallocateArrays(pf)

            allocate(pf%BigC(pf%NNodes, pf%NNodes))
            allocate(pf%BigC_Inv(pf%NNodes, pf%NNodes))
            allocate(pf%a(pf%NNodes))
            allocate(pf%b(pf%NNodes))
            allocate(pf%c(pf%NNodes))
            allocate(pf%d(pf%NNodes))
            allocate(pf%BigA(pf%NNodes))
            allocate(pf%Omega(pf%NNodes))
            pf%IsAllocated = .true.
        end subroutine AllocateArrays

end module class_Planform
```

K.4 Module of Setter Functions (`liftinglinesetters.f90`)

```
module LiftingLineSetters
    use class_Planform
    implicit none

contains
    subroutine InitPlanform(pf)
        type(Planform), intent(inout) :: pf

        call SetParallelHingeLine(pf)
    end subroutine InitPlanform

    ! Planform Parameters
    subroutine SetWingType(pf, wingType)
        type(Planform), intent(inout) :: pf
        integer, intent(in) :: wingType

```

```
        if (pf%WingType /= wingType) then
            pf%WingType = wingType
            call DeallocateArrays(pf)

            if (pf%WingType == Combination) then
                call SetCombinationWingCoefficients(pf)
            else if (pf%ParallelHingeLine) then
                call SetParallelHingeLine(pf)
            end if
            if (pf%WingType == Elliptic) then
                pf%WashoutDistribution = Linear
            end if
        end if
    end subroutine SetWingType

    subroutine SetWashoutDistribution(pf, washoutDist)
        type(Planform), intent(inout) :: pf
        integer, intent(in) :: washoutDist

        if (pf%WingType /= Elliptic .and. pf%WashoutDistribution /= washoutDist) then
            pf%WashoutDistribution = washoutDist
            call DeallocateArrays(pf)
        end if
    end subroutine SetWashoutDistribution

    subroutine SetLowAspectRatioMethod(pf, lowAspectRatioMethod)
        type(Planform), intent(inout) :: pf
        integer, intent(in) :: lowAspectRatioMethod

        if (pf%LowAspectRatioMethod /= lowAspectRatioMethod) then
            pf%LowAspectRatioMethod = lowAspectRatioMethod
            call DeallocateArrays(pf)
        end if
    end subroutine SetLowAspectRatioMethod

    subroutine SetTransitionPoint(pf, tp)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: tp

        if (Compare(pf%TransitionPoint, tp, zero) /= 0) then
            pf%TransitionPoint = tp
            call DeallocateArrays(pf)
            call SetCombinationWingCoefficients(pf)
        end if
    end subroutine SetTransitionPoint

    subroutine SetTransitionChord(pf, tc)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: tc

        if (Compare(pf%TransitionChord, tc, zero) /= 0) then
            pf%TransitionChord = tc
            call DeallocateArrays(pf)
            call SetCombinationWingCoefficients(pf)
        end if
    end subroutine SetTransitionChord

    subroutine SetCombinationWingCoefficients(pf)
        type(Planform), intent(inout) :: pf

```

```fortran
        real*8 :: u, asin_u

        pf%C1 = pf%TransitionPoint
        pf%C2 = (1.0d0 - pf%TransitionChord) / pf%C1
        pf%C4 = (pf%C1 - 0.25d0 * pf%C2) / (pf%C1 * pf%C2 - pf%C2 + 1.0d0)
        u = (pf%C1 - pf%C4) / (0.5d0 - pf%C4)
        asin_u = asin(u)
        pf%C3 = (1.0d0 - pf%C1 * pf%C2) / sqrt(1.0d0 - u**2)
        pf%C5 = 1.0d0 / (pf%AspectRatio * (2.0d0 * pf%C1 - pf%C1**2 * pf%C2 + &
            & 0.5d0 * pf%C3 * (0.5d0 - pf%C4) * (pi - 2.0d0 * asin_u - &
            & sin(2.0d0 * asin_u))))

        if (pf%ParallelHingeLine) then
            call SetParallelHingeLine(pf)
        end if
    end subroutine SetCombinationWingCoefficients

    logical function AreCombinationWingCoefficientsValid(pf) result(isValid)
        type(Planform), intent(in) :: pf

        real*8 :: u, d1, d2

        u = (pf%C1 - pf%C4) / (0.5d0 - pf%C4)
        d1 = -pf%C2
        d2 = -(pf%C3 * u) / (sqrt(1.0d0 - u**2) * (0.5d0 - pf%C4))

        if (Compare(pf%C1, 0.0d0, zero) /= 1 .and. &
            & Compare(pf%C1, 0.5d0, zero) /= -1) then
            isValid = .false.
        else if (Compare(pf%C1 * pf%C2 - pf%C2 + 1.0d0, 0.0d0, zero) == 0) then
            isValid = .false.
        else if (Compare(pf%C4, 0.5d0, zero) == 0) then
            isValid = .false.
        else if (Compare(dabs(u), 1.0d0, zero) /= -1) then
            isValid = .false.
        else if (Compare(d1, d2, zero) /= 0) then
            isValid = .false.
        else
            isValid = .true.
        end if
    end function AreCombinationWingCoefficientsValid

    subroutine SetNNodes(pf, npss)
        type(Planform), intent(inout) :: pf
        integer, intent(in) :: npss

        integer :: nnodes

        nnodes = npss * 2 - 1
        if (pf%NNodes /= nnodes) then
            pf%NNodes = nnodes
            call DeallocateArrays(pf)
        end if
    end subroutine SetNNodes

    subroutine SetAspectRatio(pf, ra)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: ra

        if (Compare(pf%AspectRatio, ra, zero) /= 0) then
            pf%AspectRatio = ra
```

```
            call DeallocateArrays(pf)
        end if
    end subroutine SetAspectRatio

    subroutine SetTaperRatio(pf, rt)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: rt

        if (Compare(pf%TaperRatio, rt, zero) /= 0) then
            pf%TaperRatio = rt
            call DeallocateArrays(pf)

            if (pf%ParallelHingeLine) then
                call SetParallelHingeLine(pf)
            end if
        end if
    end subroutine SetTaperRatio

    subroutine SetSectionLiftSlope(pf, cla_sec)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: cla_sec

        if (Compare(pf%SectionLiftSlope, cla_sec, zero) /= 0) then
            pf%SectionLiftSlope = cla_sec
            call DeallocateArrays(pf)
        end if
    end subroutine SetSectionLiftSlope

    subroutine SetAileronRoot(pf, ar)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: ar

        if (Compare(pf%AileronRoot, ar, zero) /= 0) then
            pf%AileronRoot = ar
            call DeallocateArrays(pf)
        end if
    end subroutine SetAileronRoot

    subroutine SetAileronTip(pf, at)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: at

        if (Compare(pf%AileronTip, at, zero) /= 0) then
            pf%AileronTip = at
            call DeallocateArrays(pf)
        end if
    end subroutine SetAileronTip

    subroutine SetParallelHingeLine(pf)
        type(Planform), intent(inout) :: pf

        real*8 :: cfc_root_par

        pf%ParallelHingeLine = .true.
        cfc_root_par = ParallelRootFlapFraction(pf)
        if (Compare(pf%FlapFractionRoot, cfc_root_par, zero) /= 0) then
            pf%FlapFractionRoot = cfc_root_par
            call DeallocateArrays(pf)
        end if
    end subroutine SetParallelHingeLine
```

```fortran
    subroutine SetFlapFractionRoot(pf, cfc_root)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: cfc_root

        pf%ParallelHingeLine = .false.
        pf%DesiredFlapFractionRoot = cfc_root
        if (Compare(pf%FlapFractionRoot, cfc_root, zero) /= 0) then
            pf%FlapFractionRoot = cfc_root
            call DeallocateArrays(pf)
        end if
    end subroutine SetFlapFractionRoot

    subroutine SetFlapFractionTip(pf, cfc_tip)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: cfc_tip

        if (Compare(pf%FlapFractionTip, cfc_tip, zero) /= 0) then
            pf%FlapFractionTip = cfc_tip
            call DeallocateArrays(pf)

            if (pf%ParallelHingeLine) then
                call SetParallelHingeLine(pf)
            end if
        end if
    end subroutine SetFlapFractionTip

    subroutine SetHingeEfficiency(pf, eff_hinge)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: eff_hinge

        if (Compare(pf%HingeEfficiency, eff_hinge, zero) /= 0) then
            pf%HingeEfficiency = eff_hinge
            call DeallocateArrays(pf)
        end if
    end subroutine SetHingeEfficiency

    subroutine SetDeflectionEfficiency(pf, eff_def)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: eff_def

        if (Compare(pf%DeflectionEfficiency, eff_def, zero) /= 0) then
            pf%DeflectionEfficiency = eff_def
            call DeallocateArrays(pf)
        end if
    end subroutine SetDeflectionEfficiency

    subroutine ToggleOutputMatricies(pf)
        type(Planform), intent(inout) :: pf

        pf%OutputMatrices = .not. pf%OutputMatrices
    end subroutine ToggleOutputMatricies

    subroutine SetFileName(pf, filename)
        type(Planform), intent(inout) :: pf
        character*80, intent(in) :: filename

        pf%FileName = trim(filename)
    end subroutine SetFileName


    ! Operating Conditions
```

```fortran
    subroutine SetAngleOfAttack(pf, alpha)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: alpha

        pf%SpecifyAlpha = .true.
        pf%DesiredAngleOfAttack = alpha * pi / 180.0d0
        pf%AngleOfAttack = pf%DesiredAngleOfAttack
        if (pf%UseOptimumWashout) then
            call SetOptimumWashout(pf)
        end if
        pf%LiftCoefficient = CL1(pf%CLa, pf%AngleOfAttack, pf%EW, pf%Washout)
    end subroutine SetAngleOfAttack

    subroutine SetLiftCoefficient(pf, cl)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: cl

        pf%SpecifyAlpha = .false.
        pf%DesiredLiftCoefficient = cl
        pf%LiftCoefficient = pf%DesiredLiftCoefficient
        if (pf%UseOptimumWashout) then
            call SetOptimumWashout(pf)
        end if
        pf%AngleOfAttack = RootAlpha(pf%CLa, pf%LiftCoefficient, pf%EW, pf%Washout)
    end subroutine SetLiftCoefficient

    subroutine SetAileronDeflection(pf, da)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: da

        pf%AileronDeflection = da * pi / 180.0d0
    end subroutine SetAileronDeflection

    subroutine SetWashout(pf, washout)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: washout

        pf%DesiredWashout = washout * pi / 180.0d0
        pf%Washout = pf%DesiredWashout
        pf%UseOptimumWashout = .false.
        if (pf%SpecifyAlpha) then
            pf%LiftCoefficient = CL1(pf%CLa, pf%AngleOfAttack, pf%EW, pf%Washout)
        else
            pf%AngleOfAttack = RootAlpha(pf%CLa, pf%LiftCoefficient, pf%EW, pf%Washout)
        end if
    end subroutine SetWashout

    subroutine SetOptimumWashout(pf)
        type(Planform), intent(inout) :: pf

        logical :: cont
        integer :: i
        real*8 :: oldCL, newCL, resCL
        real*8 :: oldOmega, newOmega, resOmega

        if (pf%SpecifyAlpha) then
            oldCL = pf%LiftCoefficient
            oldOmega = pf%Washout
```

```
            cont = .true.
            i = 0
            do while (i < 100 .or. (cont .and. i < 1000))
                i = i + 1
                newOmega = OptimumWashout1(pf%KDL, oldCL, pf%KDW, pf%CLa)
                newCL = CL1(pf%CLa, pf%AngleOfAttack, pf%EW, newOmega)

                resCL = Residual(oldCL, newCL)
                resOmega = Residual(oldOmega, newOmega)

                cont = (Compare(resCL, 0.0d0, zero) /= 0 .or. Compare(resOmega, 0.0d0, zero) /= 0)
                oldOmega = newOmega
                oldCL = newCL
            end do

            if (i >= 1000) then
                stop "*** Max number of convergence iterations reached! ***"
            else
                pf%LiftCoefficient = newCL
                pf%OptimumWashout1 = newOmega
            end if
        end if

        pf%OptimumWashout1 = OptimumWashout1(pf%KDL, pf%LiftCoefficient, &
            & pf%KDW, pf%CLa)
        if (pf%WashoutDistribution == Optimum) then
            pf%OptimumWashout2 = OptimumWashout2(pf, pf%LiftCoefficient, &
                pf%AspectRatio, pf%SectionLiftSlope)
        end if
        pf%Washout = pf%OptimumWashout1
        pf%UseOptimumWashout = .true.

        if (.not. pf%SpecifyAlpha) then
            pf%AngleOfAttack = RootAlpha(pf%CLa, pf%LiftCoefficient, pf%EW, pf%Washout)
        end if
    end subroutine SetOptimumWashout

    subroutine SetRollingRate(pf, rollingrate)
        type(Planform), intent(inout) :: pf
        real*8, intent(in) :: rollingrate

        pf%DesiredRollingRate = rollingrate
        pf%RollingRate = rollingrate
        pf%UseSteadyRollingRate = .false.
    end subroutine

    subroutine SetSteadyRollingRate(pf)
        type(Planform), intent(inout) :: pf

        real*8 :: steady_pbar

        pf%RollingRate = SteadyRollingRate(pf)
        pf%UseSteadyRollingRate = .true.
    end subroutine SetSteadyRollingRate

    real*8 function ParallelRootFlapFraction(pf) result(cfc_root_par)
        type(Planform), intent(in) :: pf

        real*8 :: cb_root, cfc_root
```

```
        real*8 :: cb_tip, cfc_tip

        cb_tip = c_over_b_zb(pf, pf%AileronTip)
        cfc_tip = pf%FlapFractionTip
        cb_root = c_over_b_zb(pf, pf%AileronRoot)
        cfc_root_par = 0.75d0 - cb_tip / cb_root * (0.75d0 - cfc_tip)
    end function ParallelRootFlapFraction

    real*8 function RootAlpha(cla, cl, ew, omega) result(alpha)
        real*8, intent(in) :: cla
        real*8, intent(in) :: cl
        real*8, intent(in) :: ew
        real*8, intent(in) :: omega

        alpha = cl / cla + ew * omega
    end function RootAlpha

    real*8 function SteadyRollingRate(pf) result(pbar_steady)
        type(Planform), intent(in) :: pf

        ! Calculate steady dimensionless rolling rate (Eq. 1.8.59)
        pbar_steady = -pf%CRM_da / pf%CRM_pbar * pf%AileronDeflection
    end function SteadyRollingRate

    real*8 function OptimumWashout1(kdl, cl, kdw, cla) result(ow1)
        real*8, intent(in) :: kdl
        real*8, intent(in) :: cl
        real*8, intent(in) :: kdw
        real*8, intent(in) :: cla

        ow1 = (kdl * cl) / (2.0d0 * kdw * cla)
    end function OptimumWashout1

    real*8 function OptimumWashout2(pf, cl, ra, ls) result(ow2)
        type(Planform), intent(in) :: pf
        real*8, intent(in) :: cl
        real*8, intent(in) :: ra
        real*8, intent(in) :: ls

        ow2 = (4.0d0 * cl) / (pi * ra * ls * c_over_b(pf, pi / 2.0d0))
    end function OptimumWashout2

    real*8 function CL1(cla, alpha, ew, w) result(cl)
        real*8, intent(in) :: cla
        real*8, intent(in) :: alpha
        real*8, intent(in) :: ew
        real*8, intent(in) :: w

        cl = cla * (alpha - ew * w)  ! Eq. 1.8.24
    end function CL1

    real*8 function CL2(ra, bigA1) result(cl)
        real*8, intent(in) :: ra
        real*8, intent(in) :: bigA1

        cl = pi * ra * bigA1  ! Eq. 1.8.5
    end function CL2

    real*8 function CDi1(pf) result(cdi)
        type(Planform), intent(in) :: pf

```

```fortran
        real*8 :: n, a0, A, r1, r2

        a0 = pf%SectionLiftSlope
        A = pf%AspectRatio

        ! Set the low aspect ratio method parameters
        if (pf%LowAspectRatioMethod == Hodson) then
            r1 = a0
            r2 = A * (pi - atan((2.0 * a0) / (pi * A)))
        else if (pf%LowAspectRatioMethod == ModifiedSlender) then
            r1 = a0
            r2 = 0.5 * pi * A
        else if (pf%LowAspectRatioMethod == Kuchemann) then
            n = 1.0 - 0.5 * (1.0 + (a0 / (pi * A))**2)**(-0.25)
            r1 = 2 * n * a0 / (1.0 - pi * n / tan(pi * n))
            r2 = pi * A / (2.0 * n)
        else ! Assume Classical
            r1 = a0
            r2 = pi * A
        end if

        cdi = (pf%CL1**2 * (1.0d0 + pf%KD) - pf%KDL * pf%CL1 * pf%CLa * &
            & pf%Washout + pf%KDW * (pf%CLa * pf%Washout)**2) / (pi * A) &
            & * (r1 / r2) * (pi * A) / a0  ! Low-RA correction
    end function CDi1

    real*8 function CDi2(pf) result(cdi)
        type(Planform), intent(in) :: pf

        integer :: i

        cdi = 0.0d0
        do i = 1, pf%NNodes
            cdi = cdi + real(i, 8) * pf%BigA(i)**2
        end do
        cdi = cdi * pi * pf%AspectRatio
    end function CDi2

    real*8 function CDi3(pf) result(cdi)
        type(Planform), intent(in) :: pf

        integer :: i
        real*8 :: ri

        cdi = 0.0d0
        do i = 1, pf%NNodes
            ri = real(i, 8)
            cdi = cdi + ri * pf%BigA(i)**2
        end do
        cdi = (cdi - 0.5d0 * pf%RollingRate * pf%BigA(2)) * pi * pf%AspectRatio
    end function CDi3

end module LiftingLineSetters
```

K.5 Solver Module (`liftinglinesolver.f90`)

```fortran
module LiftingLineSolver
    use class_Planform
    use LiftingLineSetters
    use LiftingLineOutput
```

```
    use matrix
    implicit none

contains
    subroutine ComputeCMatrixAndCoefficients(pf)
        type(Planform), intent(inout) :: pf

        write(6, '(a)') "Calculating C matrix and Fourier coefficients, please wait..."
        write(6, '(a, a, a)') "Estimated calculation time: ", &
            & trim(FormatReal(pf%NNodes**2 * 1.0d-5, 3)), " seconds"
        write(6, *)

        if (.not. pf%IsAllocated) then
            if (pf%WingType == Combination) then
                call SetCombinationWingCoefficients(pf)
            end if

            call AllocateArrays(pf)

            call ComputeC(pf, pf%BigC)
            call ComputeCInverse(pf, pf%BigC_Inv)
            call ComputeFourierCoefficients_a(pf, pf%a)
            call ComputeFourierCoefficients_b(pf, pf%b, pf%Omega)
            call ComputeFourierCoefficients_c(pf, pf%c)
            call ComputeFourierCoefficients_d(pf, pf%d)

            call ComputeLiftCoefficientParameters(pf)
            call ComputeDragCoefficientParameters(pf)
            call ComputeRollCoefficientParameters(pf)
            call ComputeFlightConditions(pf)
        end if
    end subroutine ComputeCMatrixAndCoefficients

    subroutine ComputeFourierCoefficients_a(pf, a)
        type(Planform), intent(in) :: pf
        real*8, intent(out) :: a(pf%NNodes)

        real*8 :: ones(pf%NNodes)
        integer :: i
        integer :: nnodes

        nnodes = pf%NNodes
        ones = (/ (1.0d0, i=1, nnodes) /)
        if (pf%WingType == Tapered .and. Compare(pf%TaperRatio, 0.0d0, zero) == 0) then
            ones(1) = 0.0d0
            ones(nnodes) = 0.0d0
        end if

        a = matmul(pf%BigC_Inv, ones)
    end subroutine ComputeFourierCoefficients_a

    subroutine ComputeFourierCoefficients_b(pf, b, omega)
        type(Planform), intent(in) :: pf
        real*8, intent(out) :: b(pf%NNodes)
        real*8, intent(out) :: omega(pf%NNodes)

        real*8 :: croot_over_b, theta
        integer :: i
        integer :: nnodes
```

```
        nnodes = pf%NNodes
        if (pf%WashoutDistribution == Linear) then
            omega = (/ (dabs(cos(theta_i(i, nnodes))), i=1, nnodes) /)
            if (pf%WingType == Tapered .and. Compare(pf%TaperRatio, 0.0d0, zero) == 0) then
                omega(1) = 0.0d0
                omega(nnodes) = 0.0d0
            end if
        else if (pf%WashoutDistribution == Optimum) then
            croot_over_b = c_over_b(pf, pi / 2.0d0)
            do i = 1, nnodes
                theta = theta_i(i, nnodes)
                omega(i) = 1.0d0 - sin(theta) / (c_over_b(pf, theta) / croot_over_b)
            end do

            if (pf%WingType == Combination) then
                omega(1) = 1.0d0 - sqrt(1.0d0 - 2.0d0 * pf%C4) / pf%C3
                omega(nnodes) = omega(1)
            else if (pf%WingType == Tapered .and. Compare(pf%TaperRatio, 0.0d0, zero) == 0) then
                omega(1) = 2.0d0
                omega(nnodes) = 2.0d0
            end if
        else
            write(6, '(a)') "Unknown washout distribution type!"
            stop
        end if

        b = matmul(pf%BigC_Inv, omega)
    end subroutine ComputeFourierCoefficients_b

    subroutine ComputeFourierCoefficients_c(pf, c)
        type(Planform), intent(in) :: pf
        real*8, intent(out) :: c(pf%NNodes)

        real*8 :: chi(pf%NNodes)
        real*8 :: zbi
        integer :: i
        integer :: nnodes

        nnodes = pf%NNodes
        do i = 1, nnodes
            zbi = z_over_b_i(i, nnodes)
            chi(i) = -sign(FlapEffectiveness(pf, i), zbi)
        end do

        if (pf%WingType == Tapered .and. Compare(pf%TaperRatio, 0.0d0, zero) == 0) then
            chi(1) = 0.0d0
            chi(nnodes) = 0.0d0
        end if

        c = matmul(pf%BigC_Inv, chi)
    end subroutine ComputeFourierCoefficients_c

    subroutine ComputeFourierCoefficients_d(pf, d)
        type(Planform), intent(in) :: pf
        real*8, intent(out) :: d(pf%NNodes)

```

```fortran
        real*8 :: cos_theta(pf%NNodes)
        integer :: i
        integer :: nnodes

        nnodes = pf%NNodes
        cos_theta = (/ (cos(theta_i(i, nnodes)), i=1, nnodes) /)
        if (pf%WingType == Tapered .and. Compare(pf%TaperRatio, 0.0d0, zero) == 0) then
            cos_theta(1) = 0.0d0
            cos_theta(nnodes) = 0.0d0
        end if

        d = matmul(pf%BigC_Inv, cos_theta)
    end subroutine ComputeFourierCoefficients_d

    subroutine ComputeBigACoefficients(pf, bigA)
        type(Planform), intent(in) :: pf
        real*8, intent(out) :: bigA(pf%NNodes)

        integer :: i

        do i = 1, pf%NNodes
            bigA(i) = pf%a(i) * pf%AngleOfAttack - pf%b(i) * pf%Washout + &
                & pf%c(i) * pf%AileronDeflection + pf%d(i) * pf%RollingRate
        end do
    end subroutine ComputeBigACoefficients

    subroutine ComputeLiftCoefficientParameters(pf)
        type(Planform), intent(inout) :: pf

        pf%KL = Kappa_L(pf%AspectRatio, pf%SectionLiftSlope, pf%a(1))
        pf%EW = Epsilon_Omega(pf%a(1), pf%b(1))
        pf%CLa = C_L_alpha(pf%AspectRatio, pf%a(1))
    end subroutine ComputeLiftCoefficientParameters

    subroutine ComputeDragCoefficientParameters(pf)
        type(Planform), intent(inout) :: pf

        pf%KD = Kappa_D(pf%NNodes, pf%a)
        pf%ES = SpanEfficiencyFactor(pf%KD)
        pf%KDL = Kappa_DL(pf%NNodes, pf%a, pf%b)
        pf%KDW = Kappa_DOmega(pf%NNodes, pf%a, pf%b)
    end subroutine ComputeDragCoefficientParameters

    subroutine ComputeRollCoefficientParameters(pf)
        type(Planform), intent(inout) :: pf

        pf%CRM_da = CRM_dAlpha(pf%AspectRatio, pf%c(2))
        pf%CRM_pbar = CRM_PBar(pf%AspectRatio, pf%d(2))
    end subroutine ComputeRollCoefficientParameters

    real*8 function Kappa_L(ra, cla_section, a1) result(kl)
        real*8, intent(in) :: ra
        real*8, intent(in) :: cla_section
        real*8, intent(in) :: a1

        kl = 1.0d0 / ((1.0d0 + pi * ra / cla_section) * a1) - 1.0d0
    end function Kappa_L

    real*8 function Epsilon_Omega(a1, b1) result(ew)
        real*8, intent(in) :: a1
```

```
        real*8, intent(in) :: b1

        ew = b1 / a1
    end function Epsilon_Omega

    real*8 function C_L_alpha(ra, a1) result(cla)
        real*8, intent(in) :: ra
        real*8, intent(in) :: a1

        cla = pi * ra * a1
    end function C_L_alpha

    real*8 function Kappa_D(nnodes, a) result(kd)
        integer, intent(in) :: nnodes
        real*8, intent(in) :: a(nnodes)

        integer :: i

        kd = 0.0d0
        do i = 2, nnodes
            kd = kd + real(i, 8) * (a(i) / a(1))**2
        end do
    end function Kappa_D

    real*8 function SpanEfficiencyFactor(kd) result(es)
        real*8, intent(in) :: kd
        es = 1.0d0 / (1.0d0 + kd)
    end function SpanEfficiencyFactor

    real*8 function Kappa_DL(nnodes, a, b) result (kdl)
        integer, intent(in) :: nnodes
        real*8, intent(in) :: a(nnodes)
        real*8, intent(in) :: b(nnodes)

        integer :: i

        kdl = 0.0d0
        do i = 2, nnodes
            kdl = kdl + real(i, 8) * a(i) / a(1) * &
                & (b(i) / b(1) - a(i) / a(1))
        end do
        kdl = kdl * 2.0d0 * b(1) / a(1)
    end function Kappa_DL

    real*8 function Kappa_DOmega(nnodes, a, b) result(kdw)
        integer, intent(in) :: nnodes
        real*8, intent(in) :: a(nnodes)
        real*8, intent(in) :: b(nnodes)

        integer :: i

        kdw = 0.0d0
        do i = 2, nnodes
            kdw = kdw + real(i, 8) * (b(i) / b(1) - a(i) / a(1))**2
        end do
        kdw = kdw * (b(1) / a(1))**2
    end function Kappa_DOmega

    real*8 function CRM_dAlpha(ra, c2) result(crmda)
        real*8, intent(in) :: ra
        real*8, intent(in) :: c2
```

```fortran

        crmda = -pi * ra / 4.0d0 * c2
    end function CRM_dAlpha

    real*8 function CRM_PBar(ra, d2) result(crmpbar)
        real*8, intent(in) :: ra
        real*8, intent(in) :: d2

        crmpbar = -pi * ra / 4.0d0 * d2
    end function CRM_PBar

    subroutine ComputeFlightConditions(pf)
        type(Planform), intent(inout) :: pf

        ! Make sure planform characteristics have been computed
        if (.not. pf%IsAllocated) then
            call ComputeCMatrixAndCoefficients(pf)
        end if

        ! Compute root aerodynamic angle of attack, if necessary
        if (.not. pf%SpecifyAlpha) then
            pf%AngleOfAttack = RootAlpha(pf%CLa, pf%LiftCoefficient, pf%EW, pf%Washout)
        else
            pf%LiftCoefficient = CL1(pf%CLa, pf%AngleOfAttack, pf%EW, pf%Washout)
        end if

        ! Compute optimum total washout, if necessary
        if (pf%UseOptimumWashout) then
            call SetOptimumWashout(pf)
        else
            call SetWashout(pf, pf%DesiredWashout * 180.0d0 / pi)
        end if

        ! Compute steady rolling rate, if necessary
        if (pf%UseSteadyRollingRate) then
            call SetSteadyRollingRate(pf)
        end if

        ! Compute BigA Fourier Coefficients
        call ComputeBigACoefficients(pf, pf%BigA)

        ! Compute lift coefficients
        call ComputeLiftCoefficients(pf)

        ! Compute drag coefficient
        call ComputeDragCoefficients(pf)

        ! Compute roll coefficient
        pf%CRM = CRoll(pf%CRM_da, pf%CRM_pbar, pf%AileronDeflection, pf%RollingRate)

        ! Compute yaw coefficient
        pf%CYM = CYaw(pf, pf%CL1, pf%BigA)
    end subroutine ComputeFlightConditions

    subroutine ComputeLiftCoefficients(pf)
        type(Planform), intent(inout) :: pf

        pf%CL1 = CL1(pf%CLa, pf%AngleOfAttack, pf%EW, pf%Washout)
```

```
        pf%CL2 = CL2(pf%AspectRatio, pf%BigA(1))
    end subroutine ComputeLiftCoefficients

    subroutine ComputeDragCoefficients(pf)
        type(Planform), intent(inout) :: pf

        pf%CDi1 = CDi1(pf)
        pf%CDi2 = CDi2(pf)
        pf%CDi3 = CDi3(pf)
    end subroutine ComputeDragCoefficients

    real*8 function CRoll(crmda, crmpbar, da, pbar) result(crm)
        real*8, intent(in) :: crmda
        real*8, intent(in) :: crmpbar
        real*8, intent(in) :: da
        real*8, intent(in) :: pbar

        crm = crmda * da + crmpbar * pbar
    end function CRoll

    real*8 function CYaw(pf, cl, bigA) result(cym)
        type(Planform), intent(in) :: pf
        real*8, intent(in) :: cl
        real*8, intent(in) :: bigA(pf%NNodes)

        integer :: i
        integer :: nnodes
        nnodes = pf%NNodes

        cym = cl / 8.0d0 * (6.0d0 * bigA(2) - pf%RollingRate) + &
            & pi * pf%AspectRatio / 8.0d0 * (10.0d0 * bigA(2) - &
            & pf%RollingRate) * bigA(3)
        do i = 4, nnodes
            cym = cym + 0.25d0 * pi * pf%AspectRatio * &
                & (2.0d0 * real(i, 8) - 1.0d0) * bigA(i-1) * bigA(i)
        end do
    end function CYaw

    subroutine ComputeC(pf, c)
        type(Planform), intent(in) :: pf
        real*8, intent(inout) :: c(pf%NNodes, pf%NNodes)

        integer :: i
        integer :: nnodes

        nnodes = pf%NNodes

        ! Compute values for i=1, i=N
        call C1j_Nj(c, pf)

        ! Compute values for i=2 to i=N-1
        do i = 2, nnodes-1
            call Cij(c, i, pf)
        end do
    end subroutine ComputeC

    subroutine ComputeCInverse(pf, c_inv)
        type(Planform), intent(in) :: pf
        real*8, intent(inout) :: c_inv(pf%NNodes, pf%NNodes)

        call matinv_gauss(pf%NNodes, pf%BigC, c_inv)
```

```
end subroutine ComputeCInverse

subroutine C1j_Nj(c, pf)
    real*8, dimension(:,:), intent(inout) :: c
    type(Planform), intent(in) :: pf

    integer :: j
    integer :: jsq
    integer :: nnode
    real*8 :: cb0

    nnode = pf%NNodes
    do j = 1, nnode
        jsq = j**2
        c(1, j) = real(jsq, 8)
        c(nnode, j) = real((-1)**(j + 1) * jsq, 8)
    end do

    cb0 = c_over_b(pf, pi)
    if (dabs(cb0) < 1.0d-10) then
        call C1j_Nj_zero_chord(c, pf)
    end if

end subroutine C1j_Nj

subroutine Cij(c, i, pf)
    real*8, dimension(:,:), intent(inout) :: c
    integer, intent(in) :: i
    type(Planform), intent(in) :: pf

    integer :: j
    integer :: nnode
    real*8 :: theta
    real*8 :: cb
    real*8 :: sin_theta
    real*8 :: a0, A, r1, r2, r1y
    real*8 :: n

    nnode = pf%NNodes
    theta = theta_i(i, nnode)
    cb = c_over_b_i(pf, i)
    sin_theta = sin(theta)

    a0 = pf%SectionLiftSlope
    A = pf%AspectRatio

    ! Set the low aspect ratio method parameters
    if (pf%LowAspectRatioMethod == Hodson) then
        r1 = a0 * (A / cb * sin_theta)**exp(-8.0 * A)
        r2 = A * (pi - atan((2.0 * a0) / (pi * A)))
    else if (pf%LowAspectRatioMethod == ModifiedSlender) then
        r1 = a0
        r2 = 0.5 * pi * A
    else if (pf%LowAspectRatioMethod == Kuchemann) then
        n = 1.0 - 0.5 * (1.0 + (a0 / (pi * A))**2)**(-0.25)
        r1 = 2 * n * a0 / (1.0 - pi * n / tan(pi * n))
        r2 = pi * A / (2.0 * n)
    else ! Assume Classical
        r1 = a0
        r2 = pi * A
    end if
```

```fortran

        do j = 1, nnode
            c(i, j) = (4.0d0 / (pf%SectionLiftSlope * cb) * (a0 / r1) + &
                & real(j, 8) / sin_theta * ((pi * A) / r2)) * sin(real(j, 8) * theta)
        end do
    end subroutine Cij

    subroutine C1j_Nj_zero_chord(c, pf)
        real*8, dimension(:,:), intent(inout) :: c
        type(Planform), intent(in) :: pf

        integer :: j, n

        n = pf%NNodes

        if (pf%WingType == Tapered) then
            do j = 1, n
                c(1, j) = 2.0d0 * pf%AspectRatio * (1.0d0 + real(j, 8))
                c(n, j) = real((-1)**(j + 1), 8) * c(1, j)
            end do
        else if (pf%WingType == Elliptic) then
            do j = 1, n
                c(1, j) = c(1, j) + real(j, 8) * pi * &
                    & pf%AspectRatio / pf%SectionLiftSlope
                c(n, j) = c(n, j) + real((-1)**(j + 1) * j, 8) * pi * &
                    & pf%AspectRatio / pf%SectionLiftSlope
            end do
        else if (pf%WingType == Combination) then
            ! TODO: Add code for combination wing type
            do j = 1, n
                c(1, j) = c(1, j) + 4.0d0 * real(j, 8) * &
                    & sqrt(1.0d0 - 2.0d0 * pf%C4) / &
                    & (pf%C3 * pf%C5 * pf%SectionLiftSlope)
                c(n, j) = c(n, j) + 4.0d0 * real((-1)**(j + 1) * j, 8) * &
                    & sqrt(1.0d0 - 2.0d0 * pf%C4) / &
                    & (pf%C3 * pf%C5 * pf%SectionLiftSlope)
            end do
        else
            stop "*** Unknown Wing Type ***"
        end if

    end subroutine C1j_Nj_zero_chord

end module LiftingLineSolver
```

K.6 Output Module (`liftinglineoutput.f90`)

```fortran
module LiftingLineOutput
    use Utilities
    use class_Planform
    use LiftingLineSetters
    use matrix
    implicit none

contains
    subroutine OutputHeader()
        integer :: i

        write(6, ('(80a)')) ("*", i=1, 80)
```

```
        write(6, '(34x, a)') "Pralines v1.0"
        write(6, *)
        write(6, '(28x, a)') "Author: Josh Hodson"
        write(6, '(28x, a)') "Release Date: 20 Nov 2013"
        write(6, *)
        write(6, ('(80a)')) ("*", i=1, 80)
    end subroutine OutputHeader

    subroutine OutputPlanform(pf)
        type(Planform), intent(in) :: pf

        ! Open a clean file for output
        open(unit=10, file=pf%FileName, action='WRITE')

        ! Output the planform summary to output file
        call OutputPlanformSummary(10, pf)

        ! Output C matrix and fourier coefficients to output file
        if (pf%OutputMatrices) then
            call OutputC(10, pf%NNodes, pf%BigC)
            call OutputCInverse(10, pf%NNodes, pf%BigC_Inv)
            call OutputFourierCoefficients(10, pf)
        end if

        ! Close the output file
        close(unit=10)
    end subroutine OutputPlanform

    subroutine OutputLiftCoefficientParameters(u, pf)
        integer, intent(in) :: u  ! Output unit
        type(Planform), intent(in) :: pf

        write(u, '(a)') "Lift Coefficient Parameters:"
        write(u, '(2x, a, f20.15)') "KL    = ", pf%KL
        write(u, '(2x, a, f20.15)') "CL,a  = ", pf%CLa
        write(u, '(2x, a, f20.15)') "EW    = ", pf%EW
        write(u, *)
    end subroutine OutputLiftCoefficientParameters

    subroutine OutputDragCoefficientParameters(u, pf)
        integer, intent(in) :: u  ! Output unit
        type(Planform), intent(in) :: pf

        write(u, '(a)') "Drag Coefficient Parameters:"
        write(u, '(2x, a, f20.15)') "KD    = ", pf%KD
        write(u, '(2x, a, f20.15)') "KDL   = ", pf%KDL
        write(u, '(2x, a, f20.15)') "KDW   = ", pf%KDW
        write(u, '(2x, a, f20.15)') "es    = ", pf%ES
        write(u, *)
    end subroutine OutputDragCoefficientParameters

    subroutine OutputRollCoefficientParameters(u, pf)
        integer, intent(in) :: u  ! Output unit
        type(Planform), intent(in) :: pf

        write(u, '(a)') "Rolling Moment Coefficient Parameters:"
        write(u, '(2x, a, f20.15)') "Cl,da = ", pf%Crm_da
        write(u, '(2x, a, f20.15)') "Cl,pb = ", pf%Crm_pbar
        write(u, *)
    end subroutine OutputRollCoefficientParameters

```

```fortran
    subroutine OutputFlightConditions(pf)
        type(Planform), intent(in) :: pf

        ! Open the file and append flight conditions to end
        open(unit=10, file=pf%FileName, access="append")

        ! Output flight conditions to output file
        call OutputOperatingConditions(10, pf)
        call OutputFlightCoefficients(10, pf)

        ! Close the output file
        close(unit=10)
    end subroutine OutputFlightConditions

    subroutine OutputOperatingConditions(u, pf)
        integer, intent(in) :: u  ! Output unit
        type(Planform), intent(in) :: pf

        write(u, '(a15, 19x, 1x, a1, f20.15, 1x, a)') "Optimum washout", &
            & "=", pf%OptimumWashout1 * 180.0d0 / pi, "degrees (Eq. 1.8.37)"
        if (pf%WashoutDistribution == Optimum) then
            write(u, '(a15, 19x, 1x, a1, f20.15, 1x, a)') "Optimum washout", &
                & "=", pf%OptimumWashout2 * 180.0d0 / pi, "degrees (Eq. 1.8.42)"
        end if
        write(u, '(a28, 6x, 1x, a1, f20.15, 1x, a)') "Washout used in calculations", &
            & "=", pf%Washout * 180.0d0 / pi, "degrees"
        write(u, '(a18, 16x, 1x, a1, f20.15, 1x, a)') &
            & "Aileron deflection", "=", &
            & pf%AileronDeflection * 180.0d0 / pi, "degrees"
        write(u, '(a33, 1x, 1x, a1, f20.15)') &
            & "Steady dimensionless rolling rate", "=", SteadyRollingRate(pf)
        write(u, '(a31, 3x, 1x, a1, f20.15)') &
            & "Dimensionless rolling rate used", "=", pf%RollingRate
        write(u, '(a32, 2x, 1x, a1, f20.15, 1x, a)') &
            & "Root aerodynamic angle of attack", "=", &
            & pf%AngleOfAttack * 180.0d0 / pi, "degrees"
        write(u, *)
    end subroutine OutputOperatingConditions

    subroutine OutputFlightCoefficients(u, pf)
        integer, intent(in) :: u  ! Output unit
        type(Planform), intent(in) :: pf

        write(u, '(a)') "Flight Coefficients:"
        write(u, '(2x, a, f20.15, a)') "CL    = ", pf%CL1, " (Eq. 1.8.24)"
        write(u, '(2x, a, f20.15, a)') "CL    = ", pf%CL2, " (Eq. 1.8.5)"
        write(u, '(2x, a, f20.15, a)') "CDi   = ", pf%CDi1, " (Eq. 1.8.25)"
        write(u, '(2x, a, f20.15, a)') "CDi   = ", pf%CDi2, " (Eq. 1.8.6)"
        write(u, '(2x, a, f20.15, a)') "CDi   = ", pf%CDi3, " (Exact)"
        write(u, '(2x, a, f20.15)') "Croll = ", pf%CRM
        write(u, '(2x, a, f20.15)') "Cyaw  = ", pf%CYM
        write(u, *)
    end subroutine OutputFlightCoefficients

    subroutine OutputPlanformSummary(u, pf)
        integer, intent(in) :: u
        type(Planform), intent(in) :: pf

        character*80 :: fmt_str
        integer :: len_nnodes
```

```
len_nnodes = int(log10(real(pf%NNodes))) + 1
write(fmt_str, '(a,i1,a,i1,a)') "(2x, a15, 11x, 1x, a1, 3x, i", &
    & len_nnodes, ", 1x, a, i", len_nnodes, ",a)"

write(u, '(a)') "Planform Summary:"

! Wing type
write(u, '(2x, a9, 17x, 1x, a1, 3x, a)') "Wing type", "=", &
    & trim(GetWingType(pf))

! Number of nodes
write(u, fmt_str) "Number of nodes", "=", pf%NNodes, " (", &
    & (pf%NNodes + 1) / 2, " nodes per semispan)"

! Section Lift Slope
write(u, '(2x, a26, 1x, a1, f20.15)') &
    & "Airfoil section lift slope", "=", pf%SectionLiftSlope

! Aspect Ratio
write(u, '(2x, a12, 14x, 1x, a1, f20.15)') &
    & "Aspect Ratio", "=", pf%AspectRatio

! Taper Ratio
if (pf%WingType == Tapered) then
    write(u, '(2x, a11, 15x, 1x, a1, f20.15)') &
        & "Taper Ratio", "=", pf%TaperRatio
end if

! Transition from tapered to elliptic
if (pf%WingType == Combination) then
    write(u, '(2x, a20, 6x, 1x, a1, f20.15)') "Transition Point z/b", &
        & "=", pf%TransitionPoint
    write(u, '(2x, a20, 6x, 1x, a1, f20.15)') "Transition Point c/croot", &
        & "=", pf%TransitionChord
end if

! Washout distribution type
if (pf%WingType /= Elliptic) then
    write(u, '(2x, a20, 6x, 1x, a1, 3x, a)') "Washout Distribution", &
        & "=", trim(GetWashoutDistributionType(pf))
end if

! Location of aileron root, tip
write(u, '(2x, a19, 7x, 1x, a1, f20.15)') &
    & "z/b at aileron root", "=", pf%AileronRoot
write(u, '(2x, a18, 8x, 1x, a1, f20.15)') &
    & "z/b at aileron tip", "=", pf%AileronTip

! Flap fraction at aileron root, tip
write(u, '(2x, a20, 6x, 1x, a1, f20.15)') &
    & "cf/c at aileron root", "=", pf%FlapFractionRoot
write(u, '(2x, a19, 7x, 1x, a1, f20.15)') &
    & "cf/c at aileron tip", "=", pf%FlapFractionTip

! Hinge Efficiency Factor
write(u, '(2x, a16, 10x, 1x, a1, f20.15)') &
    & "Hinge Efficiency", "=", pf%HingeEfficiency

! Deflection efficiency factor
```

```
        write(u, '(2x, a21, 5x, 1x, a1, f20.15)') &
            & "Deflection Efficiency", "=", pf%DeflectionEfficiency

        write(u, *)

        call OutputLiftCoefficientParameters(u, pf)
        call OutputDragCoefficientParameters(u, pf)
        call OutputRollCoefficientParameters(u, pf)
    end subroutine OutputPlanformSummary

    subroutine OutputFourierCoefficients(u, pf)
        integer, intent(in) :: u
        type(Planform), intent(in) :: pf

        integer :: i

        write(u, '(a)') "Fourier Coefficients:"
        write(u, '(a3, 4(2x, a20))') &
            & "i", "a(i)", "b(i)", "c(i)", "d(i)"
        do i = 1, pf%NNodes
            write(u, '(i3, 4(2x, f20.15))') &
                & i, pf%a(i), pf%b(i), pf%c(i), pf%d(i)
        end do
        write(u, *)
    end subroutine OutputFourierCoefficients

    subroutine OutputC(u, nnodes, c)
        integer, intent(in) :: u
        integer, intent(in) :: nnodes
        real*8, intent(in) :: c(nnodes, nnodes)

        write(u, *) "[C] Matrix:"
        call printmat(u, nnodes, nnodes, c)
        write(u, *)

    end subroutine OutputC

    subroutine OutputCInverse(u, nnodes, c_inv)
        integer, intent(in) :: u
        integer, intent(in) :: nnodes
        real*8, intent(in) :: c_inv(nnodes, nnodes)

        write(u, *) "[C]^-1 Matrix:"
        call printmat(u, nnodes, nnodes, c_inv)
        write(u, *)

    end subroutine OutputCInverse

    subroutine OutputHingeLine(u, pf)
        integer, intent(in) :: u
        type(Planform), intent(in) :: pf

        integer :: i

        do i = 1, pf%NNodes
            write(u, '(i3, 2x, f20.15, 2x, f20.15)') i, z_over_b_i(i, pf%NNodes), y_i(pf, i)
        end do
    end subroutine OutputHingeLine

    subroutine PlotPlanform(pf)
```

```
        type(Planform), intent(in) :: pf

        integer :: i

        ! Generate temporary text file for plotting
        open(unit=11, file='.\Output\planform.dat')
        write(11, '(a)') "$ Planform Geometry"

        ! Write data points for planform
        write(11, '(a)') "! Wing"
        do i=1, pf%NNodes
            write(11, '(f22.15, a, 2x, f22.15)') &
                & z_over_b_i(i, pf%NNodes), ";", 0.25d0 * c_over_b_i(pf, i)
            write(11, '(f22.15, a, 2x, f22.15)') &
                & z_over_b_i(i, pf%NNodes), ";", -0.75d0 * c_over_b_i(pf, i)
            write(11, '(f22.15, a, 2x, f22.15)') &
                & z_over_b_i(i, pf%NNodes), ";", 0.25d0 * c_over_b_i(pf, i)
        end do
        do i=pf%NNodes, 1, -1
            write(11, '(f22.15, a, 2x, f22.15)') &
                & z_over_b_i(i, pf%NNodes), ";", -0.75d0 * c_over_b_i(pf, i)
        end do
        write(11, '(f22.15, a, 2x, f22.15)') &
            & z_over_b_i(1, pf%NNodes), ";", 0.25d0 * c_over_b_i(pf, 1)

        ! Write data points for right aileron
        write(11, '(a)') "$"
        write(11, '(a)') "! Right Aileron"
        write(11, '(f22.15, a, 2x, f22.15)') pf%AileronRoot, ";", &
            & -0.75d0 * c_over_b_zb(pf, pf%AileronRoot)
        write(11, '(f22.15, a, 2x, f22.15)') pf%AileronRoot, ";", &
            & (-0.75d0 + pf%FlapFractionRoot) * c_over_b_zb(pf, pf%AileronRoot)
        write(11, '(f22.15, a, 2x, f22.15)') pf%AileronTip, ";", &
            & (-0.75d0 + pf%FlapFractionTip) * c_over_b_zb(pf, pf%AileronTip)
        write(11, '(f22.15, a, 2x, f22.15)') pf%AileronTip, ";", &
            & -0.75d0 * c_over_b_zb(pf, pf%AileronTip)

        ! Write data points for left aileron
        write(11, '(a)') "$"
        write(11, '(a)') "! Left Aileron"
        write(11, '(f22.15, a, 2x, f22.15)') -pf%AileronRoot, ";", &
            & -0.75d0 * c_over_b_zb(pf, pf%AileronRoot)
        write(11, '(f22.15, a, 2x, f22.15)') -pf%AileronRoot, ";", &
            & (-0.75d0 + pf%FlapFractionRoot) * c_over_b_zb(pf, pf%AileronRoot)
        write(11, '(f22.15, a, 2x, f22.15)') -pf%AileronTip, ";", &
            & (-0.75d0 + pf%FlapFractionTip) * c_over_b_zb(pf, pf%AileronTip)
        write(11, '(f22.15, a, 2x, f22.15)') -pf%AileronTip, ";", &
            & -0.75d0 * c_over_b_zb(pf, pf%AileronTip)

        ! Close the geometry file
        close(unit=11)

        ! System call to plot planform
        call system('"C:\Program Files (x86)\ESPlot v1.3c\esplot.exe" ' &
            & // '.\Output\planform.dat .\Templates\planform.qtp')
    end subroutine PlotPlanform

    subroutine PlotWashout(pf)
        type(Planform), intent(in) :: pf

        integer :: i
```

```

        open(unit=11, file='.\Output\washout.dat')
        write(11, '(a)') "$ Dimensionless Washout Distribution"

        ! Write washout distribution
        do i = 1, pf%NNodes
            write(11, '(f22.15, a, 2x, f22.15)') &
                & z_over_b_i(i, pf%NNodes), ";", pf%Omega(i)
        end do

        close(unit=11)

        call system('"C:\Program Files (x86)\ESPlot v1.3c\esplot.exe" ' &
            & // '.\Output\washout.dat .\Templates\washout.qtp')
    end subroutine PlotWashout

    subroutine WriteSectionLiftDistribution(pf)
        type(Planform), intent(in) :: pf

        integer :: i
        real*8 :: zb, cl(pf%NNodes)

        call GetLiftDistribution(pf, cl)

        open(unit=11, file='liftdistribution.dat')
        write(11, '(a)') "$ Section Lift Distribution"

        do i = 1, pf%NNodes
            zb = z_over_b_i(i, pf%NNodes)
            write(11, '(f22.15, a, 2x, f22.15)') zb, ";", cl(i)
        end do

        close(unit=11)


    end subroutine

    subroutine PlotSectionLiftDistribution(pf)
        type(Planform), intent(in) :: pf

        call WriteSectionLiftDistribution(pf)
        call system('"C:\Program Files (x86)\ESPlot v1.3c\esplot.exe" ' &
            & // 'liftdistribution.dat .\Templates\liftdistribution.qtp')
    end subroutine PlotSectionLiftDistribution

    subroutine WriteNormalizedLiftCoefficient(pf)
        type(Planform), intent(in) :: pf

        integer :: i
        real*8 :: zb, cb, cl1, cl_over_cl, cl(pf%NNodes)
         ! Don't normalize if CL1 == 0
        if (Compare(pf%CL1, 0.0d0, zero) == 0) then
            cl1 = 1.0d0
        else
            cl1 = pf%CL1
        end if

        call GetLiftDistribution(pf, cl)

        open(unit=11, file='liftcoefficient.dat')
        write(11, '(a)') "$ Normalized Section Lift Coefficient"
```

```
        do i = 1, pf%NNodes
            zb = z_over_b_i(i, pf%NNodes)
            cb = c_over_b_zb(pf, zb)
            if (Compare(cb, 0.0d0, zero) == 0) then
                if (Compare(cl(i), 0.0d0, zero) == 0) then
                    if (pf%WingType == Elliptic) then
                        cl_over_cl = NLC_ZeroChord_Elliptic(pf, zb, cb, cl1)
                    else if (pf%WingType == Tapered) then
                        cl_over_cl = NLC_ZeroChord_Tapered(pf, zb, cb, cl1)
                    else if (pf%WingType == Combination) then
                        cl_over_cl = NLC_ZeroChord_Tapered(pf, zb, cb, cl1)
                    else
                        stop "***Unknown Wing Type***"
                    end if
                else
                    ! Finite lift from zero-chord section, should never happen
                    cl_over_cl = 1.0d0 / zero
                end if
            else
                cl_over_cl = cl(i) / cb / cl1
            end if
            write(11, '(f22.15, a, 2x, f22.15)') zb, ";", cl_over_cl
        end do

        close(unit=11)
    end subroutine

    subroutine PlotNormalizedLiftCoefficient(pf)
        type(Planform), intent(in) :: pf

        call WriteNormalizedLiftCoefficient(pf)
        call system('"C:\Program Files (x86)\ESPlot v1.3c\esplot.exe" ' &
            & // 'liftcoefficient.dat .\Templates\liftcoefficient.qtp')
    end subroutine PlotNormalizedLiftCoefficient

    subroutine GetLiftDistribution(pf, cl)
        type(Planform), intent(in) :: pf
        real*8, intent(out) :: cl(pf%NNodes)

        integer :: i, j
        real*8 :: zb, theta

        do i = 1, pf%NNodes
            zb = z_over_b_i(i, pf%NNodes)
            theta = theta_zb(zb)
            cl(i) = 0.0d0
            do j = 1, pf%NNodes
                cl(i) = cl(i) + pf%BigA(j) * sin(real(j, 8) * theta)
            end do
            cl(i) = cl(i) * 4.0d0
        end do
    end subroutine GetLiftDistribution

    real*8 function NLC_ZeroChord_Elliptic(pf, zb, cb, cl) result(cl_over_cl)
        type(Planform), intent(in) :: pf
        real*8, intent(in) :: zb
        real*8, intent(in) :: cb
        real*8, intent(in) :: cl

        integer :: i
```

```fortran
        real*8 :: theta

        theta = theta_zb(zb)
        cl_over_cl = 0.0d0
        do i = 1, pf%NNodes
            cl_over_cl = cl_over_cl + real(i, 8) * pf%BigA(i) * &
                & cos(real(i, 8) * theta) / cos(theta)
        end do
        cl_over_cl = cl_over_cl * pi * pf%AspectRatio / cl
    end function NLC_ZeroChord_Elliptic

    real*8 function NLC_ZeroChord_Tapered(pf, zb, cb, cl) result(cl_over_cl)
        type(Planform), intent(in) :: pf
        real*8, intent(in) :: zb
        real*8, intent(in) :: cb
        real*8, intent(in) :: cl

        integer :: i
        real*8 :: theta, cb2

        if (zb < 0) then
            theta = 1.0d-5
        else
            theta = pi - 1.0d-5
        end if
        cb2 = c_over_b(pf, theta)
        cl_over_cl = 0.0d0
        do i = 1, pf%NNodes
            cl_over_cl = cl_over_cl + pf%BigA(i) * sin(real(i, 8) * theta)
        end do
        cl_over_cl = 4.0d0 * cl_over_cl / cb2 / cl
    end function NLC_ZeroChord_Tapered
end module LiftingLineOutput
```

K.7 Matrix Solver Module (`matrix.f90`)

```fortran
module matrix
    implicit none

contains
    subroutine matinv_gauss(n, mat, mat_inv)
        integer, intent(in) :: n
        real*8, intent(in) :: mat(n,n)
        real*8, intent(out) :: mat_inv(n,n)

        real*8 :: b(n, n), c, d, temp(n)
        integer :: i, j, k, m, imax(1), ipvt(n)

        b = mat
        ipvt = (/ (i, i=1, n) /)

        do k = 1, n
            imax = maxloc(abs(b(k:n, k)))
            m = k - 1 + imax(1)

            if (m /= k) then
                ipvt( (/m, k/) ) = ipvt( (/k, m/) )
                b( (/m, k/), :) = b( (/k, m/), :)
            end if

```

```fortran
            d = 1.0d0 / b(k, k)

            temp = b(:, k)
            do j = 1, n
                c = b(k, j) * d
                b(:, j) = b(:, j) - temp * c
                b(k, j) = c
            end do
            b(:, k) = temp * (-d)
            b(k, k) = d
        end do

        mat_inv(:, ipvt) = b
    end subroutine matinv_gauss


    subroutine printmat(u, m, n, mat)
        integer, intent(in) :: u
        integer, intent(in) :: m
        integer, intent(in) :: n
        real*8, intent(in) :: mat(m, n)

        integer :: i, j
        character*80 :: format_string

        if (u == 6) then
            write(format_string, '(a, i10, a)') "(", n, "(F9.5, 2x))"
        else
            write(format_string, '(a, i10, a)') "(", n, "(F24.15, 2x))"
        end if

        do i = 1, m
            write(u, format_string) (mat(i, j), j=1, n)
        end do
    end subroutine printmat

end module matrix
```

K.8 Utilities Module (`utilities.f90`)

```fortran
module Utilities
    implicit none

    real*8, parameter :: pi = acos(-1.0d0)
    real*8, parameter :: zero = 1.0d-10

contains
    integer function Compare(a, b, tol) result(eq)
    ! Comparison function
    ! Inputs:
    !   a = First argument to compare
    !   b = Second argument to compare
    !   tol = Relative tolerance for comparison
    ! Return Value (eq):
    !   -1 = a < (b - tol)
    !    0 = a == b (within tolerance)
    !    1 = a > (b + tol)
        real*8, intent(in) :: a, b, tol

        if (abs(a) < tol .and. abs(b) < tol) then
```

```
            eq = 0
        else if (abs(a - b) / max(abs(a), abs(b)) < tol) then
            eq = 0
        else if (a < b) then
            eq = -1
        else
            eq = 1
        end if
    end function Compare

    real*8 function Residual(oldVal, newVal) result(res)
        real*8, intent(in) :: oldVal
        real*8, intent(in) :: newVal

        res = dabs(oldVal - newVal) / max(dabs(oldVal), dabs(newVal), zero)
    end function Residual

    integer function CompareFiles(a, b) result(badline)
        character*80, intent(in) :: a, b  ! Filenames of files to compare

        integer :: i, ios1, ios2
        character*5000 :: results_line, work_line

        open(unit=11, file=a)
        open(unit=12, file=b)

        badline = 0
        ios1 = 0
        i = 0
        do while (badline == 0 .and. ios1 == 0)
            i = i + 1

            results_line(1:5000) = " "
            read(11, '(A)', iostat=ios1, end=99) results_line

            work_line(1:5000) = " "
            read(12, '(A)', iostat=ios2, end=99) work_line

            if (ios1 == 0) then
                if (work_line /= results_line) then
                    badline = i
                end if
            else if (len(trim(work_line)) /= 0) then
                badline = i
            end if
        end do

        close(unit=11)
        close(unit=12)
99  continue
    end function CompareFiles

    character*2 function GetCharacterInput(def) result(inp)
        character*2, intent(in) :: def  ! Default value if invalid input

        integer :: i
        character :: a

        read(5, '(a)') inp
        if (len(inp) > 2 .or. len(inp) < 1) then
            inp = def
```

```
        else
            do i = 1, 2
                a = inp(i:i)
                if(iachar(a) >= iachar('a') .and. iachar(a) <= iachar('z')) then
                    inp(i:i) = char(iachar(a) - 32)
                end if
            end do
        end if
    end function GetCharacterInput

    character*80 function GetStringInput(def) result(inp)
        character*80, intent(in) :: def  ! Default value if invalid input

        integer :: i, ios
        character :: a

        read(5, '(a)', iostat=ios) inp
        if (len(trim(inp)) < 1 .or. ios /= 0) then
            inp = def
        end if
    end function GetStringInput

    integer function GetIntInput(mn, mx, def) result(inp)
        integer, intent(in) :: mn   ! Minimum accepted value
        integer, intent(in) :: mx   ! Maximum accepted value
        integer, intent(in) :: def  ! Default value if invalid input

        logical :: cont
        character*80 :: inp_str
        integer :: ios
        integer :: len_mn, len_mx
        character*80 :: msg_fmt

        cont = .true.
        do while (cont)
            read(5, '(a)', iostat=ios) inp_str
            if (ios == 0 .and. trim(inp_str) /= "") then
                read(inp_str, *, iostat=ios) inp
                if (ios /= 0 .or. inp < mn .or. inp > mx) then
                    len_mn = int(log10(real(abs(mn)))) + 1
                    if (mn < 0) len_mn = len_mn + 1

                    len_mx = int(log10(real(abs(mx)))) + 1
                    if (mx < 0) len_mx = len_mx + 1

                    write(msg_fmt, '(a, i1, a, i1, a)') "(a, a, i", len_mn, &
                        & ", a, i", len_mx, ", a)"

                    write(6, *)
                    write(6, msg_fmt) "Invalid input. Please ", &
                        & "specify an integer between ", mn, " and ", mx, ","
                    write(6, '(a)') "or press <ENTER> to accept the default value."
                else
                    cont = .false.
                end if
            else
                inp = def
                cont = .false.
            end if
        end do
```

```
    end function GetIntInput

    real*8 function GetRealInput(mn_orig, mx_orig, dflt_orig) result(inp)
        real*8, intent(in) :: mn_orig   ! Minimum accepted value
        real*8, intent(in) :: mx_orig   ! Maximum accepted value
        real*8, intent(in) :: dflt_orig ! Default value for input

        logical :: cont
        character*80 :: inp_str
        integer :: ios
        integer :: len_mn, ndec_mn
        integer :: len_mx, ndec_mx
        character*80 :: msg_fmt
        real*8 :: mn, mx, dflt

        if (Compare(mn_orig, 0.0d0, zero) == 0) then
            mn = 0.0d0
        else
            mn = mn_orig
        end if

        if (Compare(mx_orig, 0.0d0, zero) == 0) then
            mx = 0.0d0
        else
            mx = mx_orig
        end if

        if (Compare(dflt_orig, 0.0d0, zero) == 0) then
            dflt = 0.0d0
        else
            dflt = dflt_orig
        end if

        cont = .true.
        do while (cont)
            read(5, '(a)', iostat=ios) inp_str
            if (ios == 0 .and. trim(inp_str) /= "") then
                ios = ParseFormula(trim(inp_str), inp)
                write(*,*) ios, inp_str, inp
                if (ios /= 0 .or. inp < mn .or. inp > mx) then
                    write(6, *)
                    write(6, '(a, a, a, a, a, a)') "Invalid input. Please ", &
                        & "enter a number between ", trim(FormatReal(mn, 5)), &
                        & " and ", trim(FormatReal(mx, 5)), ","
                    write(6, '(a)') "or press <ENTER> to accept the default value."
                else
                    cont = .false.
                end if
            else
                inp = dflt
                cont = .false.
            end if
        end do
    end function GetRealInput

    integer function ParseFormula(inp_str, num) result(estat)
        character(len=*), intent(in) :: inp_str
        real*8, intent(out) :: num

        integer :: i, j, n_oper, last_ind, strlen, ios
```

```fortran
        character*40 :: operators, temp_num
        real*8, Dimension(41) :: numbers

        estat = 0
        strlen = len(trim(inp_str))
        n_oper = 0
        last_ind = 0
        do i = 2, strlen
            if (inp_str(i:i) == '*' .or. inp_str(i:i) == '/') then
                n_oper = n_oper + 1
                operators(n_oper:n_oper) = inp_str(i:i)
                temp_num = "                                        "
                temp_num(1:i-last_ind-1) = inp_str(last_ind+1:i-1)
                if ((temp_num(1:1) == 'P' .or. temp_num(1:1) == 'p') .and. &
                    & (temp_num(2:2) == 'I' .or. temp_num(2:2) == 'i')) then
                    numbers(n_oper) = pi
                else
                    read(temp_num, *, iostat=ios) numbers(n_oper)
                    if (ios /= 0) then
                        estat = 1
                    end if
                end if
                last_ind = i
            end if
        end do

        temp_num = "                                        "
        temp_num(1:strlen-last_ind) = inp_str(last_ind+1:strlen)
        if ((temp_num(1:1) == 'P' .or. temp_num(1:1) == 'p') .and. &
            & (temp_num(2:2) == 'I' .or. temp_num(2:2) == 'i')) then
            numbers(n_oper + 1) = pi
        else
            read(temp_num, *, iostat = ios) numbers(n_oper + 1)
            if (ios /= 0) then
                estat = 1
            end if
        end if

        num = numbers(1)
        do i = 1, n_oper
            if (operators(i:i) == '*') then
                num = num * numbers(i + 1)
            else if (operators(i:i) == '/') then
                num = num / numbers(i + 1)
            else
                estat = 2
            end if
        end do
    end function ParseFormula

    recursive character*80 function FormatReal(r, ndigits) result(real_str)
        real*8, intent(in) :: r
        integer, intent(in) :: ndigits

        integer :: order, width, ndecimal
        character*80 :: real_fmt
        real*8 :: r_div_pi
        integer :: num, denom

        if (Compare(r, 0.0d0, zero) /= 0 .and. (IsFactorOfPi(r, ndigits) &
            & .or. IsFractionOfPi(r, num, denom))) then
```

```fortran
            if (Compare(r, pi, zero) == 0) then
                write(real_str, '(a)') "PI"
            else if (IsFractionOfPi(r, num, denom)) then
                if (denom == 1) then
                    write(real_str, '(a, a)') trim(FormatInteger(num)), "*PI"
                else
                    write(real_str, '(a, a, a, a)') trim(FormatInteger(num)), &
                        & "/", trim(FormatInteger(denom)), "*PI"
                end if
            else
                r_div_pi = r / pi
                write(real_str, '(a, a)') trim(FormatReal(r_div_pi, ndigits)), &
                    & "*PI )"
            end if
        else
            if (Compare(r, 0.0d0, zero) == 0) then
                order = 1
            else
                ! Determine the location of the first non-zero digit in the number
                order = int(log10(real(abs(r), 8))) + 1
            end if

            ! Check for sizes that should use exponential format
            if (order <= -4 .or. order >= ndigits) then
                if (r < 0.0d0) then
                    width = ndigits + 6  ! e.g. -1.2345E+67 - 5 digits + 6 other
                else
                    width = ndigits + 5  ! e.g. 1.2345E-67 - 5 digits + 5 other
                end if

                write(real_fmt, '(a, i2, a, i2, a)') "(ES", width, ".", &
                    & ndigits - 1, ")"
            else
                if (r < 0.0d0) then
                    width = ndigits + 2  ! e.g. -12.345 - 5 digits + 2 other
                else
                    width = ndigits + 1  ! e.g. 123.45 - 5 digits + 1 other
                end if

                if (order <= 0) then
                    width = width - order + 1  ! e.g. -0.012345 - additional for leading 0
                end if

                write(real_fmt, '(a, i2, a, i2, a)') "(F", width, ".", &
                    & ndigits - order, ")"
            end if
            write(real_str, real_fmt) r
        end if
    end function FormatReal

    character*80 function FormatInteger(i) result(int_str)
        integer, intent(in) :: i

        integer :: len_i
        character*80 :: int_fmt

        if (i == 0) then
            len_i = 1
        else
```

```
            len_i = int(log10(real(abs(i)))) + 1
            if (i < 0) then
                len_i = len_i + 1
            end if
        end if

        write(int_fmt, '(a, i2, a)') "(i", len_i, ")"
        write(int_str, int_fmt) i
    end function FormatInteger

    logical function IsFactorOfPi(r, ndigits) result(isFactor)
        real*8, intent(in) :: r
        integer, intent(in) :: ndigits

        real*8 :: rx, rx_trunc

        rx = r / pi * 10**ndigits
        rx_trunc = real(int(rx), 8)

        if (Compare(rx, rx_trunc, zero) == 0) then
            isFactor = .true.
        else
            isFactor = .false.
        end if
    end function IsFactorOfPi

    logical function IsFractionOfPi(r, num, denom) result(isFraction)
        real*8, intent(in) :: r
        integer, intent(out) :: num
        integer, intent(out) :: denom

        integer :: i, num2, denom2
        real*8 :: r_div_pi, r_div_pi_i

        isFraction = .false.
        r_div_pi = r / pi
        do i = 1, 360
            r_div_pi_i = r_div_pi * real(i, 8)
            if (Compare(r_div_pi_i , real(int(r_div_pi_i), 8), zero) == 0) then
                num = int(r_div_pi_i)
                denom = i
                isFraction = .true.
                return
            end if
        end do
    end function IsFractionOfPi

end module Utilities
```

L PROOF OF EQUATION (5.2.5)

The derivation of the induced velocity at a point P due to a single straight vortex segment has been presented in Sec. 5.2. Eq. (5.2.5) gives the solution to the integral of the differential velocity over the vortex filament $\overline{OR}$ (see Figure 5.2). Here we present the complete proof to this integral.

The first equality in Eq. (5.2.5) gives the integral to be solved, specifically

$$\mathbf{V} = \frac{\Gamma\left(\mathbf{l}\times\mathbf{r}_1\right)}{4\pi}\int_0^1 \frac{d\zeta}{\left(r_1^2 - 2\zeta\mathbf{r}_1\cdot\mathbf{l} + \zeta^2 l^2\right)^{3/2}} \tag{L.1}$$

To solve this integral we apply u-substitution, with

$$u = \left(r_1^2 - 2\zeta\mathbf{r}_1\cdot\mathbf{l} + \zeta^2 l^2\right)^{1/2} \tag{L.2}$$

$$du = \frac{-\mathbf{r}_1\cdot\mathbf{l} + \zeta l^2}{u}d\zeta \tag{L.3}$$

This gives for Eq. (L.1)

$$\mathbf{V} = \frac{\Gamma\left(\mathbf{l}\times\mathbf{r}_1\right)}{4\pi}\int_0^1 \frac{du}{u^2\left(-\mathbf{r}_1\cdot\mathbf{l} + \zeta l^2\right)} \tag{L.4}$$

We now look for an appropriate expression v so that we can apply the quotient rule of differentiation, namely

$$\frac{v}{u} = \int \frac{udv - vdu}{u^2} \tag{L.5}$$

where

$$udv - vdu = \frac{du}{\left(-\mathbf{r}_1\cdot\mathbf{l} + \zeta l^2\right)} = \frac{d\zeta}{u} \tag{L.6}$$

We try

$$v = c\left(-\mathbf{r}_1\cdot\mathbf{l} + \zeta l^2\right) \tag{L.7}$$

$$dv = cl^2 d\zeta \tag{L.8}$$

where c is a constant found by substitution of Eqs. (L.2), (L.3), (L.7), and (L.8) into Eq. (L.6). This gives

$$c = \frac{1}{r_1^2 l^2 - \left(\mathbf{r}_1\cdot\mathbf{l}\right)^2} \tag{L.9}$$

which suggests that our guess for v is valid since c is indeed constant with respect to ζ. The indefinite integral of Eq. (L.1) is then

$$\frac{\Gamma(\mathbf{l}\times\mathbf{r}_1)}{4\pi}\int\frac{udv-vdu}{u^2}=\frac{\Gamma(\mathbf{l}\times\mathbf{r}_1)}{4\pi}\frac{v}{u}=\frac{\Gamma(\mathbf{l}\times\mathbf{r}_1)}{4\pi\left(r_1^2l^2-(\mathbf{r}_1\cdot\mathbf{l})^2\right)}\frac{-\mathbf{r}_1\cdot\mathbf{l}+\zeta l^2}{\left(r_1^2-2\zeta\mathbf{r}_1\cdot\mathbf{l}+\zeta^2l^2\right)^{1/2}} \tag{L.10}$$

Applying the limits from Eq. (L.1) gives

$$\mathbf{V}=\frac{\Gamma(\mathbf{l}\times\mathbf{r}_1)}{4\pi\left(r_1^2l^2-(\mathbf{r}_1\cdot\mathbf{l})^2\right)}\left[\frac{l^2-\mathbf{r}_1\cdot\mathbf{l}}{\left(r_1^2-2\mathbf{r}_1\cdot\mathbf{l}+l^2\right)^{1/2}}+\frac{\mathbf{r}_1\cdot\mathbf{l}}{r_1}\right] \tag{L.11}$$

We now wish to rewrite this equation in terms of $\mathbf{r}_1$ and $\mathbf{r}_2$ instead of $\mathbf{r}_1$ and $\mathbf{l}$. We use the vector identities $\mathbf{l}=\mathbf{r}_1-\mathbf{r}_2$ and $\mathbf{l}\times\mathbf{r}_1=\mathbf{r}_1\times\mathbf{r}_2$ to get

$$\mathbf{V}=\frac{\Gamma(\mathbf{r}_1\times\mathbf{r}_2)}{4\pi\left(r_1^2(\mathbf{r}_1-\mathbf{r}_2)\cdot(\mathbf{r}_1-\mathbf{r}_2)-\left[\mathbf{r}_1\cdot(\mathbf{r}_1-\mathbf{r}_2)\right]^2\right)}\left[\frac{l^2-\mathbf{r}_1\cdot(\mathbf{r}_1-\mathbf{r}_2)}{r_2}+\frac{\mathbf{r}_1\cdot(\mathbf{r}_1-\mathbf{r}_2)}{r_1}\right] \tag{L.12}$$

Further algebraic manipulation of Eq. (L.12) gives

$$\mathbf{V}==\frac{\Gamma(\mathbf{r}_1\times\mathbf{r}_2)(r_1+r_2)}{4\pi r_1r_2\left(r_1r_2+\mathbf{r}_1\cdot\mathbf{r}_2\right)} \tag{L.13}$$

which is the result given in Eq. (5.2.5).

M TABULATED PROPERTIES OF THE NACA X410 FAMILY OF AIRFOILS

Tables M.1-M.4 present aerodynamic performance characteristics of the NACA X410 family of airfoils computed using XFOIL. The airfoil geometries were modeled using XFOIL's internal NACA airfoil modeler with a grid size of 200 nodes. Nodes were clustered near the leading and trailing edges to obtain higher resolution in areas of large curvature and flow gradients. The analyses were run assuming incompressible flow with a Reynolds number of $Re = 2.4 \times 10^5$. Turbulence in the boundary layer was modeled using XFOIL's turbulence transition model with a value of 2.6 for the *Ncrit* input parameter. This value was estimated based on information provided in a wind tunnel survey of the AFRL Vertical Wind Tunnel (VWT). The data tabulated below were used in the numerical and experimental comparisons discussed in Sec. 5.4.

Table M.1 Airfoil Coefficient Data for the NACA 2410 Airfoil

α (deg)	c_l	c_d (rad^{-1})
-20	-0.4578	0.19659
-19	-0.4441	0.18661
-17.95	-0.4185	0.17502
-16.95	-0.4196	0.16702
-15.95	-0.3829	0.15457
-14.95	-0.3734	0.14418
-13.95	-0.358	0.13471
-12.95	-0.3544	0.1246
-11.95	-0.3844	0.11457
-10.95	-0.3606	0.10158
-9.95	-0.4168	0.08026
-8.9	-0.6626	0.04901
-7.9	-0.6604	0.03277
-6.9	-0.6058	0.02358
-5.9	-0.5249	0.01891
-4.9	-0.4348	0.01624
-3.9	-0.2883	0.0137
-2.85	-0.1259	0.01164
-1.8	0.0218	0.00935
-0.8	0.1708	0.00856
0.15	0.3087	0.00816
1.15	0.4067	0.00817
2.15	0.5054	0.00858

3.2	0.6095	0.00929
4.2	0.7083	0.01014
5.25	0.8103	0.01121
6.25	0.8984	0.01317
7.25	0.9721	0.01722
8.25	1.0462	0.02107
9.25	1.1188	0.02536
10.25	1.1926	0.03082
11.25	1.2514	0.03812
12.25	1.2716	0.0481
13.25	1.214	0.06136
14.25	1.1301	0.08412

Table M.2 Airfoil Coefficient Data for the NACA 4410 Airfoil

α (deg)	c_l	c_d (rad^{-1})
-10	-0.3503	0.11021
-9	-0.3582	0.09857
-8	-0.4013	0.08906
-7	-0.3943	0.07846
-6	-0.2827	0.02824
-5	-0.1411	0.01988
-4	0.0082	0.0158
-3	0.1372	0.01314
-2	0.2545	0.01145
-0.95	0.3644	0.00937
0	0.4777	0.00853
1	0.5842	0.00884
2	0.6907	0.00941
3	0.7966	0.01014
4	0.9022	0.01097
5	1.0061	0.01186
6	1.1027	0.0127
7	1.1945	0.01376
8	1.258	0.01667
9.05	1.2808	0.02292
10.05	1.304	0.0285
11.05	1.3313	0.03488
12.05	1.368	0.04179

13.05	1.4054	0.04913
14.05	1.4185	0.05986
15.05	1.3891	0.07539
16.05	1.317	0.09885
17.05	1.2122	0.13502

Table M.3 Airfoil Coefficient Data for the NACA 6410 Airfoil

α (deg)	c_l	c_d (rad^{-1})
-10	-0.2956	0.12505
-9	-0.2606	0.11127
-8	-0.1966	0.09564
-7	-0.1127	0.07956
-6	-0.0078	0.06244
-5	0.1103	0.0435
-3.9	0.2533	0.01859
-2.9	0.3706	0.01451
-1.9	0.4829	0.01268
-0.9	0.5939	0.01163
0.05	0.6974	0.01096
1.1	0.8051	0.01
2.1	0.9118	0.01072
3.1	1.019	0.01159
4.1	1.1249	0.01253
5.1	1.2305	0.01356
6.1	1.3345	0.01469
7.2	1.4319	0.01557
8.2	1.5097	0.01661
9.2	1.5531	0.01862
10.2	1.4981	0.02819
11.2	1.4764	0.03854
12.2	1.4658	0.04972
13.2	1.4592	0.06147

Table M.4 Airfoil Coefficient Data for the NACA 8410 Airfoil

α (deg)	c_l	c_d (rad^{-1})
-10	-0.0797	0.11796
-8.98	-0.0003	0.10197
-7.98	0.0609	0.09024
-6.98	0.1152	0.07774
-5.98	0.1736	0.06648
-4.97	0.2692	0.0537
-3.97	0.3837	0.04202
-2.96	0.5574	0.02251
-1.95	0.6862	0.01502
-0.95	0.7981	0.01335
0.04	0.9052	0.01285
1.05	1.0128	0.01285
2.06	1.1137	0.01225
3.06	1.2194	0.01322
4.06	1.325	0.01428
5.06	1.4289	0.01542
6.06	1.5333	0.01667
7.06	1.6341	0.01806
8.06	1.7177	0.0191
9.06	1.7757	0.02009
10.06	1.791	0.0221
11.06	1.7452	0.03019
12.07	1.6346	0.04922
13.07	1.5755	0.06769
14.07	1.5411	0.08482
15.07	1.5224	0.09989
16.07	1.5298	0.11027
17.07	1.565	0.11521
18.07	1.5887	0.12357
19.07	1.6011	0.13384

CURRICULUM VITAE

Joshua D. Hodson
July 2019

EDUCATION

Ph.D. Mechanical Engineering, *Utah State University*, Jun. 2019 (3.96 GPA)
Numerical Analysis and Spanwise Shape Optimization for Finite Wings of Arbitrary Aspect Ratio

M.S. Mechanical Engineering, *Utah State University*, Aug. 2007 (3.96 GPA)
RANS Modeling of Nuclear Reactor Lower Plenum Geometries

B.S. Mechanical Engineering, *Utah State University*, Aug. 2007 (3.77 GPA, *Cum Laude*)

TEACHING AND PROFESSIONAL EXPERIENCE

03/18 – present	**Aerospace Research Engineer**, *Air Force Research Laboratory*
12/06 – 02/18	**Sr. Principle Mechanical Engineer (Analysis)**, *Orbital ATK Flight Systems*
05/17 – 08/17	**Graduate Research Assistant**, *Air Force Research Laboratory*
01/17 – 05/17	**Teaching Assistant (Computational Fluid Dynamics)**, *Utah State University*
08/16 – 12/16	**Instructor (Aerodynamics)**, *Utah State University*
08/13 – 12/13	**Teaching Assistant (Thermodynamics II)**, *Utah State University*
05/05 – 12/06	**Graduate Research Assistant**, *Utah State University*
05/04 – 04/05	**Mechanical Engineer Intern (Design)**, *Space Dynamics Laboratory*
01/04 – 05/04	**Math Tutor**, *Utah State University*

JOURNAL PUBLICATIONS

[1] D. Hunsaker, J. Taylor, J. Hodson, and O. Pope, "Aerodynamic Centers of Arbitrary Airfoils below Stall," *J. Aircraft*, accepted for publication.

[2] B. Limb, D. Work, J. Hodson, and B. Smith, "The Inefficacy of Chauvenet's Criterion for Elimination of Data Points," *J. Fluids Eng.*, Vol. 139, No. 5, Jun. 2016.

[3] J. Hodson, R. Spall, and B. Smith, "Turbulence Model Assessment for Flow across a Row of Confined Cylinders," *Nuclear Technology*, Vol. 161, pp. 268-276, 2008.

CONFERENCE PRESENTATIONS AND PAPERS

[1] J. Hodson, G. Reich, J. Deaton, A. Pankonien, P. Beran, "Aerostructure Design of a 2D Morphing Airfoil in Unsteady Supersonic Flow using Bio-Inspired Evolutionary Design Processes and Low-Fidelity Physics Models," *ASME SMASIS*, Louisville, KY, Sep. 2019, accepted for presentation, SMASIS2019-5598.

[2] B. Bielefeldt, D. Hartl, J. Hodson, G. Reich, P. Beran, A. Pankonien, J. Deaton, "Graph-Based Interpretation of L-System Encodings Toward Aeroelastic Topology Optimization of a Morphing Airfoil in Supersonic Flow," *Proceedings of the ASME Conference on Smart Materials, Adaptive Structures and Intelligent Systems*, Louisville, KY, Sep. 2019, accepted for publication, SMASIS2019-5609.

[3] J. Hodson, A. Christopherson, J. Deaton, A. Pankonien, G. Reich, and P. Beran, "Aeroelastic Topology Optimization of a Morphing Airfoil in Supersonic Flow using Evolutionary Design," *Multidisciplinary Design Optimization, AIAA SciTech Forum*, San Diego, CA, Jan. 2019, AIAA-2019-1466.

[4] D. Hunsaker, O. Pope, J. Hodson, and J. Rosqvist, “Aerodynamic Centers of Arbitrary Airfoils,” *AIAA Aerospace Sciences Meeting, AIAA SciTech Forum*, Kissimmee, FL, Jan. 2018.

[5] R. Spall and J. Hodson, “Educational Results obtained using an Improved Two-Dimensional Panel Method Code in Undergraduate Fluid Dynamics and Aerodynamics Courses,” *ASME Fluids Engineering Division Summer Meeting*, Waikoloa, HI, Jul. 2017, FEDSM2017-69031.

[6] J. Hodson, D. Hunsaker, B. Andrews, and J. Joo, “Experimental Results for a Variable Camber Compliant Wing,” *35th AIAA Applied Aerodynamics Conference, AIAA Aviation Forum*, Denver, CO, Jun. 2017, AIAA-2017-4222.

[7] J. Hodson, D. Hunsaker, and R. Spall, “Wing Optimization using Dual Number Automatic Differentiation in MachUp,” *55th AIAA Aerospace Sciences Meeting, AIAA SciTech Forum*, Grapevine, TX, Jan. 2017, AIAA-2017-0033.

[8] H. Dewey, J. Hodson, and K. Illum, “Asset Monitoring with the Internet of Things,” *Orbital ATK Internal Conference*, Apr. 2016.

[9] J. Hodson, H. Dewey, and D. DeVries, “Individual Motor Monitoring and Data Management Solutions to Solid Rocket Prognostic Health Management,” *JANNAF 43rd Structures and Mechanical Behavior Subcommittee Meeting*, Salt Lake City, UT, Dec. 2015.

[10] J. Hodson, J. Forsmann, and R. Spall, “Dual Number Automatic Differentiation in RELAP5-3D”, *International RELAP5 User Group Meeting*, Idaho Falls, ID, Aug. 2015.

[11] J. Hodson and H. Dewey, “IMLM DAAS Software Design and Implementation,” *Orbital ATK Internal Conference*, Apr. 2015.

[12] J. Hodson and R. Spall, “Dual Number Automatic Differentiation for CFD Optimization,” *Utah Fluids Meeting*, Aug. 2014.

[13] R. Monson and J. Hodson, “Real-Time Model Updates using Sensor Data in IMLM DAAS,” *JANNAF 9th Modeling and Simulation Subcommittee Meeting*, Colorado Springs, CO, Apr. 2013.

[14] J. Hodson, R. Spall, and B. Smith, “RANS Predictions in an Idealized Lower-Plenum Model,” *14th International Conference on Nuclear Engineering*, Jul. 2006. ICONE14-89222.

[15] J. Hodson, E. Thorston, R. Spall, and B. Smith, “CFD Validation of Flow Regimes in an Idealized Lower-Plenum Model,” *ANS Winter Meeting and Technology Expo.*, Washington, D.C., Nov. 2005.