12-2022

Multi-Fidelity Predictions for Control Allocation on the NASA Ikhana Research Aircraft to Minimize Drag

Justice T. Schoenfeld
Utah State University

MULTI-FIDELITY PREDICTIONS FOR CONTROL ALLOCATION ON THE NASA IKHANA RESEARCH AIRCRAFT TO MINIMIZE DRAG

by

Justice T. Schoenfeld

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

of

MASTER OF SCIENCE

in

Mechanical Engineering

Approved:

Douglas F. Hunsaker, Ph.D. Major Professor	Stephen A. Whitmore, Ph.D. Committee Member
Tianyi He, Ph.D. Committee Member	D. Richard Cutler, Ph.D. Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2022

ABSTRACT

Multi-Fidelity Predictions for Control Allocation on the NASA Ikhana Research Aircraft to Minimize Drag

by

Justice T. Schoenfeld, Master of Science
Utah State University, 2022

Major Professor: Douglas F. Hunsaker, Ph.D.
Department: Mechanical and Aerospace Engineering

Camber scheduling can be used by aircraft to minimize drag at various operating conditions during flight. In this work, camber schedules for minimum drag on the NASA Ikhana are obtained over a range of lift coefficients. A modern numerical lifting-line algorithm is used to predict the lift and drag of the aircraft as a function of operating condition and camber. The SLSQP optimization algorithm is used to solve for the camber schedule that minimizes drag for a given operating condition. The process is repeated, varying the number of control sections to evaluate the benefit of additional control sections in minimizing drag on the aircraft. Results show that there are diminishing returns with increased numbers of control sections. For the NASA Ikhana, the limit on the number of control sections added before diminishing results were obtained was found to be 2 control sections. With 2 control sections the NASA Ikhana achieved between a 4.5% and 26.3% reduction in drag for lift coefficients between $0.1 - 0.9$ when compared to the baseline Ikhana with no control sections. Adding an additional 2 control sections reduced the drag by less than 0.75%. Results from the optimization can be used in flight algorithms to schedule camber during flight such that drag and fuel burn are minimized. Results can also be used to inform the design of future aircraft with distributed control surfaces, especially in the growing small unmanned aerial vehicle (UAV) market where many designs are aerodynamically less efficient than commercial and research aircraft, such as the NASA Ikhana.

(103 pages)

PUBLIC ABSTRACT

Multi-Fidelity Predictions for Control Allocation on the NASA Ikhana Research Aircraft to Minimize Drag

Justice T. Schoenfeld

Optimal control settings (camber scheduling) can be used by aircraft to minimize drag at various operating conditions during flight. In this work, camber schedules for minimum drag on the NASA Ikhana are obtained over a range of lift coefficients. A modern numerical lifting-line algorithm is used to predict the lift and drag of the aircraft as a function of operating condition and wing section shape (airfoil camber). The SLSQP optimization algorithm is used to solve for the camber schedule that minimizes drag for a given operating condition. The process is repeated, varying the number of control sections to evaluate the benefit of additional control sections in minimizing drag on the aircraft. Results show that there are diminishing returns with increased numbers of control sections. For the NASA Ikhana, the limit on the number of control sections added before diminishing results were obtained was found to be 2 control sections. With 2 control sections the NASA Ikhana achieved between a 4.5% and 26.3% reduction in drag for lift coefficients between $0.1 - 0.9$ when compared to the baseline Ikhana with no control sections. Adding an additional 2 control sections reduced the drag by less than 0.75%. Results from the optimization can be used in flight algorithms to schedule camber during flight such that drag and fuel burn are minimized. Results can also be used to inform the design of future aircraft with distributed control surfaces, especially in the growing small unmanned aerial vehicle (UAV) market where many designs are aerodynamically less efficient than commercial and research aircraft, such as the NASA Ikhana.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor Dr. Douglas F Hunsaker for the support and guidance he provided throughout this work. I would also like to thank Dr. Jeff Taylor for his guidance throughout my thesis and help with the optimization process, Dr. Zachary Montgomery for his help with the optimization methods used, and Cory Goates for all of his help with and knowledge of MachUpX.

I would also like to give special thanks to my wife for her support not only as I worked on my thesis but in all that I do, and to my family for their support throughout schooling and over the years. I am extremely blessed to have my wife and family in my life and cannot express my gratitude to them enough.

Justice Schoenfeld

CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

α	angle of attack
α_{L0}	zero-lift angle of attack
β	sideslip angle
b	span
c	chord
$\bar{c}$	mean geometric chord
c_{ref}	reference chord length
$c_{root/tip}$	chord at wing root/tip
C_D	drag coefficient
C_{D_0}	drag coefficient at zero lift
C_{D_1}	coefficient on the linear term in the parabolic approximation of the drag coefficient as a function of the lift coefficient
C_{D_2}	coefficient on the quadratic term in the parabolic approximation of the drag coefficient as a function of the lift coefficient
C_L	lift coefficient
$\tilde{C}_L$	section lift coefficient
$C_{L,\alpha}$	lift slope
$C_{L,\text{Desired}}$	lift coefficient specified by user for optimization
C_m	pitching moment coefficient
$C_{m,\alpha}$	pitching moment slope
$C_{m,L0}$	pitching moment coefficient at zero lift
$CG_{x,y,z}$	center of gravity x, y, and z components
CS	control section(s)
δ_h	horizontal stabilizer deflection (mounting angle adjustment)
δ_i	camber at control section i
e	oswald efficiency factor
h	altitude
l_{ref}	reference longitudinal length
L	total wing lift
$\tilde{L}$	local wing section lift
ν	kinematic viscosity
ρ	density
R_A	aspect ratio
S_w	planform reference area
V	velocity
W	weight
$\chi_{x\to y}$	ratio of percent change in C_D between condition x and condition y
$\zeta_{x\to y}$	percent change in C_D from condition x to condition y
z	spanwise location along the wing $(0 \leq \lvert z\rvert \leq b/2)$

CHAPTER 1

INTRODUCTION AND LITERATURE REVIEW

Morphing wing designs offer potential improvements and reduced drag compared to non-morphing designs. Minimizing drag is desirable due to drag's large contribution to fuel burn and detrimental effect on efficiency. In general, morphing refers to the ability to change an aircraft's shape in flight. Articulated flaps are one method of morphing that is commonly seen on both commercial and private aircraft. Conformal flaps are another morphing method which utilize continuous camber, such as with parabolic flaps. The difference between articulating and parabolic flaps can be seen in Fig. 1.1. Continuous camber flaps, such as parabolic flaps, produce smoother airfoil sections than articulated flaps when deflected and are often used in morphing designs. For the purposes of this work, morphing refers to the ability to change camber during flight to minimize drag at any given lift coefficient.

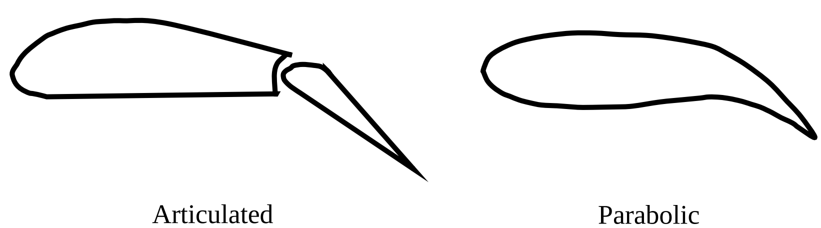

Fig. 1.1: Articulated vs Parabolic flaps.

To examine the effects of morphing designs on reducing drag it is necessary to calculate the forces and moments acting on an aircraft. There are many tools capable of doing this, one such tool is lifting-line theory. Prandtl developed classical lifting-line theory in his 1918 paper [3]. Classical lifting-line as introduced by Prandtl was developed under assumptions of unswept wings with an aspect ratio greater than about 4, with a straight quarter chord, operating in incompressible flow [4]. From lifting-line theory, Prandtl found that an untwisted elliptic lift distribution minimized drag; however, elliptic wings are more costly and time consuming to manufacture than rectangular or tapered wings. By adding linear taper to a rectangular wing Glauert found that at certain taper ratios drag could be reduced to nearly match the elliptic distribution, thus presenting a reasonable alternative to the costly elliptic wing design [5].

Using aerodynamic or geometric twist, the elliptic lift distribution and minimum drag can be achieved with a non-elliptic wing. Geometric twist is defined as spanwise variation in the geometric angle of attack, whereas aerodynamic twist is defined as spanwise variation in the zero-lift angle of attack [4]. Phillips presented an analytical solution for the optimum washout distribution that can be used for wings of any planform and produces the same minimum induced drag as an elliptic wing with the same aspect ratio and no washout [6]. Phillips, Fugal, and Spall verified this solution with Computational Fluid Dynamics (CFD) and showed that through controlling the twist distribution, washout can be optimized to yield a wing of arbitrary planform with the same minimum induced drag as an elliptic wing with the same aspect ratio [7].

Research focusing on the design of the physical methods of morphing technology is ongoing; however, there are currently several successful designs including but not limited to the following: the NASA-Ames/Boeing VCCTEF, the AFRL VCCW, the FlexSys FlexFoil, and the Moulton ARCS and KINCS designs [8–15]. Current morphing wing designs typically utilize aerodynamic twist to adjust the camber of a wing versus changing the geometric twist of the wing. This work will not focus on the physical method of morphing. Instead, this work will focus on using distributed control surfaces with varying camber to present results that highlight the benefits of morphing wings that could be obtained with any of the above morphing technologies.

Regardless of the method used to deflect the wing, there are some physical constraints on the number of locations where a wing can be deflected. Each control section increases the cost and complexity of the design. Every additional control section requires more control mechanisms and parts which will contribute to a weight penalty. More control sections also contribute to more complex control algorithms. All of these considerations contribute to a physical, set limit on the number of control sections that can be used on a morphing design. At the beginning of this work it was hypothesized that there is also a theoretical limit to the number of control sections that can be added to an aircraft before diminishing returns are seen in the drag reduction.

This work examines the effects of wing camber morphing on minimizing drag on the NASA Ikhana aircraft, a variant of the Predator B unmanned aerial vehicle (UAV) shown in Fig. 1.2, through the optimization of aerodynamic twist in a trimmed flight configuration. The objectives of this work include the following:

- Understanding how performance can be improved on the NASA Ikhana air frame by using camber scheduling.
- Understanding how optimal camber scheduling changes with flight condition.
- Understanding how increasing the number of control sections affects the performance of the NASA Ikhana aircraft.

To achieve these objectives, optimization was performed using low fidelity tools which allow for rapid exploration of complex design spaces. This work presents the use of a low fidelity numerical lifting-line tool, MachUpX, combined with a Sequential Least-Squares Quadratic Programming (SLSQP) method to optimize the NASA Ikhana at multiple design conditions with the resultant drag polars and camber schedules. In order to use these lower fidelity tools, code was developed that links the SLSQP optimization method with MachUpX such that the camber of a wing can be varied at multiple control sections in order to minimize drag at a given fixed lift coefficient. This code was then run over a range of lift coefficients in order to produce drag polars and optimal camber schedules.

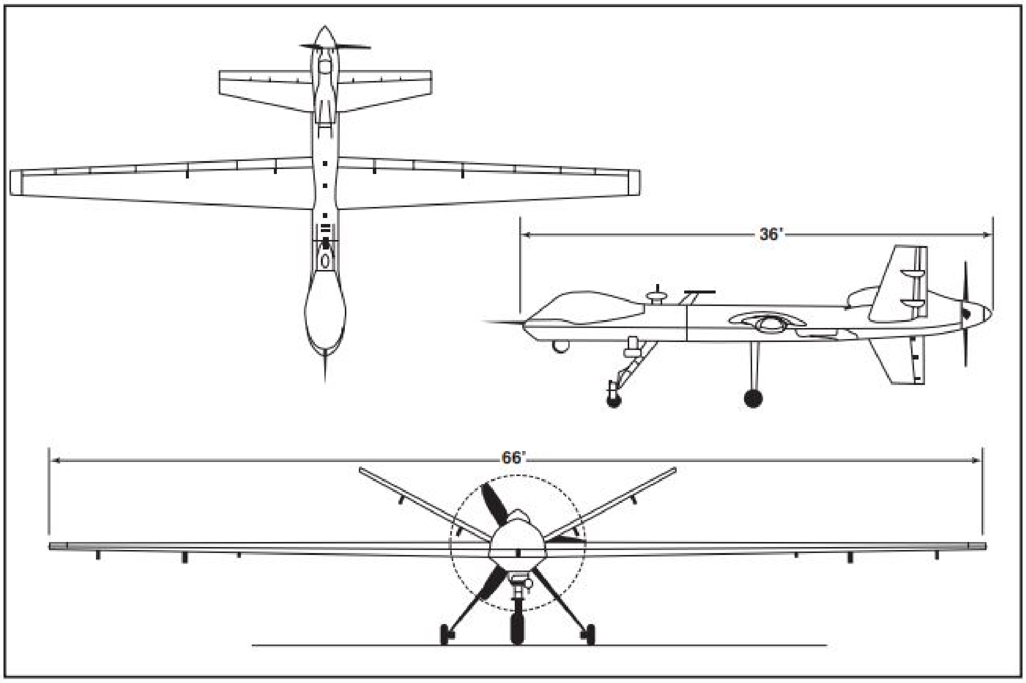

Fig. 1.2: The NASA Ikhana aircraft [2].

CHAPTER 2
COMPUTATIONAL METHODS AND TOOLS

This work required the use of an aerodynamic modeling tool and an optimization tool. This section discusses these tools.

2.1 Aerodynamic Modeling

The aerodynamic modeling in this work utilized a software called MachUpX. MachUpX is an implementation of the Goates-Hunsaker numerical lifting-line method [16]. This method is based on Phillips and Snyder's numerical lifting-line algorithm [17] with corrections to handle singularities introduced in the governing equations when sweep or sideslip are modeled. The numerical lifting-line algorithm used in MachUpX is capable of producing very accurate results, within the limitations of lifting-line theory, without the computational overhead of higher-order methods such as CFD [18]. Phillips and Snyder suggest that lifting-line compares well with experimental data for lifting surfaces with aspect ratios greater than about four [17]. For more on the improvements to handle singularities from sweep and sideslip, see Reid and Hunsaker [19] and Goates and Hunsaker [16].

To aid in further sections, a brief overview of MachUpX is needed. MachUpX relies on two inputs. These inputs are usually JSON files, but can also be represented as python dictionaries, which are direct analogies of a JSON file [18]. The first input is the scene JSON file, which contains information about how many aircraft are in the scene and what the operating conditions and state of each aircraft are. The second file (or files for multiple aircraft) is the aircraft JSON file, which contains all information about the geometry and controls of a single aircraft. The scene JSON file needs to reference each aircraft JSON file associated with the scene. The scene JSON file is what is passed to MachUpX to create the scene class where all of the functionality of MachUpX is located. All analysis is called through the scene class. Other functions such as adding aircraft to the scene, changing aircraft control states, and various other functions as described in the MachUpX documentation [18] are also called through the scene class. Any changes for which there is not an associated function for in the scene class requires the scene class to be reinitialized with a new scene or aircraft JSON file or dictionary. An example of such a change would be to change the twist distribution associated with a given lifting surface.

MachUpX also uses the AirfoilDatabase package to calculate section properties of all airfoils. MachUpX does not have the capability to generate these databases and can only read in from previously generated databases [18]. At a minimum the database must contain information about the values of α_{L0}, $C_{L,\alpha}$, $C_{m,L0}$, $C_{m,\alpha}$, C_{D_0}, C_{D_1}, and C_{D_2} for each airfoil used. These values are then used by MachUpX to calculate the lift, drag, and pitching moment coefficients. The lift coefficient can be found using

$$C_L = C_{L,\alpha}(\alpha - \alpha_{L0}) \tag{2.1}$$

the drag coefficient is calculated using

$$C_D = C_{D_2} * {C_L}^2 + C_{D_1} * C_L + C_{D_0} \tag{2.2}$$

and the pitching moment coefficient using

$$C_m = C_{m,L0} + C_{m,\alpha}(\alpha - \alpha_{L0}) \tag{2.3}$$

MachUpX uses the lift, drag, and moment coefficients of each airfoil to calculate the total forces and moments acting on the aircraft being modeled. This is done by modeling the wing segment with a number of horseshoe vortices. The number of vortices defaults to 40 per semi-span unless otherwise defined. However, increasing the number of vortices can produce better results. In order to determine the number of vortices to use, a grid resolution study was performed. The study was performed with the NASA Ikhana at an angle of attack of 2.5°. The horizontal stabilizer had approximately half the grid density of the main wing for any given grid size. It can be seen in Fig. 2.1 that for grid densities at or above ≈ 100, both the lift coefficient and drag coefficient are well resolved. For this work, a grid density of 100 was used for the main wing and a grid density of 50 was used for the horizontal stabilizer.

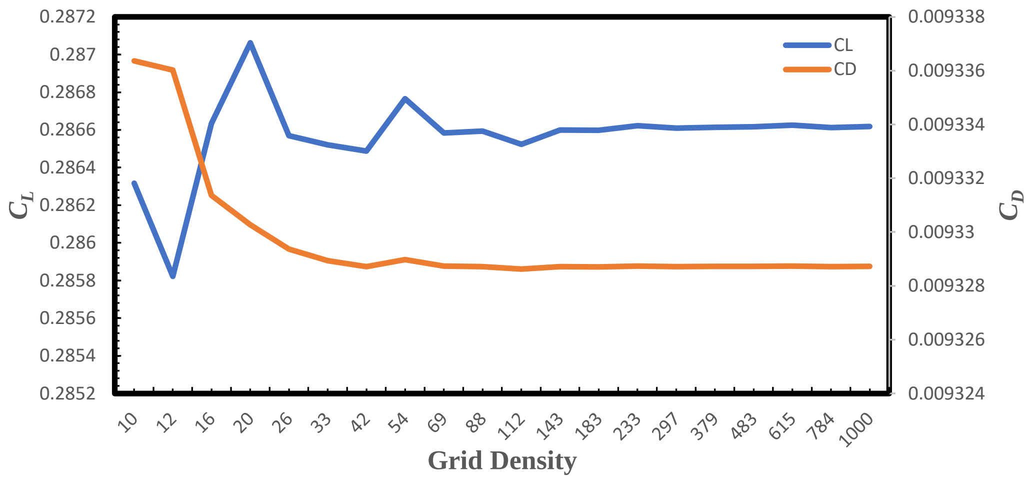

Fig. 2.1: Grid resolution study for the NASA Ikhana at $\alpha = 2.5°$.

The numerical lifting-line algorithm used in MachUpX has been evaluated in various publications [16, 17, 19]. To further evaluate the accuracy of MachUpX when compared to higher fidelity tools such as CFD, the optimization code for this work was used to generate a baseline drag polar (a drag polar with zero control sections where the optimization is only pitch trimming the aircraft) for the NASA Common Research Model (CRM), a transonic passenger jet designed for research purposes [20]. There are multiple CFD results available for the CRM, including those from the 4$^{\text{th}}$ Drag Prediction Workshop (DPW4). CFD results from DPW4 are compared against results obtained using MachUpX in Fig. 2.2. As can be seen, the results obtained with MachUpX compare well with those from DPW4 obtained with CFD, indicating that the results obtained with MachUpX are reasonable and reliable.

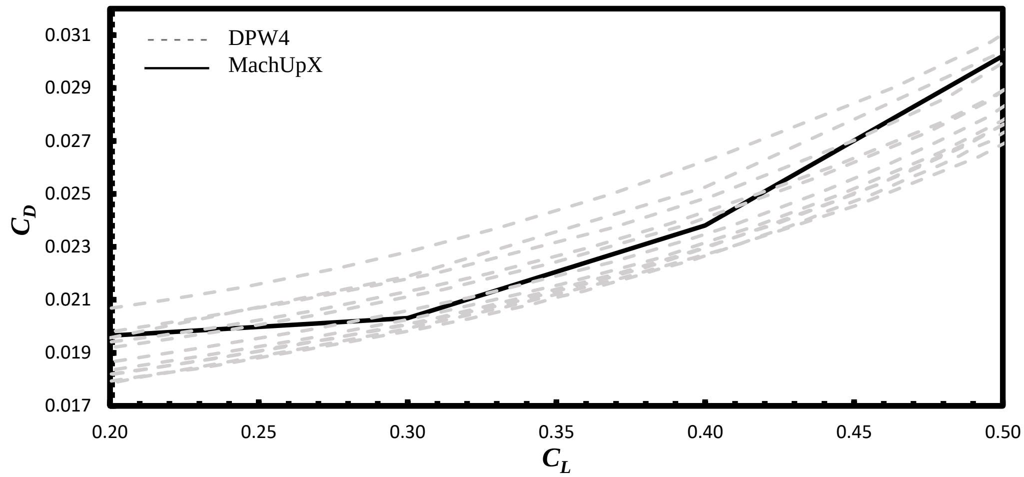

Fig. 2.2: DPW4 CFD results vs MachUpX results for the NASA Common Research Model.

2.2 Optimization Method

The optimization algorithm used in this work was the gradient based SLSQP method implemented in the SciPy software package [21]. SLSQP was chosen due to its ability to handle both bounds and constraints for the optimization of twist to minimize drag. Bounds were used to ensure that any horizontal stabilizer deflections as well as the angle of attack needed to achieve trimmed flight stayed within reasonable and physically achievable values during the optimization process. However, none of the final optimal solutions were constrained by the bounds. Constraints were used to ensure the aircraft was pitch trimmed. It was necessary to move the pitch trim functionality into the optimization due to a limitation encountered with the built in MachUpX pitch trim command. The gradients for the SLSQP were calculated numerically with finite differencing. Meaning, the cost function was called at a given value and then perturbed away from that value to calculate the gradient. The numerical gradient was then used to move in the direction that minimized the cost function.

2.3 Software Versions Used

Table 2.1 shows the software versions for MachUpX, Python, and all tools used for the optimization code in this work.

Table 2.1: Software versions used in this work.

Software Versions used for MachUpX/SLSQP Optimization	
Airfoil_db	v1.4.3
MachUpX	v2.7.1
NumPy	v1.20.3
Python	v3.8.8
SciPy	v1.7.3
Spyder	v5.0.5
XFOIL	v6.99

2.4 Optimization Approach

For this work, the NASA Ikhana was modeled in MachUpX by assuming symmetric deflections. Therefore, only a single semi-span of the wing-tail combination needed to be defined in the aircraft JSON file. MachUpX then used symmetry to model the entire aircraft. The CG, weight, reference area, twist, and operating conditions were all specified in the MachUpX input files with the values in Tables 2.2 - 2.3. The input files required an angle of attack to be specified; however, this value was varied within the optimization routine to trim the aircraft and therefore is not included in Table 2.3. Because the Ikhana has straight tapered wings with no sweep, the number of control sections was evenly divided over the span of the main wing with the first control section defined as that nearest the root of the wing and the last control section defined as the control section nearest the wingtip. All control sections were discrete and all deflections are accomplished with airfoils of varying camber. The baseline airfoil used for the NASA Ikhana was the NACA 0010 airfoil. A database of airfoils with various camber was generated using data from Hunsaker [1] who used the NACA $X410$ series of airfoils, where X represents varying values of maximum camber. A combination of average values, linear, and parabolic fits were applied to the data from Hunsaker to get values for α_{L0}, $C_{L,\alpha}$, $C_{m,L0}$, $C_{m,\alpha}$, C_{D_0}, C_{D_1}, and C_{D_2} as a function of camber. The fits are summarized in Table 2.4.

While MachUpX has an integrated command to pitch trim the aircraft, the pitch trim logic was included in the cost function as a constraint due to limitations with how MachUpX's pitch trim algorithm works. MachUpX's built in pitch trim function uses flap

Table 2.2: Physical properties of the NASA Ikhana.

NASA Ikhana Physical Properties	
Aspect Ratio, R_A	16
$CG_{x,y,z}$ (m)	0.0
Mean chord, $\overline{c}$ (m)	1.2192
Reference Area, S_w (m^2)	23.7832
Reference Longitudinal Length, l_{ref} (m)	1.2192
Span, b (m)	19.5072
Semi-span, $b/2$ (m)	9.7536
Root Chord, c_{root} (m)	1.70688
Tip Chord, c_{tip} (m)	0.73152
Weight, W (N)	31,593

Table 2.3: Operating conditions of the NASA Ikhana.

NASA Ikhana Operating Conditions	
Altitude, h (m)	6,096.0
Density @ SL, ρ (kg/m^3)	1.2250
Kinematic Viscosity @ SL, ν (m^2/s)	$1.5e^{-5}$
Sideslip Angle, β (°)	0.0
Speed, V (m/s)	102.889
Speed of Sound @ SL, C (m/s)	340

Table 2.4: Coefficient fits for the NASA Ikhana as a function of camber from data generated by Hunsaker [1].

NASA Ikhana Airfoil Database Coefficients		
Coefficient	Fit Type	Fit
α_{L0}	Linear	$-0.0183 * \delta_i - 0.0003$
$C_{L,\alpha}$	Average	6.257605
$C_{m,L0}$	Linear	$-0.0253 * \delta_i - 0.0004$
$C_{m,\alpha}$	Average	0.016353333
C_{D_0}	Parabolic	$0.0002 * {\delta_i}^2 - 0.00004 * \delta_i + 0.0049$
C_{D_1}	Linear	$-0.003 * \delta_i + 0.0002$
C_{D_2}	Parabolic	$0.0001 * {\delta_i}^2 - 0.0004 * \delta_i + 0.0095$

deflections rotated about a given chordwise pivot point on the horizontal stabilizer to trim the aircraft. For this work, an all-flying tail configuration was used. There is currently no option for an all-flying tail configuration in MachUpX, and the only way to approximate an all-flying tail is to define the flap on the horizontal stabilizer as the entire horizontal stabilizer. However, defining the entire horizontal stabilizer as a flap results in the horizontal stabilizer being rotated about its leading edge instead of the quarter chord. To address this challenge, the mounting angle of the horizontal tail, δ_h, was included as a design variable for the optimization. Including the mounting angle as a design variable allowed for dynamically changing the mounting angle of the horizontal tail by adding or subtracting the value δ_h to the original twist distribution for the horizontal tail, effectively rotating the stabilizer about the quarter chord. The angle of attack α was also included as a design variable so that it could be adjusted and used to update the state of the aircraft within MachUpX to a trimmed state. By using both δ_h and α as design variables and then placing a constraint on the pitching moment of the aircraft, a trimmed flight condition was achieved for the optimization.

A bound of $|\delta_h| \leq 25°$ was chosen to ensure the horizontal stabilizer mounting angle stayed within typically reasonable deflection angles for a control surface. A bound of $|\alpha| \leq 25°$ was also implemented to ensure that the aircraft would not try to trim at an angle of attack that would cause the wings to stall. While 25° is a relatively large angle of attack, it was found that the final angle of attack used to trim the aircraft stayed well below this bound.

Constraints were used to trim the Ikhana in the optimization as well as to specify a desired lift coefficient. Both of the constraints were implemented as equality constraints based off of the logic contained in the optimization statement below. The lift coefficient and pitching moment values used to satisfy the constraints were generated using a built in MachUpX command, which also calculated the drag coefficient to be minimized.

$$
\text{Optimization Statement} \left\| \begin{array}{ll} \text{minimize:} & C_D * 100 \\ \text{with respect to:} & \alpha,\ \delta_h,\ \delta_i \\ \text{subject to:} & C_L - C_{L,\text{Desired}} = 0 \\ & C_m = 0 \\ & |\delta_h|, |\alpha| \leq 25^\circ \end{array} \right.
$$

To ensure the drag coefficient was minimized appropriately, it was helpful to ensure that the largest constraint and the value to be minimized were on the same order of magnitude. The pitching moment was constrained to be zero, so it was assumed that it would generally be smaller than the drag coefficient value being minimized. However, the lift coefficients used were in the range of $C_L = 0.1 - 0.9$ while the drag coefficients obtained for the NASA Ikhana were in the range of $0.006 - 0.03$ which is 10 to 100 times smaller than C_L. Due to C_D being one to two orders of magnitude smaller than C_L, it was helpful to scale the drag coefficient to be on the same order of magnitude as the largest constraint. This was done by multiplying the C_D value returned from MachUpX in the cost function by 100.0.

To start the optimization, MachUpX was passed the aircraft specific input files, from which a scene class was generated. It is important to note that any changes to the aircraft or scene objects after a scene class has been created, require the scene class to be reinitialized. The scene class contained information about the aircraft state, including the angle of attack which was changed to achieve a trimmed flight condition. The aircraft object contained information about the twist distribution and was updated at each iteration of the optimization. Since both the aircraft and scene objects were updated at each iteration of the optimization it was necessary to reinitialize the scene class at the beginning of each iteration as well.

Once the initial scene class was generated, it was passed to scipy.optimize.minimize. The x array for the scipy optimization contained all design variables, including the values for the camber of each control section, the mounting angle adjustment for the horizontal stabilizer, and the angle of attack $(\delta_i, \delta_h, \alpha)$. Thus the size of the x array was equal to the total number of control sections + 2. Inside of the optimization, the values of the x array associated with the camber of each control section were used to adjust the camber on the main wing of the Ikhana. Then the function to solve for forces and moments in MachUpX was called to generate $C_D, C_L,$ and C_m so that the aircraft could be trimmed and the drag minimized. Trimming was achieved by enforcing the two equality constraints for C_L and C_m and allowing the optimization to adjust the last two elements of the x array (δ_h and α) along with the camber. The state object for the aircraft was then updated and the scene class was reinitialized since there were changes to the aircraft object. After re-initialization of the scene class, the solve forces function was again called to obtain the value of C_D, which was the return value of the cost function in the optimization. This process was repeated until the total drag had been minimized in a trimmed state. Upon completion, the camber, mounting angle adjustment for the horizontal stabilizer, angle of attack, lift coefficient, and moment coefficient $(\delta_i, \delta_h, \alpha, C_L, C_m)$ were returned along with the value of the minimum drag, C_D. A simplified flow chart of the process is depicted in Fig. 2.3.

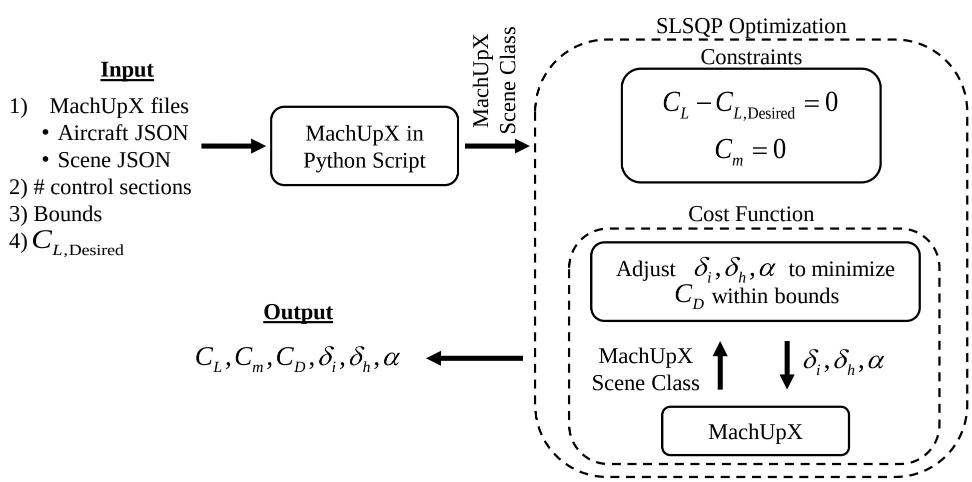

Fig. 2.3: Simplified flow chart of optimization process.

2.5 Example Ikhana Calculation

As an example, this section will go through the optimization method in greater detail with references to the optimization code for the NASA Ikhana used in this work. While this section details the specific code used in this work, care was taken to highlight the most important concepts necessary for any code/method in order to achieve similar results when combining MachUpX with an SLSQP optimization method.

2.5.1 User Input

To set up and initialize the optimization code, several pieces of information were required. An aircraft JSON file and scene JSON file were defined for use in MachUpX. These files were set up as outlined in the MachUpX documentation [18]. The NASA Ikhana aircraft and scene JSON files used in this work are shown in Appendices A.1 and A.2. The number of control sections and the desired lift coefficient were also specified. Lastly, the upper and lower bounds for the range of angles of attack and horizontal stabilizer mounting angles were specified to ensure the solution stayed within the operational envelope of the aircraft to be optimized. For this work, all necessary information was input into a set up/run commands python file and then all proceeding information was passed between functions automatically. An example of this file type is shown in Appendix B.3. The aircraft and scene JSON files, as well as the aircraft name, are given in lines 18-20 of B.3. The lift coefficient to optimize at is given on line 15, the number of control points on line 23, and the upper and lower flap bounds are specified on lines 26 and 27 of B.3. Also, as seen on line 30 of B.3, an initial guess can be specified for the camber, mounting angle adjustment, and angle of attack $(\delta_i, \delta_h, \alpha)$ to be used. An initial guess was not necessary but the ability to define one proved useful, especially when searching different areas of the solution space or when initializing the optimization using a solution from a previous lift coefficient when looping through a range of lift coefficients. Lastly, the end of the user input/run commands file shown in B.3 includes the call to the optimization with all required values passed in as parameters to the optimization call.

2.5.2 General Initialization and Creating MachUpX Scene Class

Once the aircraft JSON file, scene JSON file, number of control sections, upper and lower bounds, and the desired lift coefficient were given to the optimization code, the MachUpX scene class was created. Creating the initial MachUpX scene class was straight forward. However, the method of changing the camber, horizontal tail mounting angle, and angle of attack in order to minimize drag required additional steps. Changing the camber meant changing the airfoil being used and MachUpX needs information about the coefficients for any airfoil used. Changing the horizontal tail mounting angle in a way that rotates the tail about the quarter chord and changing the angle of attack of the aircraft both required the MachUpX scene class to be reinitialized each time a change was made. These extra steps required the use of functional type airfoils, as described in [22], and using dictionaries to represent the information in the aircraft and scene JSON files.

To minimize drag by optimizing the camber schedule, it was necessary to use a wide range of camber values for the airfoils that make up the wing. In MachUpX, each airfoil used must be defined. The definition of an airfoil includes information about the coefficients and geometry of the airfoil. Lift, drag, and pitching moment coefficients, or a way to derive them, must be specified in order to carry out the calculations internal to MachUpX. By allowing the optimization method to change the camber to minimize drag, it was not possible to define every possible airfoil and its associated coefficients. The solution to this problem lay in the ability of MachUpX to use functions to find the lift, drag, and moment coefficients. In order to generate these functions the data from Hunsaker [1] was used to generate a combination of average values, linear fits, and parabolic fits that allowed for the calculation of $C_L, C_D,$ and C_m as a function of the camber. Once the functions for $C_L, C_D,$ and C_m were generated, they were linked to the aircraft information used by MachUpX.

When using functional airfoils, placing the function names in the aircraft JSON file before it is read into MachUpX will not work as the names are no longer interpreted as functions once read in from the JSON file. This challenge was solved by using dictionaries. The aircraft and scene JSON files were read into the code as dictionaries and then the dictionary stored in the aircraft's airfoils key was replaced with a new dictionary that contained the function calls for $C_L, C_D,$ and C_m. For an example of how this was implemented in the present work, see Appendix B.6 for the creation of the new dictionary with the functions, and line 131 in Appendix B.4 for replacement of the airfoils key in the aircraft dictionary.

Lastly, using dictionaries addressed the problem of having to reinitialize the MachUpX scene class whenever δ_h or α were changed. Instead of attempting to overwrite or rewrite the given JSON files with updated values for α and δ_h and then reading the new adjusted JSON file back in, a dictionary allowed for easier access to and changing of the information within the code itself. Using dictionaries also allowed for the use of multiple copies of the aircraft and scene dictionaries in order to maintain the original information as well as a version with any changes needed for the next iteration of the optimization. Maintaining a running copy of the scene and aircraft dictionaries helped address the need to reinitialize the MachUpX scene class whenever a change to δ_h or α were made by splitting out the two processes. Changes to α were made in the scene dictionary, from which a scene class was initialized, while changes to δ_h were made in the aircraft dictionary, which had no effect on the initialization of a scene class. However, the aircraft associated with the scene class needed to be updated once changes were made. To help facilitate this process it was found to be beneficial to remove the aircraft key, and associated information, from the scene dictionary and create the MachUpX scene class with a blank scene dictionary. Initializing with a blank scene dictionary, a scene that contained no aircraft, resulted in a scene where only the operating conditions were specified. Then, once the scene class was created, a MachUpX command that adds an aircraft to the scene class was used. By keeping a running tab of the two dictionaries, a new scene class was created using the scene dictionary whenever there was a change to α. Then the aircraft dictionary was added to the scene class with any changes to δ_h needed to pitch trim the aircraft. See Appendix B.4 lines 125-142, 155-157, 289-302, 380-396 for examples of how the scene and aircraft dictionaries were used to create the scene class, set α, and change δ_h.

2.5.3 Optimization Set Up and Call

In order to set up and carry out the optimization, an initial guess, bounds, and constraints were all required. As mentioned earlier, the initial guess was set up as an array with length equal to the number of control sections + 2. For example, if two control sections were desired, the initial guess array would be 4 elements long: two elements for the control section camber values, 1 element for the horizontal stabilizer mounting angle adjustment, and 1 element for the angle of attack. The initial guess array could be given in two forms. The first assumed 0 for all values of $\delta_i, \delta_h,$ and α. The second used the initial values given by the user in the user input section for the values of $\delta_i, \delta_h,$ and α.

The user-specified values for upper and lower bounds for the angle of attack and horizontal stabilizer mounting angle were used to create bounds using the scipy.optimize.Bounds class. These bounds applied only to the last two elements of the initial guess array, x. This was done so that the values in the initial guess array corresponding to camber had no bounds applied, only the angle of attack and horizontal stabilizer mounting angle had bounds applied. Bounding the angle of attack allowed the user to keep the aircraft within a region that avoided stall and bounds on the horizontal stabilizer kept deflections within the physical limits of the aircraft's control surfaces.

Equality constraints were set up for both the lift coefficient, C_L, and the pitching moment coefficient, C_m. These constraints were evaluated using the same cost function that was used for minimizing drag. For this purpose, a flag was added to the cost function so that the cost function could determine whether to return one of the constraint values, C_L or C_m, or to return the value to be minimized, C_D.

Once the initial guess, bounds, and constraints were set up, the optimization was called. The optimization was called by passing the cost function, the initial guess, the desired lift coefficient, the bounds, and the constraints into scipy.optimize.minimize. See Appendix B.4 lines 212-247 for how this was implemented in this work.

2.5.4 Cost Function

The optimization was dependant on the cost function. Within the cost function, the horizontal stabilizer mounting angle and angle of attack were set according to the values in the x array. Changes were made to the angle of attack and horizontal stabilizer mounting angle in the scene and aircraft dictionaries and then a new scene class was initialized. The values of the x array associated with the camber for each control section were used to set the control state of the aircraft within the scene class. MachUpX was then used to solve for the forces and moments on the aircraft associated with the given operating conditions. Depending on whether the cost function was being called for a constraint or for the value to be minimized, the appropriate value was returned, either $C_L, C_m,$ or C_D.

In order to change the mounting angle of the horizontal stabilizer, the element of the x array corresponding to δ_h was added to each element of the twist array for the horizontal stabilizer. The horizontal stabilizer twist array was a n x 2 array where n represented the number of spanwise locations for which the twist was defined. The first column of the array was the spanwise location and the second column was the twist value associated with that spanwise location along the horizontal stabilizer. By adding δ_h to the twist value of each spanwise location the entire horizontal stabilizer was effectively rotated about its quarter chord. The angle of attack was also updated by using the element of the x array corresponding to α; however, this update was as simple as replacing the value associated with the alpha key in the scene dictionary with the value of α from the x array. Once these steps were accomplished, the MachUpX scene class was reinitialized and the aircraft added to the scene.

MachUpX can accept an array that defines the control state for an aircraft. The first column of this array should contain the spanwise location associated with the edges of the control section, and the second column should specify the deflection, in this case camber, associated with each control surface. In between these specified locations, MachUpX will linearly interpolate the settings. This means that if two control sections are desired, but only values of [0, 0.5, 1] are given in the spanwise location column, the resulting control sections will look like part (a) of Fig. 2.4 as viewed from the trailing edge. It was desired to have each control section represented with a single airfoil of a specific camber. To achieve this result, the array given to MachUpX needed to be 'doubled'. This was done by representing the spanwise location column as [0, 0.5, 0.5, 1.0] where the spanwise location between control surfaces were doubled. Doubling the array resulted in control sections that looked like part (b) of Fig. 2.4 as viewed from the trailing edge. Using this method, the array representing the camber associated with each control section was generated and the control state of the aircraft was set. Finally, the forces and moments were calculated using MachUpX and the appropriate value, C_L, C_m, or C_D was returned. See Appendix B.4 lines 342-424 for the cost function used in this work. It is important to note that modeling the control sections as represented in part (b) of Fig. 2.4 causes a discontinuity between control sections which can cause interference drag. MachUpX neglects the interference drag caused by this discontinuity between control sections when calculating the drag.

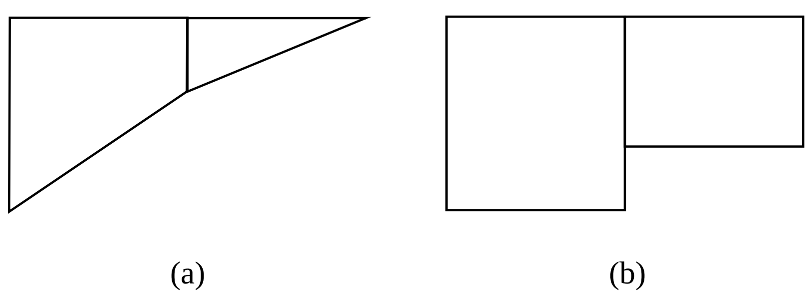

Fig. 2.4: Linearly interpolated (a) vs Discrete (b) control sections.

2.5.5 Final Forces and Moments Solution

Once the optimization section was done, only the final x array was returned to the main body of code. While the final value of minimum drag was contained in the solution object returned from the call to scipy.optimize.minimize, this value was ignored as other features of MachUpX were still needed. Instead, the final x array from the solution contained all information about the angle of attack, horizontal stabilizer mounting angle adjustment, and control section camber values that would give the minimum drag. These values were used to adjust the aircraft and scene dictionaries, reinitialize the MachUpX scene class, and calculate the final forces and moments.

The drag coefficient, angle of attack, horizontal stabilizer mounting angle adjustment, and camber settings were then returned to the user. It was also useful to record all forces and moments as well as the distributions file generated by MachUpX. The distributions file was used to get the section lift coefficient, which was used to generate the lift distribution as a function of spanwise location along the wing for comparison with the elliptic lift distribution. In some instances it was useful to take the final solution from the optimization and plug it back into the optimization as the initial guess before calculating the final forces and moments. By plugging the 'final' solution back into the optimization, comparing the new final solution to the prior final solution, and iteratively doing so until the difference between the two solutions was below a given error threshold, the accuracy with which the true local minimum could be found was increased.

2.5.6 Single C_L Case

For running the optimization at a single lift coefficient, only the above sections were required for the optimization. As mentioned in the preceding section, it was desirable to write the results of the optimization out to files for reference. It was found to be useful to save the final forces and moments, the MachUpX distributions file, and the solution returned from the optimization along with information about the initial conditions. Storing the initial conditions served as a reference for the conditions used to obtain a given result and allowed for that result to be duplicated. Example code can be seen in Appendix B.3

2.5.7 Looping Through a Range of C_L Values

For looping through a range of lift coefficients, $C_L = 0.1 - 0.9$, the process described above was placed in a loop where at each iteration a different lift coefficient was used in the initial parameters. This method also allowed for automatic creation of a drag polar, comparison of camber schedules, and using the solution from a previous lift coefficient as the initial guess for the next lift coefficient. Using the result of the previous lift coefficient as the initial guess for the next lift coefficient was found to shorten run-time and yield better results when compared to using an initial guess of zero camber, zero mounting angle adjustment, and zero angle of attack for the entire range of lift coefficients.

The code for this work also used a method where multiple previous solutions were used as initial guesses for a single lift coefficient and then the result with lowest drag was kept. This was done by using an up then down method by starting at a lift coefficient of 0.1 with an initial guess of all zeros. Once the solution for $C_L = 0.1$ was achieved, that solution was used as the initial guess for $C_L = 0.2$, then the solution for the $C_L = 0.2$ optimization was used to find the results for $C_L = 0.3$. This process was repeated all the way ***up*** to $C_L = 0.9$. The process was then conducted backwards, where the solution from $C_L = 0.9$ was used as the initial guess for $C_L = 0.8$. The solution obtained from that optimization was then compared with the previous $C_L = 0.8$ solution and the better solution (lowest C_D) was kept. The best solution for $C_L = 0.8$ was then used as the initial guess for $C_L = 0.7$ and again the new solution was compared with the previous solution and the better of the two was kept. This process was repeated all the way ***down*** to $C_L = 0.1$. This method was used because it was found that prior to its implementation, the drag polar obtained from looping through $C_L = 0.1 - 0.9$ seemed to represent two different solutions. Meaning that the C_D values associated with $C_L = 0.1 - 0.X$ would lie on one curve and then there was a distinctly different curve that the C_D values for $C_L = 0.X - 0.9$ would lie on. By going up then down, all of the drag coefficients obtained were on the same curve and lower drag values were obtained than without the up/down method. Often this shift appeared around $C_L = 0.6$, or other intermediate C_L values.

It should be noted that it is not the belief of the author that this up/down method is the only way to ensure better results from the optimization. This method stemmed from looking at the case of optimizing for two control sections, where the second optimization with an initial guess increased the chances of starting the optimization at a different location in the solution space and thus increased the likelihood of finding a more global minima. When the number of control sections was increased to more than two control sections, this method was kept as it still gave clean and consistent results even though only looking at two different initial guesses does not search a larger design space as thoroughly as it does a design space with only two control sections. While in this work it is not assumed that the up/down method is the only way to search the solution space, it was found in this work that some method of searching the solution space more thoroughly than with just one initial guess helped improve the quality of results obtained. The exception to this finding was the case of running a baseline for the aircraft with no control sections. In this case, the optimization was essentially only pitch trimming the aircraft and no deflections were set. Therefore, while using the previous solution (only a horizontal stabilizer mounting angle adjustment and angle of attack) could help the solution trim the aircraft more quickly, it was not necessary to arrive at a good solution. The optimization converged quickly and accurately on an angle of attack and horizontal stabilizer mounting angle that pitch trimmed the aircraft. In the case of finding the baseline drag polar, it was only necessary to loop through all lift coefficients once and run the optimization as described in the sections above. An example of code for obtaining the baseline drag polar for the NASA Ikhana can be found in Appendix B.1, and an example of looping through multiple lift coefficients and using the up/down method for the NASA Ikhana can be found in Appendix B.2.

CHAPTER 3
RESULTS

The methods in preceding sections were used to obtain results for the NASA Ikhana over a range of lift coefficients with varying numbers of control sections. The results will be presented and compared in this section. In order to compare the results of different configurations of the NASA Ikhana, it is useful to define a few parameters.

The Oswald Efficiency Factor e is often used to show deviation from the elliptic lift distribution, which makes e a useful parameter to measure when trying to minimize drag as the elliptic distribution has minimum induced drag. As such, the Oswald efficiency factor will be used to help compare the drag reduction seen by optimizing the NASA Ikhana with multiple camber schedules. The Oswald efficiency factor depends on the aspect ratio R_A and the quadratic part of the drag coefficient as a function of the lift coefficient C_{D_2} according to

$$e = \frac{1}{\pi * R_A * C_{D_2}} \tag{3.1}$$

For this work, the percent change in drag between any two solutions was defined as

$$\zeta_{x \to y} = \frac{C_{D_y} - C_{D_x}}{C_{D_x}} * 100 \tag{3.2}$$

where $-\zeta$ indicates a reduction in drag and $+\zeta$ indicates an increase in drag. The subscript $x \to y$ represents the change in drag when moving from condition x to condition y. For example, $\zeta_{0 \to 2} = -14\%$ represents a 14% drag reduction when moving from 0 control sections to 2 control sections.

Comparing the lift distribution of the NASA Ikhana to the elliptic lift distribution is also useful, as the elliptic lift distribution produces minimum induced drag. Prandtl introduced the elliptic lift distribution as

$$\frac{b\tilde{L}(z)}{L} = \frac{4}{\pi}\left\{\sin\left[\cos^{-1}\left(\frac{-2z}{b}\right)\right]\right\} \tag{3.3}$$

where b is the span, z is the spanwise location along the wing, $\tilde{L}(z)$ is the section lift as a function of spanwise location, and L is the total lift [23]. When comparing the lift distributions of the NASA Ikhana with the elliptic distribution, only a single semi-span will be represented due to the symmetry of the NASA Ikhana.

In order to compare results from MachUpX with the elliptic lift distribution, the lift distribution for the Ikhana needed to be generated with data from MachUpX. The distributions file from MachUpX contained information about the section lift coefficient at various spanwise locations, which was used to calculate the lift distribution. The section lift coefficient as a function of spanwise location z is defined as

$$\tilde{C_L}(z) = \frac{\tilde{L}(z)}{\frac{1}{2}\rho {V_\infty}^2 c} \tag{3.4}$$

where ρ is the freestream density, c is the chord length, V_∞ is the freestream velocity, and $\tilde{L}(z)$ is the section lift as a function of spanwise location z. Solving Eq. 3.4 for the section lift as a function of spanwise location z gives

$$\tilde{L}(z) = \tilde{C_L}(z)\,\frac{1}{2}\rho {V_\infty}^2 c \tag{3.5}$$

The total lift coefficient is defined as

$$C_L = \frac{L}{\frac{1}{2}\rho {V_\infty}^2 S_w} \tag{3.6}$$

where L is the total lift and S_w is the planform area of the wing. Solving Eq. 3.6 for the total lift gives

$$L = C_L \frac{1}{2}\rho {V_\infty}^2 S_w \tag{3.7}$$

Combining Eq. 3.5 and 3.7 to match the left hand side of Eq. 3.3 allows for calculation of the Ikhana lift distribution using

$$\frac{b\tilde{L}(z)}{L} = \frac{b\left[\tilde{C_L}(z) * c\right]}{C_L S_w} \tag{3.8}$$

where the section lift coefficient comes from MachUpX data and the total lift coefficient is the desired lift coefficient used for the optimization. Equations 3.3 and 3.8 both calculate the normalized section lift and are directly comparable. All lift distributions presented in this section were generated using Eq. 3.3, Eq. 3.8, and MachUpX data obtained from the optimizations.

3.1 NASA Ikhana

Optimization of the NASA Ikhana showed that as more control sections were added to the wing, the drag was reduced and there was a point of diminishing returns, beyond which adding control sections resulted in negligible drag reduction. For the NASA Ikhana, this point of diminishing returns proved to be very low. Figure 3.1 shows the drag polars for the NASA Ikhana with 0 control sections (the baseline case), 2 control sections, 4 control sections, and at the theoretical limit (TL) of an Oswald efficiency $e = 1$. The black dashed line in Fig. 3.1 represents the theoretical minimum drag polar that could be obtained if it were possible to achieve an Oswald efficiency of $e = 1$. It is important to note that achieving an Oswald efficiency of $e = 1$ is not physically possible, even for the elliptic distribution which has minimum induced drag. However, it can be useful to compare results against a theoretical efficiency of $e = 1$ to show how much room remains for improvement in reducing drag. Solving Eq. 3.1 for C_{D_2} when $e = 1$ for the NASA Ikhana yielded $C_{D_2} = 1/16\pi$, which was used in Eq. 2.2 with $C_{D_1} = 0.0$, and $C_{D0} = 6.563E-03$ to generate the TL drag polar. The value for C_{D_0} came from curve fitting a second-order polynomial to the drag polars for the NASA Ikhana with 0, 2, or 4 control sections and finding they all shared the same zero lift drag coefficient.

It can be seen from Fig. 3.1 that moving from 2 control sections to 4 control sections yielded negligible changes in drag reduction. Table 3.1 shows the values of C_D for the baseline Ikhana, the Ikhana with 2 control sections, and the Ikhana with 4 control sections as well as $\zeta_{0\to2}$, $\zeta_{2\to4}$, and $\zeta_{4\to\mathrm{TL}}$. Examining Fig. 3.1 and Table 3.1 shows that moving from the baseline to 2 control sections resulted in significant drag reductions, with $\approx 26\%$ drag reduction at $C_L = 0.9$. However, increasing the number of control sections from 2 to 4 yielded only a 0.7449% drag reduction, and even moving to the TL would only reduce drag another 1.2808% from the 4 control section solution.

At first, this small change in drag reduction when increasing the number of control sections was puzzling. The camber schedules shown in Fig. 3.2 and 3.3 looked reasonable. There were no indications of multiple solutions being present, which often represented itself with a split point in the camber schedules. A split point was represented by the camber schedules for lower lift coefficients following one pattern and the camber schedules for higher lift coefficients taking on a dramatically different form. Camber schedules with such behavior often indicated a solution that didn't represent the minimum drag. The camber is represented as a function of lift coefficient for each control section in Fig. 3.4 and 3.5, where smooth changes in camber are seen with increasing lift coefficient for both 2 and 4 control sections. If there were a split point in the camber schedules, then the values of camber as a function of lift coefficient represented in Fig. 3.4 and 3.5 would no longer be smooth, but would have jumps representing a change in camber schedule pattern.

Figures 3.6 and 3.7 show the airfoils resulting from optimization of the NASA Ikhana at $C_L = 0.9$ with 2 and 4 control sections compared with the NACA 0010 airfoil. Optimization of the NASA Ikhana with 2 control sections at $C_L = 0.9$ resulted in a camber of 0.1384% for control section 1 and a camber of -0.0521% for control section 2. It can be seen in Fig. 3.6 that for the 2 control section Ikhana at $C_L = 0.9$ the first control section has small positive camber compared to the NACA 0010, whereas the second control section has a small negative camber compared to the NACA 0010. For the 4 control section Ikhana optimized at $C_L = 0.9$, control section 1 has a camber of -0.1171%, control section 2 a camber of 0.3379%, control section 3 a camber of 0.5734%, and control section 4 has a negative camber of −0.9252% as shown in Fig. 3.7. Combining Fig. 3.6 and 3.7 with Fig. 3.2 - 3.5 aids in visualizing the control sections of the optimized NASA Ikhana.

Table 3.1: NASA Ikhana C_D associated with 0, 2, and 4 control sections and the associated percent reduction of C_D as the number of control sections increase.

NASA Ikhana Drag Reduction						
C_L	0 CS C_D	2 CS C_D	4 CS C_D	$\zeta_{0\rightarrow 2}$	$\zeta_{2\rightarrow 4}$	$\zeta_{4\rightarrow TL}$
0.1	7.1006E-03	6.7706E-03	6.7685E-03	-4.6481%	-0.0301%	-0.0971%
0.2	8.0127E-03	7.3871E-03	7.3784E-03	-7.8077%	-0.1173%	-0.2663%
0.3	9.5320E-03	8.4118E-03	8.3928E-03	-11.7515%	-0.2261%	-0.4685%
0.4	1.1659E-02	9.8461E-03	9.8118E-03	-15.5458%	-0.3487%	-0.6697%
0.5	1.4393E-02	1.1690E-02	1.1636E-02	-18.7809%	-0.4668%	-0.8513%
0.6	1.7735E-02	1.3937E-02	1.3861E-02	-21.4180%	-0.5410%	-0.9841%
0.7	2.1684E-02	1.6598E-02	1.6495E-02	-23.4535%	-0.6191%	-1.1167%
0.8	2.6240E-02	1.9668E-02	1.9530E-02	-25.0461%	-0.6992%	-1.2031%
0.9	3.1404E-02	2.3144E-02	2.2972E-02	-26.3024%	-0.7449%	-1.2808%

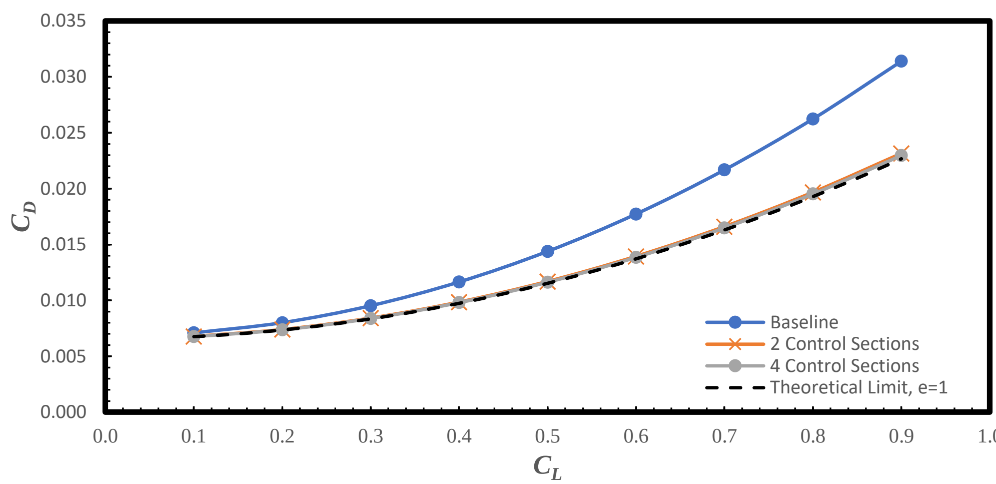

Fig. 3.1: Drag polar comparison for the NASA Ikhana with 0, 2, and 4 control sections.

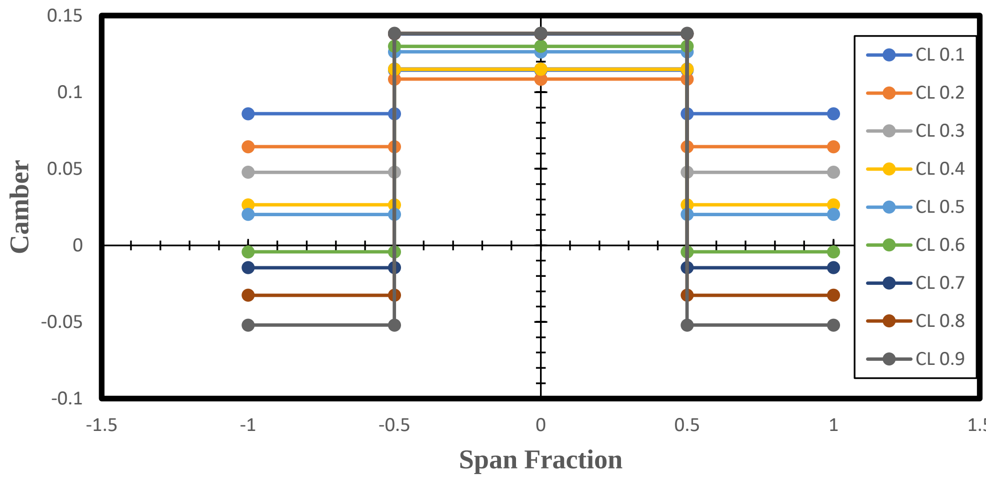

Fig. 3.2: Camber schedule for the NASA Ikhana with 2 control sections.

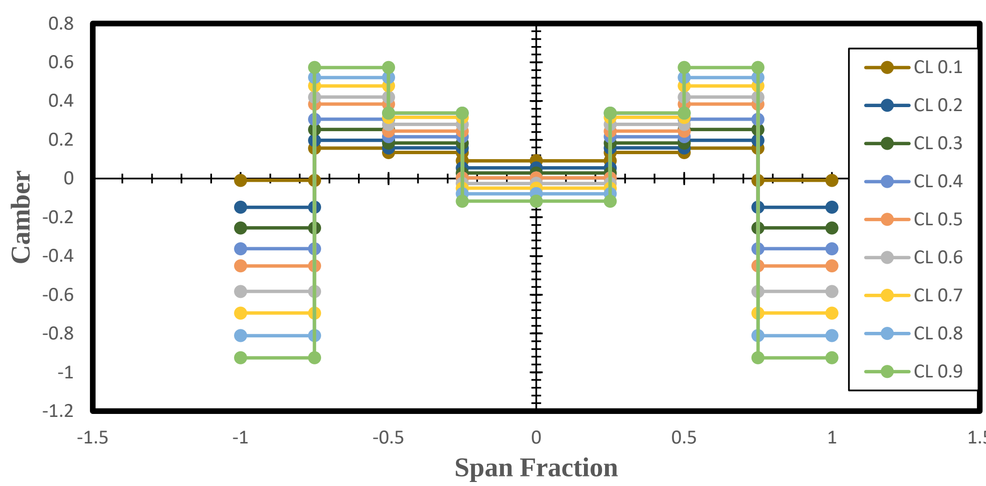

Fig. 3.3: Camber schedule for the NASA Ikhana with 4 control sections.

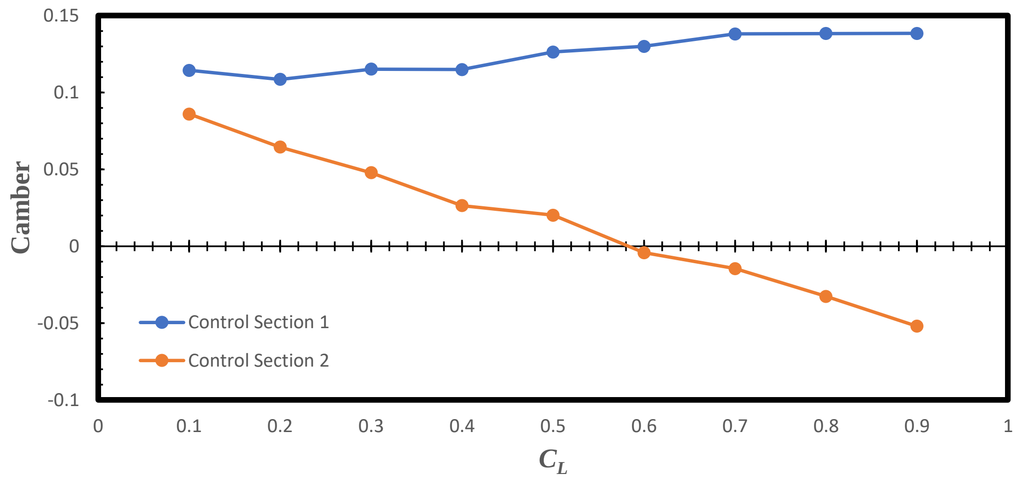

Fig. 3.4: Camber as a function of lift coefficient for the NASA Ikhana with 2 control sections.

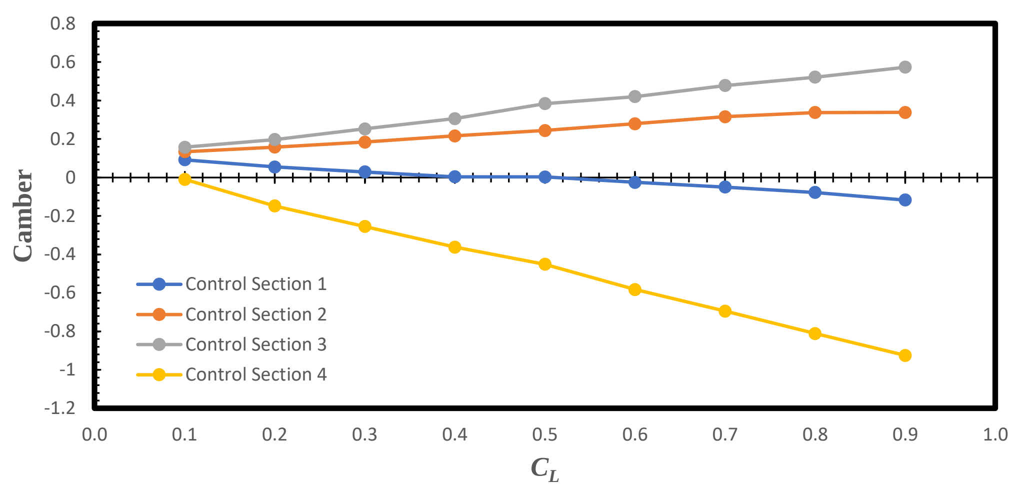

Fig. 3.5: Camber as a function of lift coefficient for the NASA Ikhana with 4 control sections.

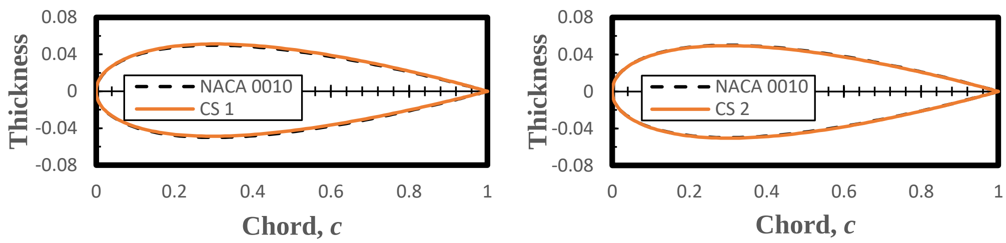

Fig. 3.6: Resultant airfoils from optimization of the NASA Ikhana at $C_L = 0.9$ with 2 control sections.

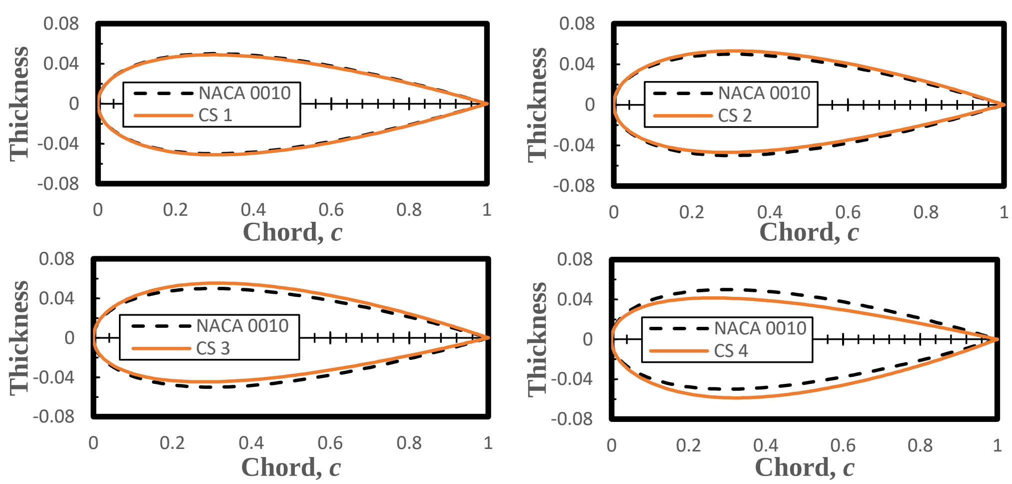

Fig. 3.7: Resultant airfoils from optimization of the NASA Ikhana at $C_L = 0.9$ with 4 control sections.

To help determine if the results obtained for the 4 control section NASA Ikhana were accurate, an attempt to force a solution in different parts of the solution space was made. In order to search more of the solution space, a lift coefficient of $C_L = 0.6$ was used with various initial guesses. A value of $C_L = 0.6$ was chosen because larger lift coefficients generally show more reduction in drag, yet $C_L = 0.6$ stays low enough that if a better solution was found, improvements would be more likely to be measurable both above and below the value used in the search. To force the search into different parts of the solution space, 16 possible high/low initial conditions for the camber of control sections were used as initial guesses, where high means positive camber and low means negative camber. Using 4 control sections there are 16 possible high/low combinations for the initial guess used in the optimization, which is why 16 combinations were used. All initial guesses had a slightly negative δ_h and positive α. This was done to avoid increasing the number of combinations from 16 to 64 and because a positive α and slightly negative δ_h were expected, given the characteristics and operating conditions of the NASA Ikhana. Table 3.2 shows the search of the solution space with the 16 initial guesses including the values of each initial guess and their associated percent change in drag when compared to the original $C_L = 0.6$ optimization solution, as shown in Fig. 3.1. For Table 3.2, positive camber values are shown in gray and negative camber values are shown in white to highlight the various patterns used as initial guesses. The camber of each control section in Table 3.2 is distinguished with $\delta_1, \delta_2, \delta_3,$ or δ_4 where δ_1 is the camber of the control section closest to the wing root and δ_4 is the camber of the control section closest to the wing tip.

The ζ value in the last column of Table 3.2 uses Eq. 3.2. Except, instead of comparing differing numbers of control sections, the original drag value from the optimization and the solution space search drag values were compared. As can be seen in Table 3.2, regardless of the initial guess, none of the runs resulted in lower drag than the original optimization at $C_L = 0.6$. Examination of the ζ values for the solution space search shows that half of the attempts to force a better solution resulted in increased drag, and half resulted in the same drag as was originally found from the optimization in Fig. 3.1, out to four decimal places.

Table 3.2: Solution space search for NASA Ikhana with 4 control sections at $C_L = 0.6$.

Initial x Array						$\zeta_{\text{original}\rightarrow\text{search}}$
δ_1	δ_2	δ_3	δ_4	δ_h	α	
2.0000	2.0000	2.0000	2.0000	-3.3513	5.3620	0.0005%
2.0000	2.0000	2.0000	-2.0000	-3.3513	5.3620	0.0000%
2.0000	1.0000	-3.0000	4.0000	-4.0000	3.5000	0.0000%
2.0000	2.0000	-2.0000	-2.0000	-3.3513	5.3620	0.0000%
2.0000	-2.0000	2.0000	2.0000	-3.3513	5.3620	0.0007%
2.0000	-2.0000	2.0000	-2.0000	-3.3513	5.3620	0.0000%
2.0000	-1.0000	-2.0000	2.0000	-4.0000	3.5000	0.0000%
2.0000	-1.0000	-2.0000	-2.0000	-4.0000	3.5000	0.1546%
-2.0000	2.0000	2.0000	2.0000	-3.3513	5.3620	0.0000%
-2.0000	2.0000	2.0000	-2.0000	-3.3513	5.3620	0.0069%
-2.0000	2.0000	-2.0000	2.0000	-3.3513	5.3620	0.0025%
-2.0000	2.0000	-2.0000	-2.0000	-3.3513	5.3620	0.0000%
-2.0000	-2.0000	2.0000	2.0000	-3.3513	5.3620	0.0104%
-2.0000	-2.0000	2.0000	-2.0000	-3.3513	5.3620	0.0008%
-2.0000	-1.0000	-2.0000	2.0000	-4.0000	3.5000	0.0000%
-2.0000	-2.0000	-2.0000	-2.0000	-3.3513	5.3620	0.0005%

The results of the solution space search combined with examination of the Oswald efficiency factor for the NASA Ikhana with 0, 2, and 4 control sections, calculated with Eq. 3.1 and shown in Fig. 3.8, led to the conclusion that the high aspect ratio tapered wing of the Ikhana is too near the elliptic distribution to significantly benefit from large numbers of control sections. The Oswald efficiency factor of the baseline Ikhana was found to be $e = 0.6551$. With 2 control sections, the efficiency improved to $e = 0.9763$, and moving to 4 control sections increased the efficiency slightly to $e = 0.9839$. For both the 2 and 4 control section results, the Ikhana approached the theoretical limit on Oswald efficiency, represented by the black dashed line in Fig. 3.8. Using Eq. 3.3, the elliptic lift distribution was compared with the lift distributions for the NASA Ikhana generated using Eq. 3.8. Figure 3.9 compares the elliptic lift distribution with the baseline, 2, and 4 control section NASA Ikhana lift distributions at a total lift coefficient of 0.8. As the number of control sections increased for the NASA Ikhana the lift distribution more closely approached the elliptic lift distribution. However, even as the number of control sections increase, no dramatic changes are seen in the lift distributions of Fig. 3.9, implying that the limit of diminishing returns for the NASA Ikhana occurs with as few as 2 to 4 control sections. For comparison, and to verify these conclusions, a version of the NASA Ikhana with a rectangular wing of the same aspect ratio was run through the optimization.

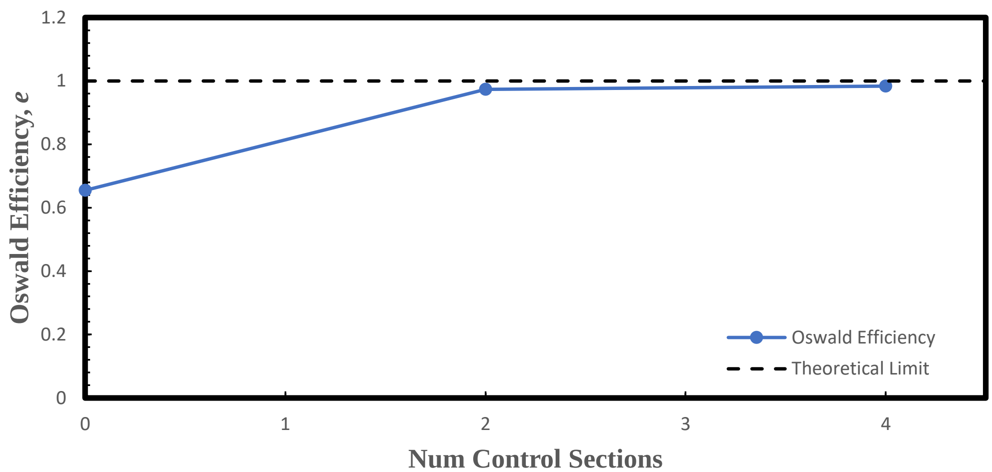

Fig. 3.8: Oswald efficiency of the NASA Ikhana with 0, 2, and 4 control sections.

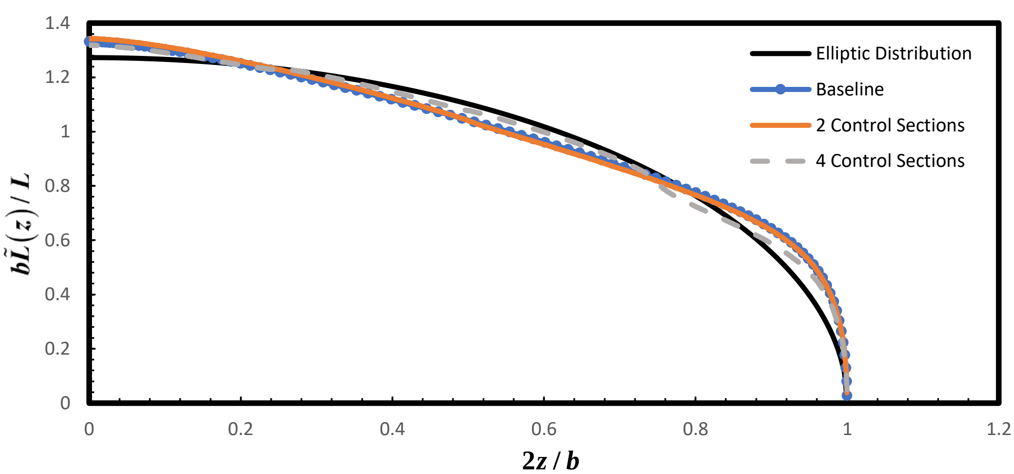

Fig. 3.9: Lift distributions for the NASA Ikhana with 0, 2, and 4 control sections vs the elliptic lift distribution.

3.2 NASA Ikhana with Rectangular Wing

The tapered wing of the NASA Ikhana was replaced with a rectangular wing by using the physical properties shown in Table 3.3. Comparing the physical properties of the rectangular Ikhana with those for the regular (tapered) Ikhana in Table 2.2, it is seen that only the chord changed. The same reference area, aspect ratio, weight, and other properties were maintained.

Table 3.3: Physical Properties of the Rectangular Ikhana.

Rectangular Ikhana Physical Properties	
Aspect Ratio, R_A	16
$CG_{x,y,z}$ (m)	0.0
Mean chord, $\overline{c}$ (m)	1.2192
Reference Area, S_w (m^2)	23.7832
Reference Longitudinal Length, l_{ref} (m)	1.2192
Span, b (m)	19.5072
Semi-span, $b/2$ (m)	9.7536
Chord, c (m)	1.2192
Weight, W (N)	31,593

Results for the rectangular Ikhana are summarized in Table 3.4, and the drag polar is shown in Fig. 3.10. The black dashed line in Fig. 3.10 represents the drag polar at the theoretical limit (TL) of an Oswald efficiency $e = 1$. Since the rectangular Ikhana and the tapered Ikhana have the same aspect ratio, solving Eq. 2.2 for C_{D_2} with $e = 1$ gave $C_{D_2} = 1/16\pi$. $C_{D_1} = 0.0$ was also used again. However, fitting a second-order polynomial to any of the drag polars for 0, 2, or 4 control sections on the rectangular Ikhana gave $C_{D_0} = 6.6562E - 03$. The drag polars in Fig. 3.10 show that results for the rectangular Ikhana with 2 and 4 control sections are still very similar, but there is more separation between the two solutions at larger lift coefficients than was seen in Fig. 3.1 for the tapered Ikhana. Also, as shown in Fig. 3.10 and Table 3.4, there is more room to reduce drag when moving from 4 control sections towards the theoretical limit. Table 3.4 shows the same trend of $\approx 27\%$ drag reduction when moving from the baseline to 2 control sections; however, there was $\approx 2\%$ drag reduction when going from 2 to 4 control sections at $C_L = 0.9$, more than double what was seen for the tapered Ikhana, and moving from 4 control sections towards the theoretical limit would give an $\approx 3.5\%$ further reduction in drag.

Figures 3.11 - 3.16 respectively show the camber schedules with 2 and 4 control sections, camber as a function of lift coefficient for 2 and 4 control sections, and the airfoils resulting from optimization of the rectangular Ikhana at $C_L = 0.9$ with 2 and 4 control sections. As was noted with the tapered Ikhana, the camber schedules and camber as a function of lift coefficient plots are consistent and do not have any jumps which would indicate multiple solutions. Figures 3.15 and 3.16 show the airfoils resulting from optimization of the rectangular Ikhana at $C_L = 0.9$. Figure 3.15 shows a larger positive camber for control section 1 and larger negative camber for control section 2, when compared to Fig. 3.6 for the tapered Ikhana. The rectangular Ikhana with 2 control sections had camber values of 1.0116% and -1.2857% for control sections 1 and 2 compared with 0.1384% and -0.0521% for the tapered Ikhana. Examination of Fig. 3.16 shows different behavior for the rectangular Ikhana with 4 control sections than was seen with the 4 control section tapered Ikhana in Fig. 3.7. For the rectangular Ikhana, control section 1 had a positive camber of 1.3204%, control section 2 a camber of 0.8367%, control section 3 a camber of -0.1322%, and control section 4 a camber of -2.3828%. Whereas with the tapered Ikhana, the camber increased from -0.1171% at control section 1 to 0.5734% at control section 3 and then control section 4 had a camber of -0.9252%.

Table 3.4: Rectangular Ikhana C_D associated with 0, 2, and 4 control sections along with the associated percent reduction of C_D as the number of control sections increase.

Rectangular NASA Ikhana Drag Reduction						
C_L	0 CS C_D	2 CS C_D	4 CS C_D	$\zeta_{0\to 2}$	$\zeta_{2\to 4}$	$\zeta_{4\to \mathrm{TL}}$
0.1	7.1218E-03	6.7815E-03	6.7756E-03	-4.7784%	-0.0861%	-0.2169%
0.2	8.0988E-03	7.4287E-03	7.4058E-03	-8.2735%	-0.3084%	-0.6490%
0.3	9.7268E-03	8.5051E-03	8.4536E-03	-12.5604%	-0.6055%	-1.1962%
0.4	1.2006E-02	1.0010E-02	9.9192E-03	-16.6205%	-0.9119%	-1.7551%
0.5	1.4936E-02	1.1945E-02	1.1802E-02	-20.0277%	-1.1911%	-2.2607%
0.6	1.8517E-02	1.4308E-02	1.4103E-02	-22.7331%	-1.4285%	-2.6890%
0.7	2.2749E-02	1.7099E-02	1.6821E-02	-24.8385%	-1.6232%	-3.0368%
0.8	2.7632E-02	2.0318E-02	1.9956E-02	-26.4709%	-1.7795%	-3.3160%
0.9	3.3166E-02	2.3965E-02	2.3508E-02	-27.7428%	-1.9047%	-3.5392%

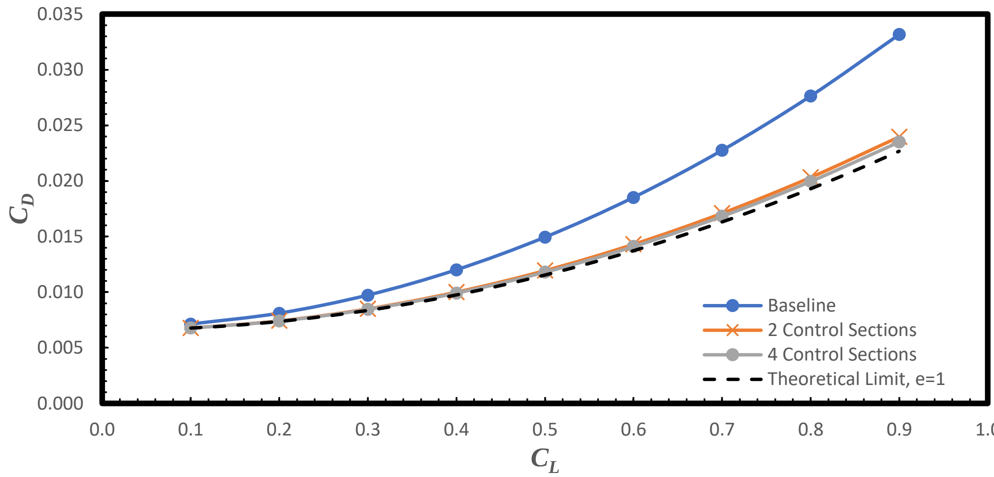

Fig. 3.10: Drag polar comparison for the rectangular Ikhana with 0, 2, and 4 control sections.

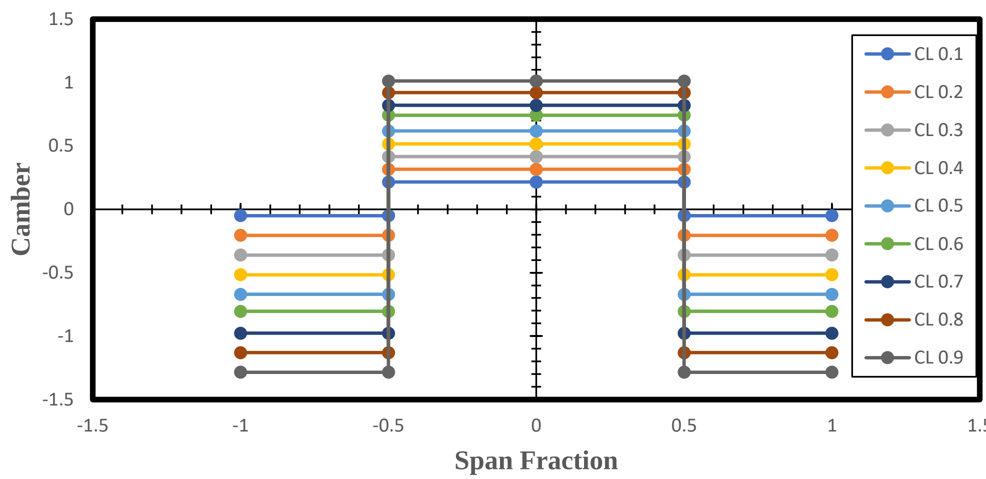

Fig. 3.11: Camber schedule for the rectangular Ikhana with 2 control sections.

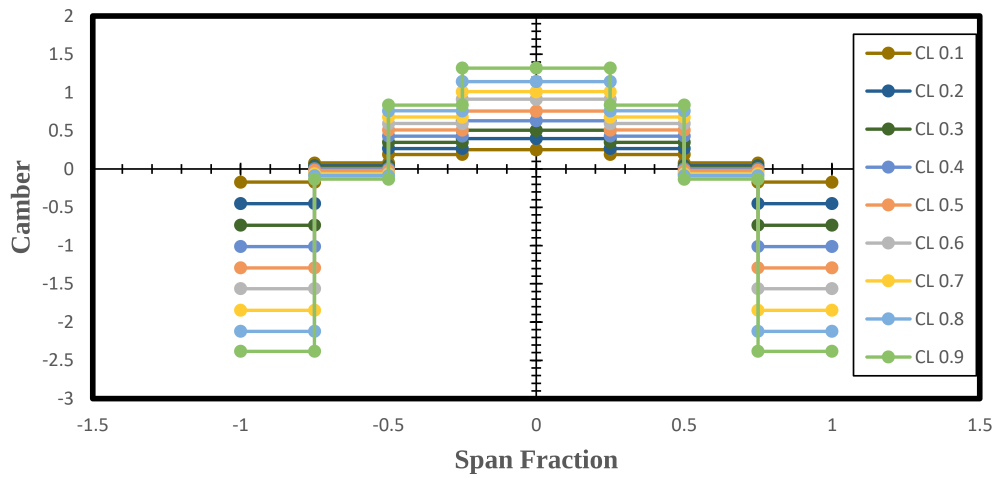

Fig. 3.12: Camber schedule for the rectangular Ikhana with 4 control sections.

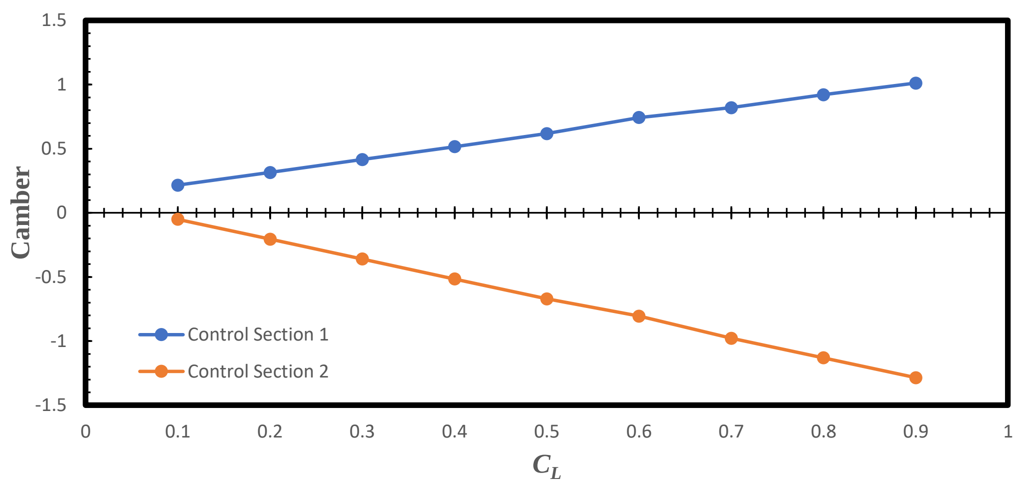

Fig. 3.13: Camber as a function of lift coefficient for the rectangular Ikhana with 2 control sections.

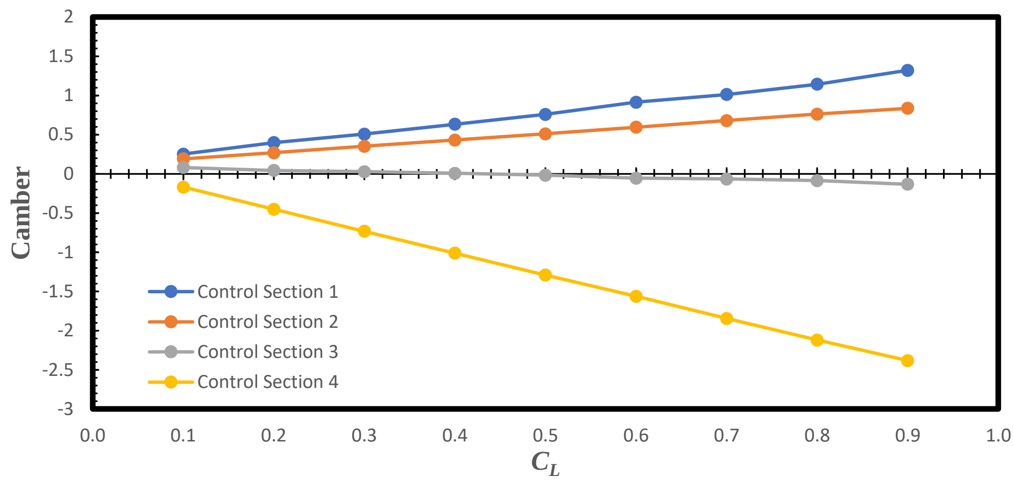

Fig. 3.14: Camber as a function of lift coefficient for the rectangular Ikhana with 4 control sections.

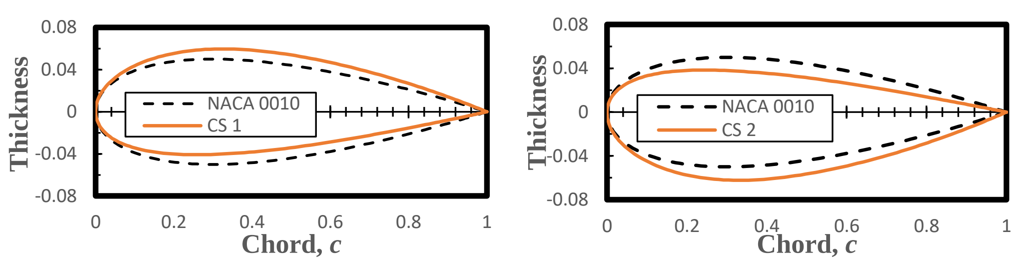

Fig. 3.15: Resultant airfoils from optimization of the rectangular Ikhana at $C_L = 0.9$ with 2 control sections.

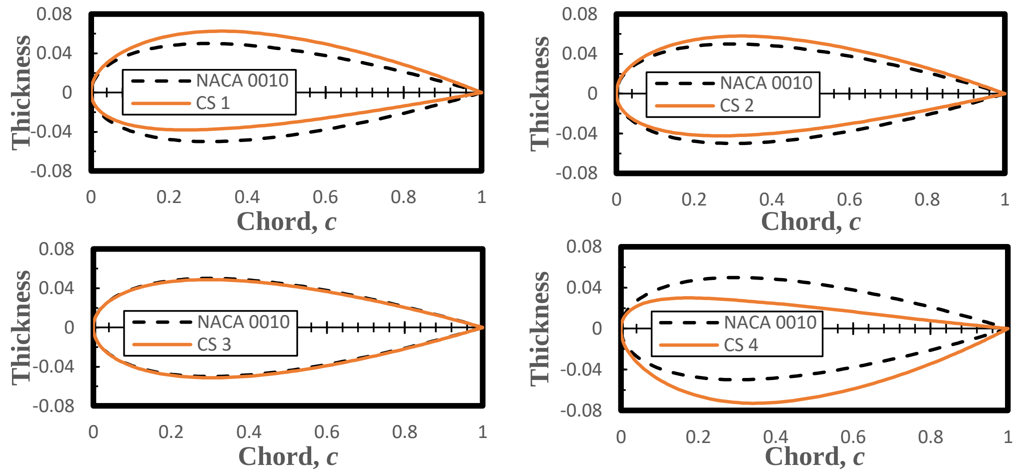

Fig. 3.16: Resultant airfoils from optimization of the rectangular Ikhana at $C_L = 0.9$ with 4 control sections.

Figure 3.17 shows that even with a rectangular wing, e increases with more control sections and approaches the theoretical limit of $e = 1$ for Oswald efficiency, as represented by the dashed black line. Figure 3.18 shows the lift distributions for the rectangular Ikhana compared with the elliptic lift distribution. As the number of control sections increased the lift distribution for the rectangular Ikhana more closely approached the elliptic distribution.

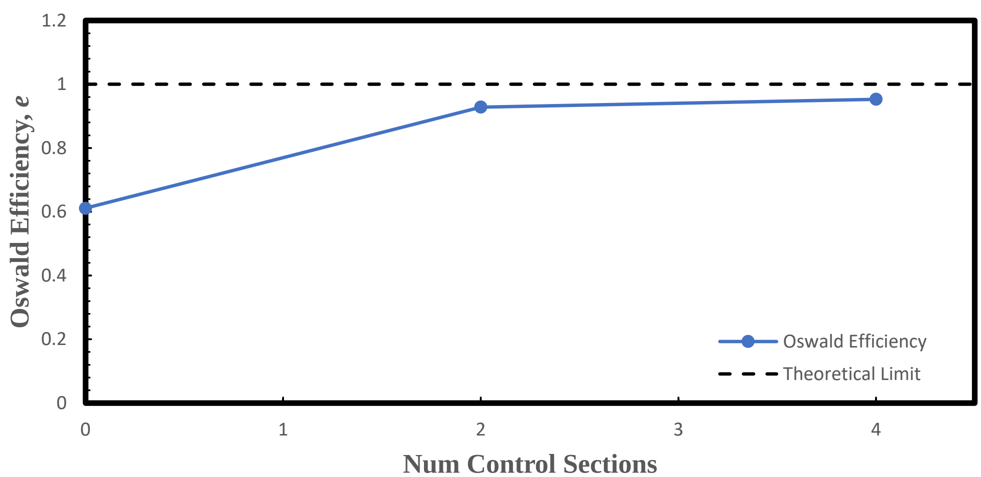

Fig. 3.17: Oswald efficiency of the rectangular Ikhana with 0, 2, and 4 control sections.

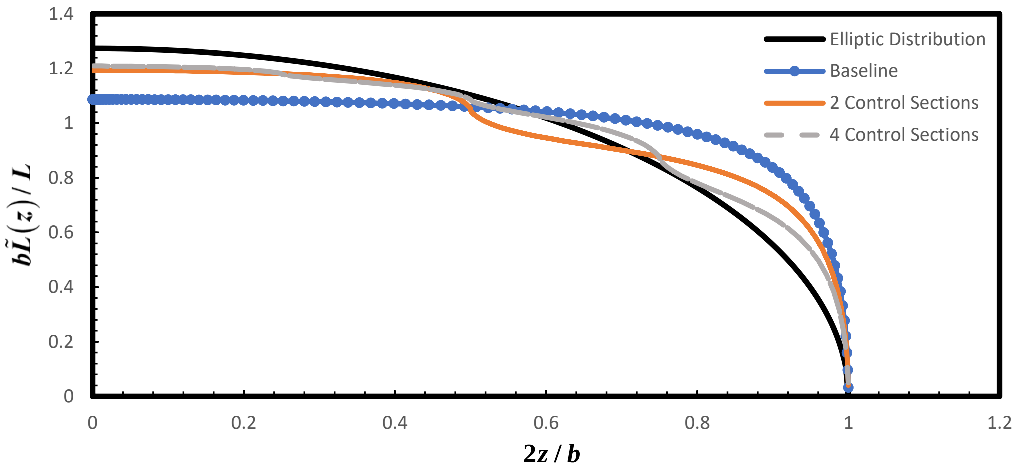

Fig. 3.18: Lift distributions for the rectangular Ikhana with 0, 2, and 4 control sections vs the elliptic lift distribution.

3.3 NASA Ikhana vs. Rectangular Ikhana Comparison

Results for the rectangular Ikhana are different enough from results for the tapered Ikhana to support the conclusion that the NASA Ikhana, with its high aspect ratio tapered wing, has a lift distribution that is very near the elliptic distribution and will not significantly benefit from large numbers of control sections. Table 3.5 compares the magnitude of $\zeta_{0\to 2}$ and $\zeta_{2\to 4}$ for the tapered Ikhana and the rectangular Ikhana. The ζ values are compared using the ratio

$$\chi_{x\to y} = \frac{\zeta_{x\to y,Rectangular}}{\zeta_{x\to y,Tapered}} \tag{3.9}$$

which compares the drag reduction seen for the rectangular Ikhana with the equivalent drag reduction seen with the tapered Ikhana. A value of $\chi = 1$ would indicate the drag reduction was exactly equivalent between the rectangular Ikhana and the tapered Ikhana. A χ greater than 1 would represent that the rectangular Ikhana saw a greater drag reduction, ζ, than the tapered Ikhana and a value less than 1 would represent the tapered Ikhana saw a greater drag reduction than the rectangular Ikhana. Data from Tables 3.4 and 3.1 were used to calculate the values for χ in Table 3.5.

Results in Table 3.5 show that both the tapered and rectangular Ikhana demonstrated roughly the same percent drag reduction when going from the baseline to 2 control sections. However, when going from 2 to 4 control sections or 4 control sections towards the theoretical limit where $e = 1$, the percent drag reduction for the rectangular Ikhana was more than double what it was for the tapered Ikhana. This indicates the rectangular Ikhana is not as optimized as the tapered Ikhana and suggests that the more a wing's lift distribution varies from the elliptic distribution, the more control sections can be added before diminishing returns are seen in drag reduction. This is supported by Fig. 3.19, which compares the Oswald efficiencies for the tapered Ikhana and the rectangular Ikhana. While e for both versions approach unity (the theoretical limit) with increased control sections, the rectangular Ikhana has a lower e with 0, 2, and 4 control sections; thus, leaving more room to reduce drag with additional control sections before reaching the efficiency of the tapered Ikhana.

Table 3.5: Comparison of the percent drag reduction, between the regular (tapered) NASA Ikhana and the rectangular Ikhana.

NASA Ikhana χ ratio			
C_L	$\chi_{0\to2}$	$\chi_{2\to4}$	$\chi_{4\to\mathrm{TL}}$
0.1	1.0280	2.8602	2.2348
0.2	1.0596	2.6294	2.4370
0.3	1.0688	2.6782	2.5534
0.4	1.0691	2.6148	2.6206
0.5	1.0664	2.5514	2.6554
0.6	1.0614	2.6407	2.7323
0.7	1.0591	2.6221	2.7195
0.8	1.0569	2.5449	2.7561
0.9	1.0548	2.5570	2.7633

Comparison of the lift distributions for the tapered Ikhana with the rectangular Ikhana also supported these results. Figure 3.20 compares the baseline tapered Ikhana and rectangular Ikhana lift distributions. It can be seen in Fig. 3.20 that the baseline tapered Ikhana has a lift distribution much closer to the elliptic distribution than the baseline rectangular Ikhana's lift distribution. Figures 3.21 and 3.22 compare the lift distributions of the tapered Ikhana and the rectangular Ikhana with 2 and 4 control sections respectively. In both Fig. 3.21 and Fig. 3.22 it can be seen that the lift distribution for the tapered Ikhana experienced little change with increased numbers of control sections. However, the rectangular Ikhana experienced far greater changes and additional control sections refined the lift distribution to more closely approximate the elliptic lift distribution.

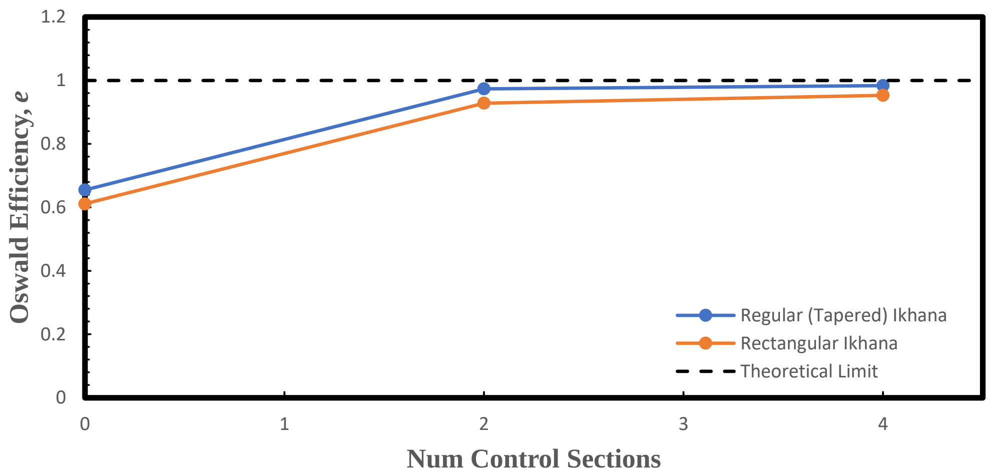

Fig. 3.19: Oswald efficiency of NASA Ikhana vs rectangular Ikhana with 0, 2, and 4 control sections.

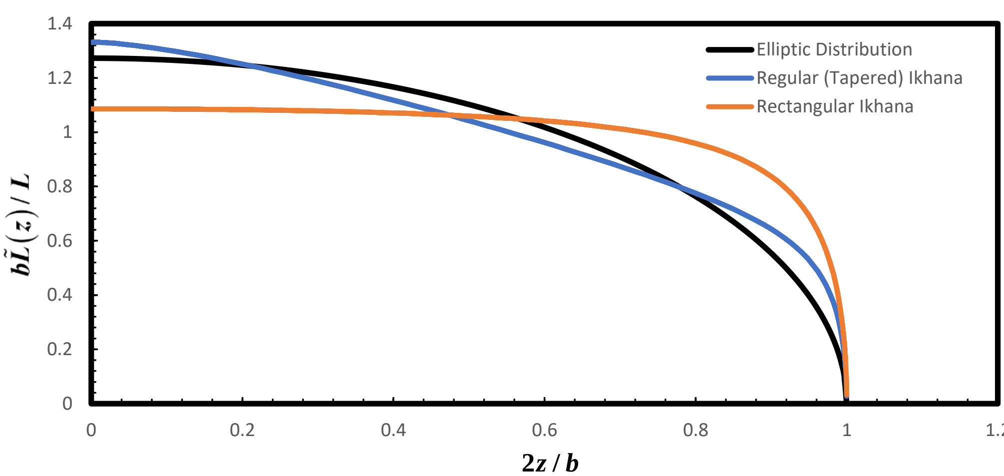

Fig. 3.20: Lift distribution comparison of the baseline NASA Ikhana and rectangular Ikhana.

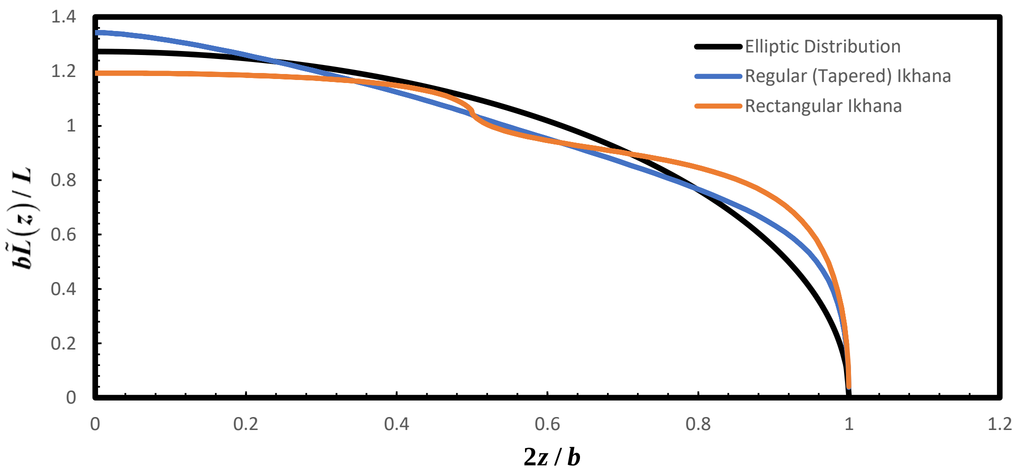

Fig. 3.21: Lift distribution comparison of the 2 control section NASA Ikhana and rectangular Ikhana.

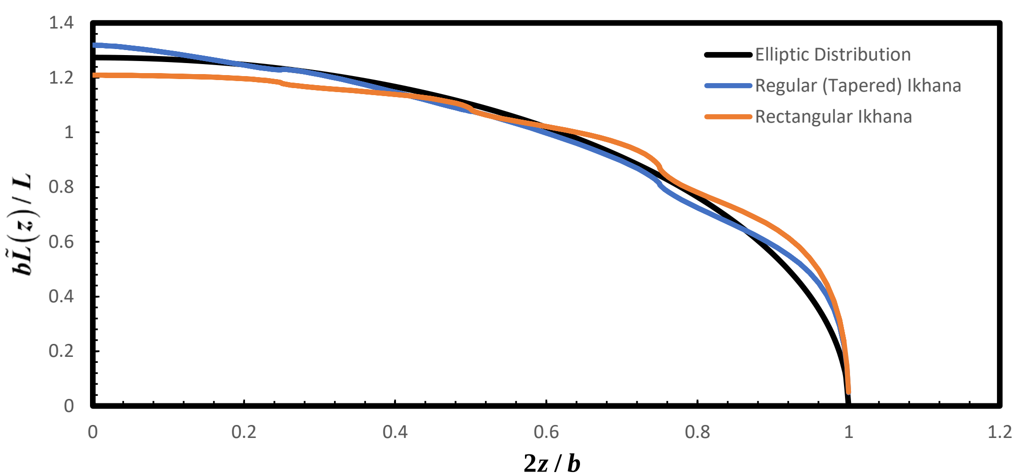

Fig. 3.22: Lift distribution comparison of the 4 control section NASA Ikhana and rectangular Ikhana.

3.4 Common Research Model

To present an example of a more complex wing geometry, the NASA Common Research Model (CRM) was optimized using the same method as the Ikhana. The airfoil database used for the CRM was obtained from Taylor [24]. For more information about the CRM airfoils see [25]. The CRM has a double tapered wing with the double taper taking place at the 30% span fraction point. The CRM was optimized with 2 control sections inboard of the double taper and 4 control sections outboard of the double taper with the resulting drag polar shown in Fig. 3.23. Camber scheduling for the optimized CRM with two inboard and four outboard control sections is shown in Fig. 3.24 for multiple lift coefficients. Camber as a function of lift coefficient for each control section of the optimized CRM is shown in Fig. 3.25, where control sections 1 and 2 are inboard of the double taper, and control sections 3-6 are outboard of the double taper. Results for the CRM are only presented for the baseline case (no control sections) and the case with 6 total control sections (2 inboard and 4 outboard). This is due to the more complicated nature of the CRM and its operating conditions. The CRM operates in the transonic regime, which means that it is possible to get regions of supersonic flow over the upper surface of the wing, which MachUpX is not equipped to handle. Also, the fits for the airfoil data are far more complex for the CRM than for the Ikhana, and obtaining well-behaved results for even this one case was time consuming and challenging to get consistent results over a wide range of lift coefficients.

Using the aerodynamic tools discussed in the above sections and Eq. 3.1, the Oswald efficiency of the baseline CRM is estimated to be $e = 0.2405$, while the Oswald efficiency of the optimized CRM is estimated to be $e = 0.6847$, a significant improvement. The baseline Oswald efficiency is extremely low for an airliner, and the improvement is dramatic, which raises questions about the accuracy of this model for the CRM. The poor accuracy is most likely due to the transonic operating conditions of the CRM, for which MachUpX is not well suited. It is also important to note that MachUpX neglects wave drag, which would significantly increase with regions of supersonic flow over the wing creating a mach bubble. The increased wave drag could even be significant enough to offset any benefits seen from optimal camber scheduling. While the numerical values of the CRM optimization need to be used with extreme caution, this example does highlight the ability of the method presented in this work to optimize more complex wing geometries. Should this method be used with a more complex geometry in subsonic operating conditions, it is believed that more reliable results would be obtained.

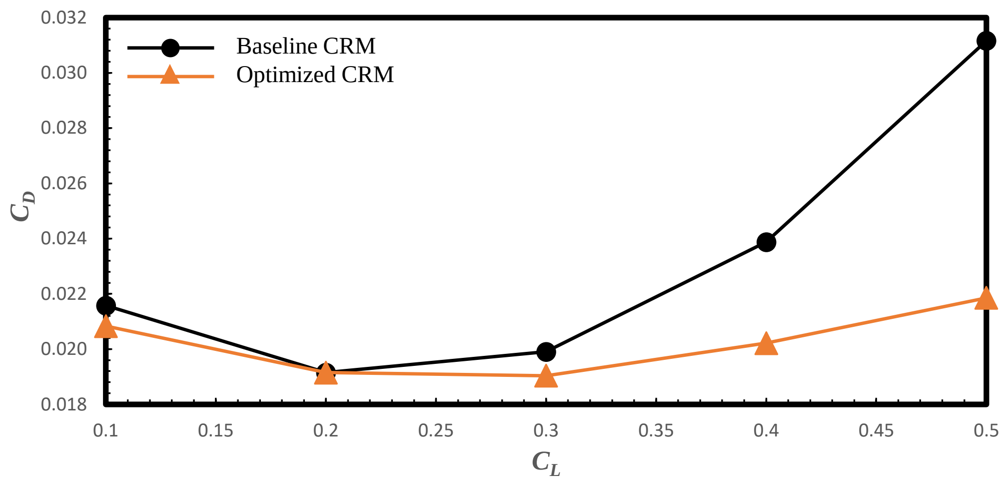

Fig. 3.23: Baseline CRM vs Optimized CRM Drag Polar.

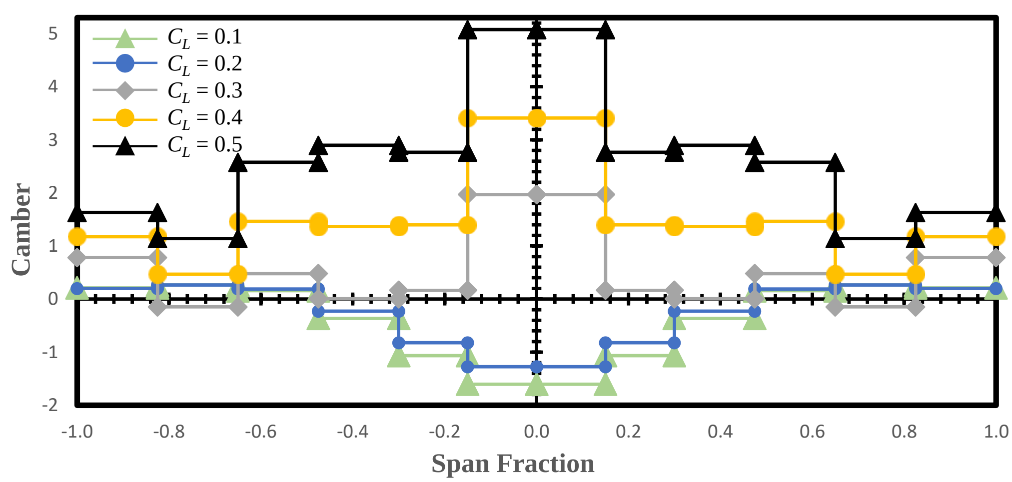

Fig. 3.24: Camber schedule for optimized CRM with 2 inboard & 4 outboard control sections at multiple C_L values.

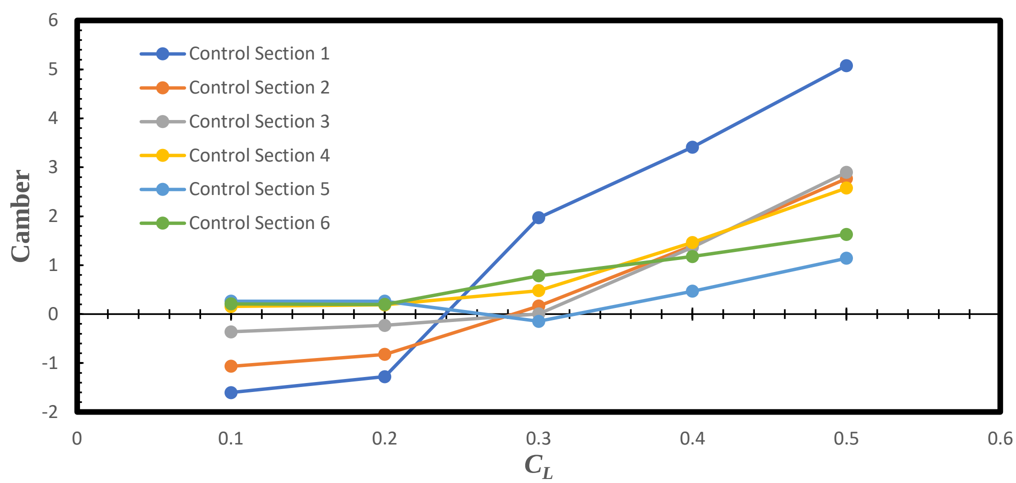

Fig. 3.25: Camber as a function of lift coefficient for the CRM with 2 inboard & 4 outboard control sections.

CHAPTER 4
SUMMARY AND CONCLUSIONS

In this work the effect of morphing (the ability to change camber during flight) on minimizing the drag of the NASA Ikhana was examined. Minimizing drag is desirable due to drag's large contribution to increased fuel burn and detrimental effect on efficiency. In order to minimize drag, a low fidelity numerical lifting-line tool, MachUpX, was combined with an SLSQP method to create code to optimize the NASA Ikhana at multiple design conditions and produce the resultant drag polars and camber schedules. This method was introduced and an example of running an optimization case given. The objectives of this work included the following:

- Understanding how performance can be improved on the NASA Ikhana air frame by using camber scheduling.
- Understanding how optimal camber scheduling changes with flight condition.
- Understanding how increasing the number of control sections affects the performance of the NASA Ikhana aircraft.

This work showed that it is possible to improve the performance of the NASA Ikhana with camber scheduling. With camber scheduling for 2 control sections, the NASA Ikhana achieved an $\approx 26\%$ reduction in drag when compared to its baseline non-optimized configuration. Changes in flight condition were represented by increasing the desired lift coefficient, and it was shown that as the lift coefficient increased, larger values of camber were needed in the camber schedule to minimize drag. It was hypothesized that there is a theoretical limit to the number of control sections that can be added to an aircraft before diminishing returns are seen in drag reduction. For the NASA Ikhana, diminishing returns were seen after the addition of only 2 control sections. After 2 control sections, any additional control sections showed only marginal gains in drag reduction. This behavior was supported with comparison of the lift distributions for 0, 2, and 4 control sections with the elliptic lift distribution. The baseline lift distribution (0 control sections) for the NASA Ikhana with no optimization compared well with the elliptic distribution. For the NASA Ikhana, the benefit achieved with increased numbers of control sections was smoothing of the lift distribution. As the number of control sections were increased, the lift distribution was refined to more closely resemble the shape of the elliptic lift distribution. This behavior led to the conclusion that for the tapered high aspect ratio wing of the NASA Ikhana, there is not much room to optimize by adding large numbers of control sections before diminishing returns are seen.

For comparison, the NASA Ikhana was modeled with a rectangular wing of the same aspect ratio as the regular (tapered) Ikhana. Examination of the NASA Ikhana with a rectangular wing showed a larger change in the drag reduction with increased numbers of control sections when compared to the actual tapered geometry of the Ikhana. Similar to the tapered Ikhana, the rectangular Ikhana demonstrated an $\approx 27\%$ reduction in drag with the addition of 2 control sections. However, when the number of control sections was increased from 2 to 4, the rectangular Ikhana experienced a drag reduction between 2.5 and 2.8 times that of the tapered Ikhana. Comparison of the lift distributions for the rectangular Ikhana with 0, 2, and 4 control sections with the elliptic distribution supported this behavior. The baseline lift distribution for the rectangular Ikhana left far more room to optimize with the addition of control sections than did the baseline lift distribution of the tapered Ikhana. With the addition of both 2 and 4 control sections, the lift distribution of the rectangular Ikhana got closer and closer to the elliptic lift distribution.

The NASA Common Research Model (CRM) was also examined using the method presented in this work. The CRM is a transonic passenger jet used for research purposes and has a much more complex wing geometry than either the tapered or rectangular Ikhana. Drag reduction was seen when moving from the baseline to the optimized CRM; however, results for the CRM were found to be unreliable and challenging to work with due to the transonic operating conditions of the CRM, which MachUpX is not suited to handle.

Results from the NASA Ikhana, rectangular Ikhana, and CRM demonstrate that the method presented in this work is best suited to wings in subsonic operating conditions. Wings that have baseline lift distributions not already close to the elliptic lift distribution will yield better results with larger numbers of control sections. However, as the number of control sections is increased, it becomes increasingly difficult to ensure that the solution is a global minima. Further work could include examination of alternate subsonic wing geometries or validation with high fidelity tools.

While results for the NASA Ikhana do not show dramatic changes to performance, the optimization technique presented in this work could be used in flight algorithms to schedule camber during flight in order to minimize drag and fuel burn, or to inform the design of future aircraft. One area that could particularly benefit from this work is the growing small unmanned aerial vehicle (UAV) market. Small UAV's are good candidates for the methods presented in this work as they operate in subsonic conditions and many small UAV's have less efficient airframes than the NASA Ikhana. These less efficient airframes have lift distributions not near the elliptic distribution and could therefore see significant benefit from camber schedule optimization to minimize induced drag. The optimal camber schedule could then be paired with a morphing wing design, and significant performance improvements could be gained with minimal design changes.

REFERENCES

[1] Hunsaker, D. F., Phillips, W. F., and Joo, J. J., "Aerodynamic Shape Optimization of Morphing Wings at Multiple Flight Conditions," *AIAA SciTech 2017 Forum*, 2017.

[2] Merlin, P. W., "Ikhana: Unmanned Aircraft System Western States Fire Missions. Monographs in Aerospace History, Number 44," Tech. rep., 2009.

[3] Prandtl, L., "Tragflügeltheorie," *Nachricten von der Gesellschaft der Wissenschaften zu Göttingen*, Geschäeftliche Mitteilungen, Klasse, 1918, pp. 451–477.

[4] Phillips, W. F., "Incompressible Flow over Finite Wings," *Mechanics of Flight*, chap. 1, John Wiley Sons, Inc., 2nd ed., 2010, pp. 46–94.

[5] Glauert, H., *The calculation of the characteristics of tapered wings*, HM Stationery Office, 1922.

[6] Phillips, W., "Lifting-line analysis for twisted wings and washout-optimized wings," *Journal of aircraft*, Vol. 41, No. 1, 2004, pp. 128–136.

[7] Phillips, F. W., Fugal, S. R., and Spall, R. E., "Minimizing Induced Drag with Wing Twist, Computational-Fluid-Dynamics Validation," *Journal of Aircraft*, Vol. 43, No. 2, 2006, pp. 437–444.

[8] Nguyen, N., "Elastically Shaped Future Air Vehicle Concept," *NASA Innovation Fund Award*, October 2010.

[9] Joo, J. J., Marks, C. R., Zientarski, L., and Culler, A. J., "Variable Camber Compliant Wing - Design," *23rd AIAA/AHS Adaptive Structures Conference*, January 2015.

[10] Hetrick, J., Osborn, R., Kota, S., Flick, P., and Paul, D., "Flight Testing of Mission Adaptive Compliant Wing," *48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Honolulu, HI, April 2007.

[11] Moulton, B. and Hunsaker, D. F., "3D-Printed Wings with Morphing Trailing-Edge Technology," *AIAA Scitech Forum Virtual Event*, American Institute of Aeronautics and Astronautics, January 2021.

[12] Woods, B. K. S. and Friswell, M. I., "Preliminary investigation of a fishbone active camber concept," *Smart Materials, Adaptive Structures and Intelligent Systems*, Vol. 45103, American Society of Mechanical Engineers, 2012, pp. 555–563.

[13] Phillips, W. F., "New twist on an old wing theory," *Aerospace America*, Vol. 43, No. 1, 2005, pp. 27–30.

[14] Vos, R., Gürdal, Z., and Abdalla, M., "Mechanism for warp-controlled twist of a morphing wing," *Journal of Aircraft*, Vol. 47, No. 2, 2010, pp. 450–457.

[15] Dale, A., Cooper, J. E., and Mosquera, A., "Adaptive Camber-Morphing Wing using 0-ν Honeycomb," *54th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2013, p. 1510.

[16] Goates, C. D. and Hunsaker, D. F., "Modern Implementation and Evaluation of Lifting-Line Theory for Complex Geometries," *Journal of Aircraft*, 2022, pp. 1–19.

[17] Phillips, W. F. and Snyder, D., "Modern Adaptation of Prandtl's Classic Lifting-Line Theory," *Journal of Aircraft*, Vol. 37, No. 4, 2000, pp. 662–670.

[18] Goates, C. and Hunsaker, D., "MachUpX Documentation," `https://machupx.readthedocs.io/en/latest/index.html`, 2020.

[19] Reid, J. T. and Hunsaker, D. F., "A General Approach to Lifting-Line Theory, Applied to Wings with Sweep," *AIAA Scitech 2020 Forum*, American Institute of Aeronautics and Astronautics, Jan 2020.

[20] Rivers, M., "CRM Family of Models," `https://commonresearchmodel.larc.nasa.gov/`, 2012.

[21] Community, T. S., "SciPy Optimize Minimize Documentation," `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html`, 2008-2020.

[22] Goates, C. and Hunsaker, D., "Airfoil Database Documentation," `https://airfoildatabase.readthedocs.io/en/latest/airfoil_class.html`, 2019.

[23] Prandtl, L., "Applications of Modern Hydrodynamics to Aeronautics," *NACA Technical Report 116*, Vol. 121, 1921.

[24] Taylor, J. D. and Hunsaker, D. F., "Characterization of the Common Research Model Wing for Low-Fidelity Aerostructural Analysis," *AIAA Scitech 2021 Forum*, 2021, p. 1591.

[25] Schoenfeld, J., "NASA CRM Github repository," `https://github.com/Justice-Schoenfeld/CRM_Airfoil_database`, 2022.

APPENDICES

APPENDIX A

IKHANA JSON INPUT FILES

A.1 Ikhana Aircraft JSON

```
{
    "CG" : [0.0, 0.0, 0.0],
    "weight" : 31593.16386,
    "reference" : {
        "longitudinal_length": 1.2192,
        "area" : 23.7832
    },
    "controls" : {
        "flaps1" : {"is_symmetric" : true},
        "elevator" : {"is_symmetric" : true}
    },
    "airfoils" : {
            "Ikhana_NACA_0010_main": {
                    "type": "linear",
            "aL0": 0.0,
            "CLa": 6.43365,
            "CmLo": 0.0,
            "Cma": 0.0,
            "CD0": 0.00513,
            "CD1": 0.0,
            "CD2": 0.00984,
                    "geometry" : {
                            "outline_points" : "AirfoilDatabase/airfoils/uCRM-9_wr0_xfoil.txt"
                            }
            },
            "Ikhana_NACA_0010" : {
            "type": "linear",
            "aL0": 0.0,
            "CLa": 6.43365,
            "CmLo": 0.0,
            "Cma": 0.0,
            "CD0": 0.00513,
            "CD1": 0.0,
            "CD2": 0.00984
            }
    },
    "wings" : {
        "main_wing" : {
            "ID" : 1,
            "side" : "both",
            "is_main" : true,
            "semispan" : 9.7536,
            "sweep" : 0.0,
```

```
            "dihedral" : 0.0,
            "chord" : [[0.0, 1.70688],
                        [1.0, 0.73152]],
                "airfoil" : "Ikhana_NACA_0010_main",
                "control_surface" : {
                "chord_fraction" : 0.6,
                "control_mixing" : {"flaps1" : 1.0}
                },
            "grid" : {
                "N" : 100,
                "flap_edge_cluster" : true,
                "reid_corrections" : true
            }
        },
        "horizontal_tail" : {
            "ID" : 2,
            "side" : "both",
            "is_main" : false,
            "connect_to" : {
                "dx" : -3.068,
                "dz" : 0.0
            },
            "semispan" : 3.68808,
            "sweep" : 0.0,
            "dihedral" : 29.0,
            "chord" : [[0.0, 1.335024],
                        [1.0, 0.758952]],
            "twist" : [[0.0, 0.0],
                        [1.0, 0.0]],
            "airfoil" : "Ikhana_NACA_0010",
            "control_surface" : {
                "chord_fraction" : 0.27,
                "control_mixing" : {"elevator" : 1.0}
            },
            "grid" : {
                "N" : 50,
                "flap_edge_cluster" : true,
                "reid_corrections" : true,
                "blending_distance" : 0.25
            }
        }
    }
}
```

A.2 Ikhana Scene JSON


```json
{
    "tag" : "Ikhana",
    "solver" : {
        "type" : "nonlinear",
        "convergence" : 1e-6,
        "relaxation" : 0.9,
        "max_iterations" : 1000
    },
    "units" : "SI",
    "scene" : {
        "atmosphere" : {
            "altitude_m" : 6096,
            "rho" : "standard"
        },
        "aircraft" : {
            "Ikhana" : {
                "file" : "Ikhana.json",
                "state" : {
                    "position": [0.0, 0.0, -6069.0],
                    "velocity" : 102.889,
                    "alpha" : 0.0,
                    "beta" : 0.0
                }
            }
        }
    }
}
```

A.3 Rectangular Ikhana Aircraft JSON

```
{
    "CG" : [0.0, 0.0, 0.0],
    "weight" : 31593.16386,
    "reference" : {
        "longitudinal_length": 1.2192,
        "area" : 23.7832
    },
    "controls" : {
        "flaps1" : {"is_symmetric" : true},
        "elevator" : {"is_symmetric" : true}
    },
    "airfoils" : {
            "Ikhana_NACA_0010_main": {
                    "type": "linear",
            "aL0": 0.0,
            "CLa": 6.43365,
            "CmLo": 0.0,
            "Cma": 0.0,
            "CD0": 0.00513,
            "CD1": 0.0,
            "CD2": 0.00984,
                    "geometry" : {
                            "outline_points" : "AirfoilDatabase/airfoils/uCRM-9_wr0_xfoil.txt"
                            }
            },
            "Ikhana_NACA_0010" : {
            "type": "linear",
            "aL0": 0.0,
            "CLa": 6.43365,
            "CmLo": 0.0,
            "Cma": 0.0,
            "CD0": 0.00513,
            "CD1": 0.0,
            "CD2": 0.00984
            }
    },
    "wings" : {
        "main_wing" : {
            "ID" : 1,
            "side" : "both",
            "is_main" : true,
            "semispan" : 9.7536,
            "sweep" : 0.0,
            "dihedral" : 0.0,
            "chord" : [[0.0, 1.2192],
                        [1.0, 1.2192]],
                "airfoil" : "Ikhana_NACA_0010_main",
                "control_surface" : {
                "chord_fraction" : 0.6,
                "control_mixing" : {"flaps1" : 1.0}
                },
            "grid" : {
                "N" : 100,
```

```
                "flap_edge_cluster" : true,
                "reid_corrections" : true
            }
        },
        "horizontal_tail" : {
            "ID" : 2,
            "side" : "both",
            "is_main" : false,
            "connect_to" : {
                "dx" : -3.068,
                "dz" : 0.0
            },
            "semispan" : 3.68808,
            "sweep" : 0.0,
            "dihedral" : 29.0,
            "chord" : [[0.0, 1.335024],
                       [1.0, 0.758952]],
            "twist" : [[0.0, 0.0],
                       [1.0, 0.0]],
            "airfoil" : "Ikhana_NACA_0010",
            "control_surface" : {
                "chord_fraction" : 0.27,
                "control_mixing" : {"elevator" : 1.0}
            },
            "grid" : {
                "N" : 50,
                "flap_edge_cluster" : true,
                "reid_corrections" : true,
                "blending_distance" : 0.25
            }
        }
    }
}
```

A.4 Rectangular Ikhana Scene JSON

```
{
    "tag" : "Ikhana",
    "solver" : {
        "type" : "nonlinear",
        "convergence" : 1e-6,
        "relaxation" : 0.9,
        "max_iterations" : 1000
    },
    "units" : "SI",
    "scene" : {
        "atmosphere" : {
            "altitude_m" : 6096,
            "rho" : "standard"
        },
        "aircraft" : {
            "Ikhana" : {
                "file" : "Ikhana_rectangular.json",
                "state" : {
                        "position": [0.0, 0.0, -6069.0],
                    "velocity" : 102.889,
                    "alpha" : 0.0,
                    "beta" : 0.0
                }
            }
        }
    }
}
```

A.5 Supporting Text File: uCRM_9_wr0_xfoil.txt

```
 1.000000       0.000000
0.9983652      0.4292748E-03
0.9963459      0.9374885E-03
0.9941594      0.1460555E-02
0.9917761      0.1998688E-02
0.9891601      0.2551332E-02
0.9862693      0.3120117E-02
0.9830548      0.3712954E-02
0.9794583      0.4343483E-02
0.9754101      0.5027672E-02
0.9708252      0.5772999E-02
0.9656038      0.6585673E-02
0.9596369      0.7475185E-02
0.9528266      0.8450682E-02
0.9451096      0.9513505E-02
0.9364803      0.1065765E-01
0.9270097      0.1186688E-01
0.9168229      0.1311923E-01
0.9060671      0.1439988E-01
0.8949012      0.1570161E-01
0.8834768      0.1700788E-01
0.8718885      0.1830496E-01
0.8602071      0.1958630E-01
0.8484756      0.2084176E-01
0.8367045      0.2206736E-01
0.8249003      0.2326601E-01
0.8130720      0.2443907E-01
0.8012260      0.2558717E-01
0.7893667      0.2671042E-01
0.7774990      0.2780967E-01
0.7656258      0.2888402E-01
0.7537491      0.2993353E-01
0.7418698      0.3095785E-01
0.7299887      0.3195685E-01
0.7181065      0.3293053E-01
0.7062235      0.3387830E-01
0.6943389      0.3479996E-01
0.6824516      0.3569560E-01
0.6705607      0.3656540E-01
0.6586652      0.3740944E-01
0.6467628      0.3822809E-01
0.6348513      0.3902215E-01
0.6229291      0.3979277E-01
0.6109948      0.4054113E-01
0.5990473      0.4126850E-01
0.5870859      0.4197638E-01
0.5751106      0.4266634E-01
0.5631221      0.4333974E-01
0.5511213      0.4399785E-01
0.5391096      0.4464181E-01
0.5270886      0.4527254E-01
0.5150598      0.4589077E-01
0.5030249      0.4649710E-01
```

```
0.4909854      0.4709180E-01
0.4789424      0.4767507E-01
0.4668967      0.4824708E-01
0.4548490      0.4880801E-01
0.4428002      0.4935780E-01
0.4307503      0.4989658E-01
0.4187007      0.5042454E-01
0.4066529      0.5094163E-01
0.3946092      0.5144763E-01
0.3825728      0.5194202E-01
0.3705474      0.5242387E-01
0.3585373      0.5289187E-01
0.3465477      0.5334418E-01
0.3345843      0.5377826E-01
0.3226529      0.5419105E-01
0.3107602      0.5457933E-01
0.2989154      0.5493878E-01
0.2871261      0.5526328E-01
0.2753975      0.5554673E-01
0.2637355      0.5578340E-01
0.2521479      0.5596680E-01
0.2406412      0.5608853E-01
0.2292176      0.5614157E-01
0.2178854      0.5611978E-01
0.2066525      0.5601409E-01
0.1955239      0.5581777E-01
0.1845128      0.5552405E-01
0.1736316      0.5512283E-01
0.1628862      0.5460556E-01
0.1522937      0.5396834E-01
0.1418762      0.5320237E-01
0.1316519      0.5229995E-01
0.1216448      0.5125835E-01
0.1118925      0.5007745E-01
0.1024379      0.4875698E-01
0.9332614E-01  0.4730263E-01
0.8461460E-01  0.4572875E-01
0.7636443E-01  0.4405217E-01
0.6862875E-01  0.4229429E-01
0.6144958E-01  0.4048167E-01
0.5485337E-01  0.3864089E-01
0.4884311E-01  0.3679455E-01
0.4339983E-01  0.3496117E-01
0.3848936E-01  0.3315410E-01
0.3407040E-01  0.3138202E-01
0.3009707E-01  0.2965020E-01
0.2652129E-01  0.2795982E-01
0.2329631E-01  0.2630962E-01
0.2038565E-01  0.2469588E-01
0.1775882E-01  0.2311748E-01
0.1538797E-01  0.2157408E-01
0.1324753E-01  0.2006188E-01
0.1131084E-01  0.1857469E-01
0.9552923E-02  0.1710594E-01
0.7954835E-02  0.1564832E-01
```

```
0.6512450E-02  0.1419354E-01
0.5226178E-02  0.1273885E-01
0.4095202E-02  0.1128603E-01
0.3115977E-02  0.9839335E-02
0.2282564E-02  0.8405119E-02
0.1589976E-02  0.6995834E-02
0.1030945E-02  0.5618390E-02
0.5974091E-03  0.4272545E-02
0.2830374E-03  0.2954199E-02
0.8439934E-04  0.1657033E-02
0.1719366E-05  0.3730672E-03
0.3875689E-04 -0.9074793E-03
0.1942123E-03 -0.2195625E-02
0.4658762E-03 -0.3501221E-02
0.8556591E-03 -0.4832680E-02
0.1368932E-02 -0.6196941E-02
0.2013734E-02 -0.7598584E-02
0.2799113E-02 -0.9037504E-02
0.3731685E-02 -0.1050510E-01
0.4817613E-02 -0.1199281E-01
0.6063151E-02 -0.1349804E-01
0.7470988E-02 -0.1502065E-01
0.9040832E-02 -0.1656526E-01
0.1077252E-01 -0.1814454E-01
0.1268000E-01 -0.1977372E-01
0.1479185E-01 -0.2146495E-01
0.1714116E-01 -0.2323119E-01
0.1976169E-01 -0.2508428E-01
0.2268748E-01 -0.2703261E-01
0.2595527E-01 -0.2908518E-01
0.2960721E-01 -0.3125464E-01
0.3369674E-01 -0.3355126E-01
0.3827954E-01 -0.3598108E-01
0.4340612E-01 -0.3854570E-01
0.4911356E-01 -0.4124013E-01
0.5542311E-01 -0.4404605E-01
0.6233735E-01 -0.4693709E-01
0.6982993E-01 -0.4988145E-01
0.7785528E-01 -0.5283856E-01
0.8636618E-01 -0.5577589E-01
0.9530988E-01 -0.5866791E-01
0.1046417     -0.6149853E-01
0.1143203     -0.6426486E-01
0.1242885     -0.6696472E-01
0.1344926     -0.6959100E-01
0.1448846     -0.7213935E-01
0.1554185     -0.7460505E-01
0.1660539     -0.7698262E-01
0.1767553     -0.7926591E-01
0.1874943     -0.8144763E-01
0.1982513     -0.8352168E-01
0.2090106     -0.8548248E-01
0.2197610     -0.8732430E-01
0.2304977     -0.8904094E-01
0.2412219     -0.9062916E-01
```

```
0.2519348     -0.9208688E-01
0.2626359     -0.9341154E-01
0.2733312     -0.9459893E-01
0.2840313     -0.9564913E-01
0.2947434     -0.9656410E-01
0.3054695     -0.9734699E-01
0.3162025     -0.9799649E-01
0.3269527     -0.9850872E-01
0.3377299     -0.9888687E-01
0.3485398     -0.9913974E-01
0.3593590     -0.9926528E-01
0.3702090     -0.9925898E-01
0.3810897     -0.9913264E-01
0.3919835     -0.9888644E-01
0.4028961     -0.9851387E-01
0.4138454     -0.9801911E-01
0.4248266     -0.9741199E-01
0.4358312     -0.9669162E-01
0.4468643     -0.9585851E-01
0.4579335     -0.9491414E-01
0.4690455     -0.9386431E-01
0.4801988     -0.9271487E-01
0.4913915     -0.9146928E-01
0.5026259     -0.9013067E-01
0.5139049     -0.8870341E-01
0.5252300     -0.8719389E-01
0.5365962     -0.8560810E-01
0.5480004     -0.8395013E-01
0.5594417     -0.8222389E-01
0.5709194     -0.8043403E-01
0.5824310     -0.7858582E-01
0.5939719     -0.7668379E-01
0.6055392     -0.7473136E-01
0.6171303     -0.7273196E-01
0.6287430     -0.7068918E-01
0.6403735     -0.6860645E-01
0.6520184     -0.6648644E-01
0.6636744     -0.6433151E-01
0.6753386     -0.6214387E-01
0.6870074     -0.5992544E-01
0.6986773     -0.5767742E-01
0.7103458     -0.5540056E-01
0.7220116     -0.5309511E-01
0.7336738     -0.5076124E-01
0.7453326     -0.4839862E-01
0.7569899     -0.4600665E-01
0.7686492     -0.4358478E-01
0.7803151     -0.4113296E-01
0.7919923     -0.3865197E-01
0.8036845     -0.3614316E-01
0.8153954     -0.3360939E-01
0.8271209     -0.3105826E-01
0.8388394     -0.2850251E-01
0.8505084     -0.2596046E-01
0.8620854     -0.2345345E-01
```

```
 0.8735402     -0.2099805E-01
 0.8848253     -0.1860937E-01
 0.8958723     -0.1631328E-01
 0.9066052     -0.1413474E-01
 0.9169144     -0.1210273E-01
 0.9266817     -0.1025302E-01
 0.9358053     -0.8611374E-02
 0.9441957     -0.7183169E-02
 0.9517876     -0.5964818E-02
 0.9585708     -0.4942371E-02
 0.9645853     -0.4091663E-02
 0.9698990     -0.3384618E-02
 0.9745920     -0.2795426E-02
 0.9787516     -0.2302361E-02
 0.9824612     -0.1886201E-02
 0.9857895     -0.1526513E-02
 0.9887891     -0.1207319E-02
 0.9915041     -0.9181047E-03
 0.9939740     -0.6532547E-03
 0.9962349     -0.4093456E-03
 0.9983176     -0.1834008E-03
  1.000000       0.000000
```

APPENDIX B

PYTHON CODE

B.1 Run Commands: Baseline - 0 Control Sections

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jan  5 15:40:08 2022

@author: justice
"""
import numpy as np
import matplotlib.pyplot as plt
import sys
sys.path.insert(0, '/home/justice/Documents/Thesis/Base-Optimization-Code')
from Ikhana_updated_twist_optimization_conditional_functional import pitch_trim_flap_optimize_functional
from timing import secondsToStr

# Aircraft, Scene, and configuration information
scene_filename = "Ikhana_scene_input.json"
aircraft_json = "Ikhana.json"
aircraft_name = "Ikhana"
num_flaps = 0
upperFlapBound = 25.0
lowerFlapBound = -25.0

# Create titles for all files, plots, and saved results
title = str(num_flaps) + "_Flaps_"  + aircraft_name + "_CL_0.1_0.9__" + secondsToStr()
deflection_title = title + 'FLAP_DEFLECTIONS'
aoa_title = title + "AOA"
horizontal_stabilizer_title = title + "HS_DEFLECTIONS"
CL_CD_graph_filename = title + ".png"
aoa_graph_title = aoa_title + ".png"
hs_graph_title = horizontal_stabilizer_title + ".png"

# Initialize an array to store all results
results = np.zeros((9,6))

# Loop through Cl = 0.1-0.9 and run the optimization
for lift_coeff in range(1,10):
    CL = lift_coeff/10
    index = lift_coeff - 1
    print("---------- Running CL: " + str(CL) + " ----------")

    # Call to optimization code
```

```
    dist_filename, CD, act_CL, act_Cm, aoa, elevator, deflections, solutions_array =
         pitch_trim_flap_optimize_functional(scene_filename, aircraft_json, aircraft_name, num_flaps, CL,
         upperFlapBound, lowerFlapBound)

    # Store Results
    results[index][0] = CL
    results[index][1] = CD
    results[index][2] = act_Cm
    results[index][3] = aoa
    results[index][4] = elevator
    results[index][5] = act_CL


# Print out and save results.
print('CL   CD   Cm   alpha   elevator   act_CL')
print(results)
np.savetxt(title, results, header='CL   CD   Cm   alpha   elevator   act_CL')

# Print out final CL and CD arrays
print('\n------------------------------------')
print("\nCL\n")
print(results[:,0])
print("\nCD\n")
print(results[:,1])


# --- Make and save plots ---
# Plot CL v CD and save
plt.figure(0)
plt.plot(results[:,0], results[:,1])
plt.title(title)
plt.xlabel("CL")
plt.ylabel("CD")
plt.savefig(CL_CD_graph_filename)

# Plot Cl v alpha
plt.figure(1)
plt.plot(results[:,0], results[:,3])
plt.title(aoa_title)
plt.xlabel("CL")
plt.ylabel("Alpha, deg")
plt.savefig(aoa_graph_title)

# Plot CL v Horizontal stabilizer angle
plt.figure(2)
plt.plot(results[:,0], results[:,4])
plt.title(horizontal_stabilizer_title)
plt.xlabel("CL")
plt.ylabel("Horizontal Stabilizer, deg")
plt.savefig(hs_graph_title)
```

B.2 Run Commands: With Control Sections, Looping Through $C_L = 0.1 - 0.9$

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jan  5 15:40:08 2022

@author: justice
"""
import numpy as np
import matplotlib.pyplot as plt
import sys
sys.path.insert(0, '/home/justice/Documents/Thesis/Base-Optimization-Code')
from Ikhana_updated_twist_optimization_conditional_functional import pitch_trim_flap_optimize_functional
from timing import secondsToStr

'''
This code uses the MachUpX/SLSQP optimization code to determine the trimmed drag coefficient
for a range of lift coefficients (CL 0.1 - 0.9). This is done by passing the aircraft and scene
jsons into MachUpX to be initialized. The configuration settings are also specified
(the number of control points (num_flaps), upper bound and lower bound on alpha and horizontal stab deflections)


Then all titles for outputs are generated based off of the aircraft name. The optimization
code is then called in a loop where one lift coefficient is solved at a time. After each lift
coefficient iteration is solved for the drag coefficient then the next lift coefficient is
passed in and the solution (camber for each control point, horizontal stabilizer deflection, angle of attack)
is used as the initial guess for the new lift coefficient.

ie:
    CL 0.1 run with initial guess of all zeros
    -Returns [0.25, 0.25, 0.4, 0.4, 3.26, 5.7]
    (cp @ 0.0 camber, cp @ 0.5 camber, cp @ 0.5 camber, cp @ 1.0 camber, hs defl in deg, alpha in deg)

    then CL 0.2 is run with initial guess of [0.25, 0.25, 0.4, 0.4, 3.26, 5.7]
    - returns xxxxxx

    then CL 0.3 is run with initial guess of xxxxxx

    and so on until CL 0.9 has been run.

After all lift coefficients (CL 0.1 - 0.9) have been run then the process is ran in revers.
Meaning that the program starts at CL 0.8 and solves using the solution from the CL 0.9 iteration.
If the new result is a lower drag coefficient then the new solution replaces the old CL 0.8 solution.
If not then the CL 0.8 solution is unchanged.

The best result for CL 0.8 (either the first solution, or the new solution achieved with initial guess of CL 0.9 results)
is then used as the initial guess for CL 0.7 and again if the solution is better than the "first"
CL 0.7 solution the first solution is replaced with the new solution and the process continues all the way to CL 0.1.

**I used "first" when referring to the first solution because it is the result obtained from the
CL xx iteration of the first for loop in this program. Inside of the optimization in the first
for loop the optimization process could have been run multiple times. This is because the optimization
```

```
code was written in such a way that once the optimization has returned, that solution can immediately
be plugged back into the optimization process as an initial guess. If this is done then the solution
will be plugged back in until it stops changing within some error limit. This functionality does not have
to be enabled, but it is in this code. So, I used "first" because the "first" solution could have been the
result of multiple optimization calls within the call to the optimization function, but it was the first
solution returned for the given lift coefficient in this code.

I decided to go "up" from CL 0.1 - 0.9 and then "down" from CL 0.9 - 0.1 to help
ensure that the results of the optimization didn't get stuck in a local minima. There
were instances before I implemented this up/down approach where looking at the results
for CL 0.1 - 0.9 it looked like there were almost two different solution valleys achieved.
The first part of the CL v CD curve would be along one parabolic function, then it would
jump to another parabolic function at some intermediate CL (indicating a different solution valley).

By going up then down it helped get all of the CD values on the same parabolic curve
and in the same solution valley.

'''


# Aircraft, Scene, and configuration information
scene_filename = "Ikhana_scene_input.json"
aircraft_json = "Ikhana.json"
aircraft_name = "Ikhana"
num_flaps = 2
upperFlapBound = 25.0
lowerFlapBound = -25.0

# Create titles for all files, plots, and saved results
title = str(num_flaps) + "_Flaps_" + aircraft_name + "_CL_0.1_0.9__" + secondsToStr()
deflection_title = title + '__FLAP_DEFLECTIONS'
aoa_title = title + "__AOA"
horizontal_stabilizer_title = title + "__HS_DEFLECTIONS"
solution_array_title = title + "__SOLUTIONS_ARRAY"
changed_title = title + "__CHANGED"

CL_CD_graph_filename = title + ".png"
flap_schedule_graph_filename = deflection_title + ".png"
aoa_graph_title = aoa_title + ".png"
hs_graph_title = horizontal_stabilizer_title + ".png"

# Initialize an array to store all results, previous solutions, and if the solution changed on the second time
     through
results = np.zeros((9,6))
previous_solution_array = np.zeros((9,num_flaps+2))
has_changed = np.zeros((9,1)) # 0 indicates no change, 1111 indicates change

#-------------------------------  Going "Up"  -------------------------------
# Go through from CL 0.1 to 0.9.
#  For CL 0.1 use initial guess of all 0's. After that,
#  use the previous CL's solution as the initial guess.
for lift_coeff in range(1,10):
    CL = lift_coeff/10
    index = lift_coeff - 1
    print("---------- Running CL: " + str(CL) + " ----------")
```

```
    if lift_coeff == 1:
        # Run with NO initial deflections (ie: they will be assumed to be zero)
        dist_filename, CD, act_CL, act_Cm, aoa, elevator, deflections, solution_array = pitch_trim_flap_optimize_functional(scene_filename, aircraft_json, aircraft_name, num_flaps, CL, upperFlapBound, lowerFlapBound, run_mult_solutions=(True))
        prev_solution = solution_array
        previous_solution_array[index,:] = solution_array
    else:
        # Run with initial deflections set to the previous solution (Start at last solution as initial guess)
        dist_filename, CD, act_CL, act_Cm, aoa, elevator, deflections, solution_array = pitch_trim_flap_optimize_functional(scene_filename, aircraft_json, aircraft_name, num_flaps, CL, upperFlapBound, lowerFlapBound, run_mult_solutions=(True), initial_defl=(prev_solution))
        prev_solution = solution_array
        previous_solution_array[index,:] = solution_array

    # Store results
    results[index][0] = CL
    results[index][1] = CD
    results[index][2] = act_Cm
    results[index][3] = aoa
    results[index][4] = elevator
    results[index][5] = act_CL

    # Store deflections for CL 0.1-0.9
    if (lift_coeff == 1): # Create the all_deflections array
        rows, cols = np.shape(deflections)
        all_deflections = np.zeros((rows, 10))
        all_deflections[:,0:2] = deflections
    else: # Add to the all_deflections array
        # Need to go one above the index (ie: lift_coeff)
        # all_deflections goes: Span Loc   0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   0.9
        all_deflections[:,lift_coeff] = deflections[:,1]

# Make and save copies of the original results and deflections
results_from_going_up = results.copy()
deflections_going_up = all_deflections.copy()
orig_prev_solutions_array = previous_solution_array.copy()

results_up_title = title + "__UP"
deflections_up_title = results_up_title + "__deflections"
prev_sol_title = results_up_title + "__orig_solutions"

np.savetxt(results_up_title ,results_from_going_up, header='CL   CD   Cm   alpha   elevator   act_CL')
np.savetxt(deflections_up_title, deflections_going_up, header='Span Loc   0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   0.9')
np.savetxt(prev_sol_title, orig_prev_solutions_array, header='Flaps...  Elevator Alpha')



#------------------------------  Going "Down"  ------------------------------
# Now go from CL 0.9 to CL 0.1 using the "previous" CL's solution as the initial guess.
#  Work way down, if the solution is better than that from going up, then update the results.
#  Also, update the hasChanged array so I know which ones were updated.
for down_num in range(9,1,-1):
```

```
    CL = (down_num - 1)/10.0
    down_index = down_num - 1
    CL_index = down_num - 2

    # Get results from CL above the CL being run (ie: results for CL 0.9 used for running CL 0.8)
    # Need to use down_index because that is down_num - 1 which will give the prev solution for down_num
    prev_results = previous_solution_array[down_index,:]
    dist_filename, CD, act_CL, act_Cm, aoa, elevator, deflections, solution_array = pitch_trim_flap_optimize_functional(scene_filename, aircraft_json, aircraft_name, num_flaps, CL, upperFlapBound, lowerFlapBound, run_mult_solutions=(True), initial_defl=(prev_results))

    # If CD is lower, replace results & deflections
    if (CD < results[CL_index][1]):
        results[CL_index][0] = CL
        results[CL_index][1] = CD
        results[CL_index][2] = act_Cm
        results[CL_index][3] = aoa
        results[CL_index][4] = elevator
        results[CL_index][5] = act_CL

        # Update all_deflections.
        # Need to go one above the CL_index to get the proper location (ie: down_index)
        # all_deflections goes: Span Loc   0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   0.9
        all_deflections[:,down_index] = deflections[:,1]

        # Update the previous solutions array
        previous_solution_array[CL_index,:] = solution_array

        # update the has_changed array to indicate there was a change
        has_changed[CL_index] = 1111

#---------------------------------------------------------------------------
#------------------------  Print & Save Results/Plots  ------------------------
#---------------------------------------------------------------------------

print('CL   CD   Cm   alpha   elevator   act_CL')
print(results)
np.savetxt(title, results, header='CL   CD   Cm   alpha   elevator   act_CL')
np.savetxt(deflection_title, all_deflections, header='Span Loc   0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   0.9')
np.savetxt(solution_array_title, previous_solution_array, header='Flaps...  Elevator Alpha')
np.savetxt(changed_title, has_changed, header="CL  Status")

# Print out the CL and CD arrays
print('\n-----------------------------------')
print("\nCL\n")
print(results[:,0])
print("\nCD\n")
print(results[:,1])

# Plot CD v CL
plt.figure(0)
plt.plot(results[:,0], results[:,1])
plt.title(title)
plt.xlabel("CL")
```

```
plt.ylabel("CD")
plt.savefig(CL_CD_graph_filename)

# Plot Camber schedule for all CL's (0.1-0.9)
plt.figure(1)
plt.plot(all_deflections[:,0], all_deflections[:,1], label = 'CL 0.1')
plt.plot(all_deflections[:,0], all_deflections[:,2], label = 'CL 0.2')
plt.plot(all_deflections[:,0], all_deflections[:,3], label = 'CL 0.3')
plt.plot(all_deflections[:,0], all_deflections[:,4], label = 'CL 0.4')
plt.plot(all_deflections[:,0], all_deflections[:,5], label = 'CL 0.5')
plt.plot(all_deflections[:,0], all_deflections[:,6], label = 'CL 0.6')
plt.plot(all_deflections[:,0], all_deflections[:,7], label = 'CL 0.7')
plt.plot(all_deflections[:,0], all_deflections[:,8], label = 'CL 0.8')
plt.plot(all_deflections[:,0], all_deflections[:,9], label = 'CL 0.9')
plt.title(title)
plt.xlabel('Span')
plt.ylabel("Camber")
plt.legend(loc='right')
plt.savefig(flap_schedule_graph_filename)

# Plot CL v Alpha
plt.figure(2)
plt.plot(results[:,0], results[:,3])
plt.title(aoa_title)
plt.xlabel("CL")
plt.ylabel("Alpha, deg")
plt.savefig(aoa_graph_title)

# Plot CL v Horizontal stabilizer deflections
plt.figure(3)
plt.plot(results[:,0], results[:,4])
plt.title(horizontal_stabilizer_title)
plt.xlabel("CL")
plt.ylabel("Horizontal Stabilizer, deg")
plt.savefig(hs_graph_title)
```

B.3 Run Commands: Single Lift Coefficient

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jan  5 15:40:08 2022

@author: justice
"""
# Get optimization code from different folder
import sys
sys.path.insert(0, '/home/justice/Documents/Thesis/Base-Optimization-Code')
from Ikhana_updated_twist_optimization_conditional_functional import pitch_trim_flap_optimize_functional
import numpy as np

# Set Desired CL
CL = 0.6

# Give aircraft and scene json names as well as aircraft name
scene_filename = "Ikhana_scene_input.json"
aircraft_json = "Ikhana.json"
aircraft_name = "Ikhana"

# Specify number of flaps
num_flaps = 4

# Specify upper and lower bounds for elevator and angle of attack deflections
upperFlapBound = 25.0
lowerFlapBound = -25.0

# Specify initial guess (number of control points + elevator + angle of attack)
init_guess = np.array([-2.0, -1.0, -2.0, 2.0, -4.0, 3.5])

# Run optimization
dist_filename, CD, act_CL, act_Cm, aoa, elevator, deflections, solution_array =
    pitch_trim_flap_optimize_functional(scene_filename, aircraft_json, aircraft_name, num_flaps, CL,
    upperFlapBound, lowerFlapBound, initial_defl=(init_guess))
```

B.4 Optimization Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Oct 20 13:53:39 2021

@author: Justice Schoenfeld
"""

import machupX as mx
import numpy as np
import matplotlib.pyplot as plt
import json
import scipy as sp
import copy
import jsonpickle
from Ikhana_join import create_span_fraction_array, double_repeat_and_join
from airfoil_functional_creation import create_Ikhana_airfoils_function_dict
from Ikhana_cos_clustering_array import create_cos_cluster_array
import timing
from timing import secondsToStr


def pitch_trim_flap_optimize_functional(orig_scene_filename, orig_aircraft_json_filename, aircraft_name,
    num_flaps, CL_to_set, upDeflBound, lowDeflBound, run_mult_solutions = False, initial_defl = None, dragType =
    "Total", write_results = True, print_results = False, show_plots = False, dump_forces_and_moments = False):
    '''
    This code is used to pitch trim the given aircraft and then find the minimum drag
    at the specified lift coefficient using the SLSQP method to minimize the drag value
    returned from the MachUpX forces and moments calculation. This code has the capability to
    re-run a solution multiple times until the difference between consecutive solutions is below
    an error threshold (similar to Optix, see note below with run_mult_solutions flag).

    This code will also write the results obtained out to txt files for saving and data
    analysis.

    Once the MachUpX files are read in and MachUpX scene class has been created, the aircraft
    in the scene class is manipulated to get to the trimmed state, then the optimization is run.

    The first part of this function is administrative: creating filenames, getting the scene
    class set up and properly configured, creating the initial x array to be passed to
    scipy.optimize.minimize, etc..

    Then the actual optimization function is defined, followed by the cost function
    definition. The cost function is used in the scipy.optimize.minimize call.

    After the cost function, this function extracts the needed reporting information:
        - angle of attack
        - elevator mounting angle
        - CD, CL, Cm
    and creates the output text files for saving the results of the optimization.

    Parameters
    ----------
```

```
    orig_scene_filename : string
        Filename of the aircraft scene json.
    orig_aircraft_json_filename : string
        Filename of the aircraft json.
    aircraft_name : string
        Name of the aircraft as given for the 'tag' in the aircraft scene json.
    num_flaps : float
        Desired number of flaps/control points to be used.
    CL_to_set : float
        Desired lift coefficient.
    upDeflBound : float
        Upper bound on the elevator and angle of attack deflections.
    lowDeflBound : float
        Lower bound on the elevator and angle of attack deflections.
    run_mult_solutions : boolean, optional
        Whether or not to take the solution from scipy.optimize.minimize and plug it back in as an initial guess before this function returns. The default is False.
    initial_defl : array, [float], optional
        An array of the initial deflections for the optimization. Length should be = num_flaps + 2 (ie: 4 control points would give [x, x, x, x, x, x]). The default is None.
    dragType : string, optional
        What type of Drag to use ('Total', 'Inviscid', or 'Viscous'). The default is "Total".
    write_results : boolean, optional
        Whether or not to write the results out to files. The default is True.
    print_results : boolean, optional
        Whether or not to print out results to the console. The default is False.
    show_plots : boolean, optional
        Whether or not to show a plot of the normalized lift distribution. The default is False.
    dump_forces_and_moments : boolean, optional
        Whether or not display forces and moments in nice json format. The default is False.

    Returns
    -------
    distributions_filename : string
        The filename for the distributions file, returned so that it can be passed to a function that generates the lift distribution.
    CD : float
        The value of the drag coefficient from the MachUpX calculated forces and moments.
    fm_CL : float
        The lift coefficient from the MachUpX calculated forces and moments.
    fm_Cm : float
        The pitching moment coefficient from the MachUpX calculated forces and moments.
    aoa : float
        The angle of attack (deg) needed to pitch trim the aircraft.
    elevator : float
        The horizontal stabilizer rotation angle (deg) needed to pitch trim the aircraft using an all flying tail.
    deflection_array : array, [float]
        The deflections used to achieve the minimum drag at the desired lift coefficient.
    solution.x : array, [float]
        The solution x array from scipy.optimize.minimize. Includes the deflections as well as the elevator and angle of attack values.
    '''
    if (dragType != "Total") and (dragType != "Inviscid") and (dragType != "Viscous"):
        print("Invalid dragType entered! Drag Type must be either 'Total' (default), 'Inviscid', or 'Viscous'.")
```

```
        return

    # --- Create Filenames ---
    partitioned_file_name = orig_scene_filename.partition('.')
    output_title = str(num_flaps) + "_FLAPS_" + partitioned_file_name[0] + "_CL_" + str(CL_to_set) + "__" +
        secondsToStr()
    force_moment_output_filename = "F_M_" + output_title + ".json"
    distributions_filename = "distributions_" + output_title

    # --- If not Base Case (0 control points) then create span fraction array ---
    if (num_flaps > 0):
        span_frac_array = create_span_fraction_array(num_flaps)

    length_x_array = num_flaps + 2    # Number of control points + elevator + alpha
    end_flap_index = num_flaps        # Index of last control point in x array
    elevator_index = num_flaps        # Index of the elevator value in x array
    aoa_index = length_x_array - 1    # Index of the aoa value in x array

    # Create unique scene and aircraft jsons for the given CL
    scene_filename = str(CL_to_set) + "_" + orig_scene_filename

    # Create aircraft dictionary
    orig_aircraft_dict = json.load(open(orig_aircraft_json_filename))

    # If not the Baseline case (0 control points) then set cosine clustering points and functions for CD, CL, Cm
    if (num_flaps > 0):
        # Set the cosine clustering points for the number of inboard and outboard flaps
        orig_aircraft_dict['wings']['main_wing']['grid']['cluster_points'] = create_cos_cluster_array(num_flaps)

        # Replace airfoil poly_fits with function calls (functional)
        orig_aircraft_dict['airfoils'] = create_Ikhana_airfoils_function_dict()

    # Create scene dictionary
    orig_scene_dict = json.load(open(orig_scene_filename))

    # Load state from scene json & save original horizontal tail twist
    scene_state_dict = copy.deepcopy(orig_scene_dict)["scene"]["aircraft"][aircraft_name]["state"]
    orig_aircraft_horizontal_twist = copy.deepcopy(orig_aircraft_dict)["wings"]["horizontal_tail"]["twist"]

    # Remove the aircraft so that I can add the aircraft dictionary with functions
    scene_dict = copy.deepcopy(orig_scene_dict)
    scene_dict['scene']['aircraft'].pop(aircraft_name)

    # --If desired, format and print the json after changes have been made. Not currently used, but wanted to
        keep functionality.
    # def notSerializable(thingToPickle):
    #     name = jsonpickle.encode(thingToPickle)
    #     return name

    # print(json.dumps(scene_dict, indent = 4))
    # print("\n\n------------------------")
    # print(json.dumps(orig_aircraft_dict, indent = 4, default = notSerializable))
    # print("\n\n------------------------")


```

```
    # Create scene and add scene with functions for airfoils
    my_scene = mx.Scene(scene_dict)
    my_scene.add_aircraft(aircraft_name, orig_aircraft_dict, scene_state_dict)

    # --Can be used to display wireframe of aircraft if so desired. Not currently used, but wanted to keep functionality.
    #my_scene.display_wireframe()

    # Declaration of optimization function, this function makes the call to scipy.optimize.minimize
    def optimize_twist_with_pitch_trim(CL_to_set):
        '''
        This is the actual function where the drag is minimized. All of the set up has been
        done prior to this point. This function then sets up the bounds and constriants for the
        optimization and makes the call to the scipy.optimize.minimize method.

        After the minimization has happend, this function can plug the solution back into
        the minimzation if so desired and do so iteratively until the new solution is within
        a given error margin of the old solution (the solution has converged). This functionality
        was inspired by Dr Hunsaker and Optix.

        Once a final solution has been achieved, the settings needed for the final
        solution are returned from this function so that the final results can be
        calculated and returned to the user.

        ** This function MUST stay within the pitch_trim_flap_optimize_functional
           function because of how the scene class in MachUpX works. This function is
           inside of the parent function so that the scene class is within scope. If it
           were to be moved otuside of the parent class it is not possible to keep
           the MachUpX scene class in scope and this code would break.

        Parameters
        ----------
        CL_to_set : float
            The desired CL to solve for the minimum drag coefficient.

        Returns
        -------
        solution : OptimizeResult object
            An OptimizeResult object containing the result of the minimization.
        deflection_array : array, [float]
            An array of the camber deflections.
        forces_and_moments : dictionary
            A dictionary of all forces and moments calcuated by MachUpX.
        CD : float
            The value of the drag coefficient calculated using MachUpX's forces and moments solver.
        aoa : float
            The angle of attack (deg) needed to pitch trim the aircraft.
        elevator : float
            The horizontal stabilize mounting angle (deg) needed to pitch trim the aircraft with an all moving tail.
        twist_data_post_solution : array, [[float], [float]]
            A (nx2 array (2D) [span location, twist] with the updated twist used to change the mounting angle (deg) on the horizontal stabilizer in order to pitch trim the aircraft.
        calc_CL : float
            The lift coefficient from the MachUpX calculated forces and moments.
```

```
        calc_Cm : float
            The pitching moment coefficient from the MachUpX calculated forces and moments.

        '''
        # Whether to use zeros as initial guess or the passed in initial deflections as the initial guess
        if initial_defl is None: # If no initial_defl given use 0 as initial guess
            x = np.zeros(length_x_array) # Flaps, Elevator, Alpha
        else: # Use the initial deflections given as the initial guess, if of proper size (num_flaps + 2).
            if len(initial_defl) == length_x_array:
                print('Initial Deflections: \n')
                print(initial_defl)
                print('\n')
                x = initial_defl
            else: # initial_defl given is of improper length and CANNOT be used.
                print("Invalid initial_deflection array.\n")
                print("Length needs to be " + str(length_x_array) + "\n")
                print("Entered length is " + str(len(initial_defl)))
                return

        # Set the bounds for the optimization. Bounds apply to elevator and angle of attack
        lowerBoundsArray = np.ones(length_x_array)*lowDeflBound
        lowerBoundsArray[elevator_index] = -np.inf
        lowerBoundsArray[aoa_index] = -np.inf

        upperBoundsArray = np.ones(length_x_array)*upDeflBound
        upperBoundsArray[elevator_index] = np.inf
        upperBoundsArray[aoa_index] = np.inf

        bnds = sp.optimize.Bounds(lowerBoundsArray, upperBoundsArray, keep_feasible = True)

        # Set the constraints necessary to pitch trim the aircraft. The constraints are on CL and Cm
        constr1 = {"type" : "eq",
                  "fun" : twist_cost_function,
                  "args" : (CL_to_set, "moment")}
        constr2 = {"type": "eq",
                   "fun" : twist_cost_function,
                   "args" : (CL_to_set, "lift")}
        constr = [constr1, constr2]

        # --- CALL TO OPTIMIZATION ---
        solution = sp.optimize.minimize(twist_cost_function, x, args = (CL_to_set), bounds = bnds, constraints = constr)

        # Plug the solution back in as initial guess and re-run optimization if desired. (This functionality mimics Optix)
        if run_mult_solutions:
            epsilon = 5.0; # Error initial value
            prev_solution = solution
            x = prev_solution.x
            run_mult_iter = 1
            print("Iteration " + str(run_mult_iter) + "\n")
            print(str(solution) + "\n\n")
            # Run until the difference in solutions is smaller than 0.0001
            while(abs(epsilon) > 0.0001): # By using the norm of the epsilon vector a threshold of 0.0001 requires all individual differences be at or below 1e-5
```

```python
                run_mult_iter += 1
                solution = sp.optimize.minimize(twist_cost_function, x, args = (CL_to_set), bounds = bnds, constraints = constr)
                epsilon = np.linalg.norm(prev_solution.x - solution.x)
                prev_solution = solution
                x = solution.x
                print("Iteration " + str(run_mult_iter) + "\n")
                print(str(solution) + "\n\n")

                '''
                The if statement and while loop above help ensure that we have actually reached the minimum value with the optimization.
                The optimization is currently running a SLSQP with bounds. As part of the SLSQP scheme the first derivative is calculated
                directly and then the differences in the first derivative are used to calculate the second derivative.

                Calculating the second derivative in this manner means that error builds up in the Jacobian inside the SLSQP optimization
                and the result may not be the actual minimum. By taking the first solution and plugging it back in as the initial guess for
                a second optimization essentially clears the error from the optimization and the optimization starts from the previous result.
                Then by comparing the solutions and setting a threshold for the difference between two consecutive solutions I can run the
                optimization as many times as necessary, each time starting at the result of the previous solution, to get to what is the "true"
                solution where my result between optimization runs isn't changing significantly.

                This was suggested by Dr Hunsaker and is similar to what he implemented in Optix, which is written for Fortran.
                '''

        # Store the angle of attack and elevator deflections
        aoa = solution.x[aoa_index]                                                 # deg
        elevator = solution.x[elevator_index]                                       # deg

        # --- Update the twist on the horizontal tail by changing the mounting angle
        aircraft_dict_post_solution = copy.deepcopy(orig_aircraft_dict)

        # Add to the mounting angle
        twist_data_post_solution = copy.deepcopy(orig_aircraft_horizontal_twist)
        for row in range(0,len(twist_data_post_solution)):
            twist_data_post_solution[row][1] += elevator                            # deg

        # Update the twist in the aircraft dictionary
        aircraft_dict_post_solution["wings"]["horizontal_tail"]["twist"] = twist_data_post_solution

        # Update the angle of attack in the scene state
        scene_state_dict["alpha"] = aoa                                             # deg

        # Re-initialize MachUpX with new angle of attack and "twist" (tail mounting angle)
        my_scene = mx.Scene(scene_dict)
        my_scene.add_aircraft(aircraft_name, aircraft_dict_post_solution, scene_state_dict)

```

```
    deflection_array = []
    if(num_flaps > 0): # If using control points, set the deflections using values from optimization.
        deflection_array = double_repeat_and_join(span_frac_array, solution.x[0:end_flap_index])
        deflections = {"flaps1" : deflection_array}
        my_scene.set_aircraft_control_state(control_state = deflections)        # deg

    # Calculate Forces & Moments as well as the Distributions and save the results
    forces_and_moments = my_scene.solve_forces(filename = force_moment_output_filename)
    my_scene.distributions(filename = distributions_filename)

    # Get the CL and Cm values and print them. They will only be printed at the end of each CL that is run, if run in a loop.
    calc_CL = forces_and_moments[aircraft_name]['total']['CL']
    calc_Cm = forces_and_moments[aircraft_name]['total']['Cm']
    print("CL: " + str(forces_and_moments[aircraft_name]["total"]["CL"]))
    print("Cm: " + str(forces_and_moments[aircraft_name]["total"]["Cm"]))

    # Get the correct drag value from the forces and moments
    if dragType == "Inviscid":
        CD = forces_and_moments[aircraft_name]["inviscid"]["CD"]["total"]
    elif dragType == "Viscous":
        CD = forces_and_moments[aircraft_name]["viscous"]["CD"]["total"]
    else:
        CD = forces_and_moments[aircraft_name]["total"]["CD"]

    # Plot normalized washout with respect to span location if desired.
    if show_plots:
        ''' Plot normalized washout from optimization. Normalize w/ respect to last deflection (-1 index)'''
        span_locations = deflections["flaps1"][:,0]             # Get the span locations that correspond to deflections
        normalized_deflections = deflections["flaps1"][:,1]     # Gets all deflections
        normalized_deflections /= normalized_deflections[-1]    # Normalizes w/ respect to last deflection

        plt.plot(span_locations, normalized_deflections, label = "Optimized Values")
        plt.show()

    # Return the results of the optimization at the given CL
    return solution, deflection_array, forces_and_moments, CD, aoa, elevator, twist_data_post_solution, calc_CL, calc_Cm


''' Optimizer function for minimizing drag by "twisting" the wing '''
def twist_cost_function(x, desired_CL ,flag = "drag"):
    '''
    The cost function to be optimized in order to minimize drag. Also used for
    the constraints.

    This function can be used for the constraints to change the horizontal
    stabilizer mounting angle and angle of attack (both in degrees) in
    order to pitch trim the aircraft.

    Or this function can be used to find the drag coefficient to be minimized.
    When the drag coefficient is found with this function, it's value is scaled
    by 100.0. This was done because it was found that the CL constraint could
    dominate the minimization, since the CL is often 1 to 2 orders of magnitude
```

```
        larger than CD. By scaling the drag coefficient it brings the CD value closer
        to the order of magnitude of CL and it was found that better results were obtained.

        ** This function MUST stay within the pitch_trim_flap_optimize_functional
           function because of how the scene class in MachUpX works. This function is
           inside of the parent function so that the scene class is within scope. If it
           were to be moved outside of the parent class it is not possible to keep
           the MachUpX scene class in scope and this code would break.

        Parameters
        ----------
        x : array, [float]
            x array from scipy.optimize.minimize.
        desired_CL : float
            The desired lift coefficient.
        flag : string, optional
            Which value to return, either 'drag', 'lift', or 'moment'. The default is "drag".

        Returns
        -------
        value : float
            The value of CL, Cm, or CD depending on the flag that was given. (**Note CD will be scaled by 100.0 to bring to same order of magnitude as CL constraint)

        '''
        # --- Update the twist on the horizontal tail by changing the mounting angle
        aircraft_dict = copy.deepcopy(orig_aircraft_dict)

        # Pull in original twist info and add new optimized mounting angle
        twist_data = copy.deepcopy(orig_aircraft_horizontal_twist)
        for row in range(0,len(twist_data)):
            twist_data[row][1] += x[elevator_index]                                  # deg

        # Set new twist
        aircraft_dict["wings"]["horizontal_tail"]["twist"] = twist_data

        # --- Set the angle of attack
        scene_state_dict["alpha"] = x[aoa_index]                                     # deg

        # Re-initialize MachUpX with new "twist" (tail mounting angle)
        my_scene = mx.Scene(scene_dict)
        my_scene.add_aircraft(aircraft_name, aircraft_dict, scene_state_dict)

        # --- Change the flap deflections if num_flaps > 0
        if (num_flaps > 0):
            deflection_array = double_repeat_and_join(span_frac_array, x[0:end_flap_index])
            deflections = {"flaps1" : deflection_array}
            my_scene.set_aircraft_control_state(control_state = deflections)        # deg

        # Call for forces and moments to get CL and Cm for constraints or CD for value to minimize.
        forces_and_moments = my_scene.solve_forces(verbose=False)

        # Get the appropriate value (either a constraint or the minimization value)
        if flag == "moment": # Get Cm for constraint
            value = forces_and_moments[aircraft_name]["total"]["Cm"]
```

```
        elif flag == "lift": # Get CL for constraint
            temp_value = forces_and_moments[aircraft_name]["total"]["CL"]
            value = abs(temp_value - desired_CL)
        else: # Return CD
            if dragType == "Inviscid":
                value = forces_and_moments[aircraft_name]["inviscid"]["CD"]["total"]
            elif dragType == "Viscous":
                value = forces_and_moments[aircraft_name]["viscous"]["CD"]["total"]
            else:
                value = forces_and_moments[aircraft_name]["total"]["CD"]

            # Scale the drag value so that it is on the same order of magnitude as CL and helps the optimization
            value *= 100.0

        return value

    ############################
    ######  Run Analysis  ######
    ############################
    # Get results from the optimization call
    solution, deflection_array, forces_and_moments, CD, aoa, elevator, hs_twist_data, fm_CL, fm_Cm =
         optimize_twist_with_pitch_trim(CL_to_set)

    # Write results out to a file
    if write_results:
        output = open(output_title, 'w')
        output.write("CL: " + str(CL_to_set) + "\n")
        if type(initial_defl) != type(None):
            output.write("Initial Defl: " + str(initial_defl) + "\n")
        output.write("Num Flaps: " + str(num_flaps) + "\n")
        output.write("Scene File Name: " + scene_filename + "\n")
        output.write(str(solution) + "\n")
        output.write(str(deflection_array))
        output.write("\n" + dragType + " Drag (CD): " + str(CD) + "\n")
        output.write("Calc CL: " + str(fm_CL) + "\n")
        output.write("Calc Cm: " + str(fm_Cm) + "\n")
        output.write("Angle of Attack: " + str(aoa) + " (deg)\n")
        output.write("Elevator: " + str(elevator) + " (deg)\n")
        output.write("\nHorizontal Stabilizer Twist: \n" + str(hs_twist_data) + "\n")
        if dump_forces_and_moments:
            output.write(json.dumps(forces_and_moments, indent = 4))
        output.close()

    # Print results out if so desired.
    if print_results:
        print(solution)
        print("\nDeflection Array: \n")
        print(deflection_array)
        print("\nDrag: ", CD)
        if dump_forces_and_moments:
            print(json.dumps(forces_and_moments, indent = 4))

    # Return the any values necessary for looping through multiple CL's
    return distributions_filename, CD, fm_CL, fm_Cm, aoa, elevator, deflection_array, solution.x
```

B.5 Supporting Code: Ikhana_join.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 17 14:08:02 2021

@author: justice
"""

import numpy as np

'''
This file is used to create the twist distribution array for MachUpX used for the
Ikhana. This code is specific to the Ikhana.
Also, this code generates the twist distribution such that we get
rectangular flaps. If you specify the twist distribution as:
     [span_frac, twist]
    ([[0.0, 0],
      [0.2, 1],
      [0.4, 2],                     Linearly Interpolated Example
      [0.6, 3],
      [0.8, 4],
      [1.0, 5]])
    you will get a linear extrapolation between the span fractions you specified.
    So at a span fraction of 0.1 your twist will be 0.5, at span frac 0.5 your twist
    will be 2.5 and so forth.

    We don't want the linear extrapolation between specified locations. We want
    rectangular flaps like you would get on an actual airplane. So to do this, we
    needed to double up the span fractions and the twist values, like this:
    ([[0.0, 0],
      [0.0, 1],
      [0.2, 1],
      [0.2, 2],
      [0.4, 2],                     Rectangular Flap Example
      [0.4, 3],
      [0.6, 3],
      [0.6, 4],
      [0.8, 4],
      [0.8, 5],
      [1.0, 5]])
    This way between 0.0 and 2.0 we maintain a constant value of 1.
    Between 2.0 and 4.0 we maintain a constant value of 2 and so on so forth.

In order to dynamically generate the doubled up span fraction list and the associated
twist's, the following code was written such that the user can specify the desired
number of inboard and outboard flaps.

Also, the code can be used with scipy.optimize.minimize by using the "create_span_fraction_array"
function at the beginning of the optimization to get the span fraction distribution that
will yield the desired number of inboard and outboard flaps.

Then each time the optimization runs, call the "double_and_repeat" function to
get the x array used in scipy.optimize.minimize in the appropriate form.
```

```

Once you have the x array doubled and repeated you can use the "join" function
to combine the span fraction array with the doubled x array and get your twist
distribution, like that in the Rectangular Flap Example, that can be passed into
MachUpX.
'''


def create_span_fraction_array(num_control_points):
    '''This function creates the span fraction array for the NASA Ikhana.
    The user can specify the number of control points and then the span fraction
    array will be created such that there will be rectangular flaps in between
    each span frac.

    Parameters
    ----------
    num_control_points : int, the number of control points desired

    Returns
    -------
    span_frac_list : list, all of the span fractions necessary to create the
    desired number of rectangular flaps.

    '''
    max_span_frac = 1.0

    step = max_span_frac / num_control_points

    span_frac_list = []
    span_frac_list.append(0.0)
    span_frac_list.append(0.0)
    last_frac = 0.0

    for x in range(0,num_control_points-1):
        span_frac_list.append(last_frac + step)
        span_frac_list.append(last_frac + step)
        last_frac += step

    span_frac_list.append(max_span_frac)

    return span_frac_list


def double_and_repeat(array):
    '''This function creates an array twice the length of the orignal by repeating
    each value in the original array. For example:
        Given:      [1,2,3,4]
        Returns:    [0,1,1,2,2,3,3,4,4]

    Parameters
    ----------
    array : list or array, the original array you want repeated

    Returns
    -------
```

```
    doubled_array : list, double the length of original array and with values repeated

    '''
    doubled_array = []
    doubled_array.append(0.0)
    for val in array:
        doubled_array.append(val)
        doubled_array.append(val)

    return doubled_array


def join(a, b):
    '''This function takes two strings and merges them together. I wrote it so that
    I could dynamically create span distributions. The A matrix represents the span
    fractions, and the B matrix represents the twist at that span fraction. The two
    arrays are then combined into the form they need to be in for reading in twist
    information based on span fraction for MachUpX. For example:
        deflections = {"flaps1" : np.array([[0.0, 0.0],
                                            [0.2, x[0]],
                                            [0.4, x[1]],
                                            [0.6, x[2]],
                                            [0.8, x[3]],
                                            [1.0, x[4]]])}

    Parameters
    ----------
    a : list or array, Span Fractions
    b : list or array, twist at the span fraction

    Returns
    -------
    output_array : 2D array for the twist distribution over the span of the wing that can
    be used in MachUpX

    '''
    length_a = len(a)
    length_b = len(b)

    if length_a != length_b:
        return
    else:
        output_array = np.full((length_a,2),0.0)
        for x in range(0,length_a):
            output_array[x][0] = a[x]
            output_array[x][1] = b[x]

        return output_array


def double_repeat_and_join(a, b):
    '''This function combines the double_and_repeat function and the join function.
    The user passes the span fractions in as A and the x array from scipy.optimize.minimize
    in as B. B is doubled and repeated and then joined with A (span_fractions)
    to get the twist distribution array needed for MachUpX.
```

```

    Parameters
    ----------
    a : list or array, the span fraction locations for flaps.
    b : list or array, the x array from scipy.optimize.minimize (the twist values
                                                                for each span_frac)

    Returns
    -------
    array, distribution array containing the span fractions and associated twist for
    rectangluar flaps.

    '''
    b = double_and_repeat(b)
    return join(a, b)


def span_frac_to_cos_cluster(doubled_span_frac):
    '''This function takes the span fraction list and pulls out only the distinct
    span fraction locations, minus 0 and 1. This is needed so that the span fraction
    locations can be passed to the cosine clustering in MachUp X.

    Example of logic for removing doubled values:
        array Index     0   1   2   3   4   5
        arrayn value    x   x   y   y   z   z
        iteration
            1           0   1   2   3   4
                        x   y   y   z   z

            2           0   1   2   3
                        x   y   z   z

            3           0   1   2
                        x   y   z

    Parameters
    ----------
    doubled_span_frac : array, the span fraction list with doubles

    Returns
    -------
    doubled_span_frac : array, the span fraction list without doubles, 0, or 1

    '''
    del doubled_span_frac[0] #Remove the first 0.0
    del doubled_span_frac[0] #Remove the second 0.0
    del doubled_span_frac[-1] #Remove the 1.0 at the end of the list

    #remove the doubled values
    last_delete = int(len(doubled_span_frac)/2) + 1
    for x in range(1, last_delete):
        del doubled_span_frac[x]

    return doubled_span_frac
```

B.6 Supporting Code: airfoil_functional_creation.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 25 13:50:03 2022

@author: justice
"""
from Ikhana_main_wing_functions import *

'''
This code is used by to get to create a dictionary used by MachUpX that uses functions
to get CL, CD, and Cm. Passing in functions cannot be set up in the aircraft json before run time
so it is necessary to read in the aircraft json (making it a dictionary) and then replace
the airfoils section of the dictionary with output of this function, which now has the
CL, CD, Cm functions inside of the dictionary.
'''

def create_Ikhana_airfoils_function_dict():
    return {
        "Ikhana_NACA_0010_main": {
                    "type" : "functional",
            "CL" : get_Ikhana_CL,
            "CD" : get_Ikhana_CD,
            "Cm" : get_Ikhana_Cm,
                    "geometry" : {
                            "outline_points" : "AirfoilDatabase/airfoils/uCRM-9_wr0_xfoil.txt"
                            }
            },
            "Ikhana_NACA_0010" : {
            "type": "linear",
            "aL0": 0.0,
            "CLa": 6.43365,
            "CmLo": 0.0,
            "Cma": 0.0,
            "CD0": 0.00513,
            "CD1": 0.0,
            "CD2": 0.00984
            }
    }
```

B.7 Supporting Code: Ikhana_main_wing_functions.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 25 14:21:05 2022

@author: justice
"""
from math import pi
'''
These are the functions that are used by MachUpX to get CL, CD, and Cm whenever they
are needed for calculations. The data used to get the coefficients comes from
Hunsaker and Phillips
"Aerodynamic Shape Optimization of Morphing Wings at Multiple Flight Conditions"
AIAA SciTech Forum
9-13 January 2017, Grapevine Texas
55th AIAA Aerospace Sciences Meeting
'''

def get_Ikhana_CL(**kws):
    c1 = kws.get("trailing_flap_deflection", 0)     # radians  0 is a default value in this syntax
    alpha = kws.get("alpha", 0)                     # radians
    c1_deg = c1 * (180/pi)                          # degrees (treating as camber)

    aL0 = get_alpha_L0(c1_deg)                      # radians
    CLa = get_CL_alpha(c1_deg)                      # 1/radians

    return CLa*(alpha-aL0)                          # unitless coefficient

def get_Ikhana_CD(**kws):
    c1 = kws.get("trailing_flap_deflection", 0)     # radians
    CL = get_Ikhana_CL()                            # unitless coefficient
    c1_deg = c1*(180/pi)                            # degrees (treating as camber)

    CD0 = get_CD0(c1_deg)                           # unitless coefficient
    CD1 = get_CD1(c1_deg)                           # unitless coefficient
    CD2 = get_CD2(c1_deg)                           # unitless coefficient

    return (CD0 + CD1*CL + CD2*CL*CL)               # unitless coefficient

def get_Ikhana_Cm(**kws):
    c1 = kws.get("trailing_flap_deflection", 0)     # radians
    alpha = kws.get("alpha", 0)                     # radians
    c1_deg = c1*(180/pi)                            # degrees (treating as camber)

    CmL0 = get_Cm_L0(c1_deg)                        # unitless coefficient
    Cma = get_Cm_alpha(c1_deg)                      # 1/radians
    aL0 = get_alpha_L0(c1_deg)                      # radians

    return CmL0 + Cma*(alpha-aL0)                   # unitless coefficient

##############################################################################
# The following formulas are based off of camber as a percentage of the chord.
def get_alpha_L0(c):
```

```
    '''
    This is a linear fit for the data generated by Hunsaker and Phillips in
    "Aerodynamic Shape Optimization of Morphing Wings at Multiple Flight Conditions"
    AIAA SciTech Forum
    9-13 January 2017, Grapevine Texas
    55th AIAA Aerospace Sciences Meeting

    Parameters
    ----------
    c : float
        camber as percentage of the chord.

    Returns
    -------
    float
        Value of alpha L0 in Radians.

    '''
    return -0.0183*c - 0.0003                       # radians

def get_CL_alpha(c):
    '''
    This is an average from the data generated by Hunsaker and Phillips in
    "Aerodynamic Shape Optimization of Morphing Wings at Multiple Flight Conditions"
    AIAA SciTech Forum
    9-13 January 2017, Grapevine Texas
    55th AIAA Aerospace Sciences Meeting


    Parameters
    ----------
    c : float
        camber as percentage of the chord.

    Returns
    -------
    float
        CL_alpha in (1/rad).
    '''
    return 6.257605                                 # 1/radians

def get_CD0(c):
    '''
    This is a parabolic fit for the data generated by Hunsaker and Phillips in
    "Aerodynamic Shape Optimization of Morphing Wings at Multiple Flight Conditions"
    AIAA SciTech Forum
    9-13 January 2017, Grapevine Texas
    55th AIAA Aerospace Sciences Meeting

    Parameters
    ----------
    c : float
        camber as percentage of the chord.

    Returns
```

```
    -------
    float
        The unitless value for the coefficient CD0.

    '''
    return 0.0002*(c**2) - (4e-5)*c + 0.0049        # unitless coefficient

def get_CD1(c):
    '''
    This is a linear fit for the data generated by Hunsaker and Phillips in
    "Aerodynamic Shape Optimization of Morphing Wings at Multiple Flight Conditions"
    AIAA SciTech Forum
    9-13 January 2017, Grapevine Texas
    55th AIAA Aerospace Sciences Meeting

    Parameters
    ----------
    c : float
        camber as percentage of the chord.

    Returns
    -------
    float
        The unitless value for the coefficient CD1.

    '''
    return -0.003*c + 0.0002                        # unitless coefficient

def get_CD2(c):
    '''
    This is a parabolic fit for the data generated by Hunsaker and Phillips in
    "Aerodynamic Shape Optimization of Morphing Wings at Multiple Flight Conditions"
    AIAA SciTech Forum
    9-13 January 2017, Grapevine Texas
    55th AIAA Aerospace Sciences Meeting

    Parameters
    ----------
    c : float
        camber as percentage of the chord.

    Returns
    -------
    float
        The unitless value for the coefficient CD2.

    '''
    return 0.0001*(c**2) - 0.0004*c + 0.0095        # unitless coefficient

def get_Cm_L0(c):
    '''
    This is a linear fit for the data generated by Hunsaker and Phillips in
    "Aerodynamic Shape Optimization of Morphing Wings at Multiple Flight Conditions"
    AIAA SciTech Forum
    9-13 January 2017, Grapevine Texas
```

```
    55th AIAA Aerospace Sciences Meeting

    Parameters
    ----------
    c : float
        camber as percentage of the chord.

    Returns
    -------
    float
        The unitless value for the coefficient Cm_L0.

    '''
    return -0.0253*c - 0.0004                       # unitless coefficient

def get_Cm_alpha(c):
    '''
    This is an average from the data generated by Hunsaker and Phillips in
    "Aerodynamic Shape Optimization of Morphing Wings at Multiple Flight Conditions"
    AIAA SciTech Forum
    9-13 January 2017, Grapevine Texas
    55th AIAA Aerospace Sciences Meeting


    Parameters
    ----------
    c : float
        camber as percentage of the chord.

    Returns
    -------
    float
        CM_alpha in (1/rad).
    '''
    return 0.016353333                              # 1/radians
```

B.8 Supporting Code: Ikhana_cosine_clustering.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jan  5 15:42:29 2022

@author: justice
"""

from Ikhana_join import *

def create_cos_cluster_array(num_control_points, print_results = False):
    '''
    This function can be used to get a cosine clustering array for any given
    number of control points for the NASA Ikhana. This code is used in the main
    optimization code (Ikhana_update_twist_optimization_conditional_functional.py)
    to set the cluster points used for the grid for the Ikhana. This function is needed
    because the cluster points are set dynamically during run time of the optimization,
    meaning any number of control points can be passed in and this function
    will be used to get the appropriate cluster points.

    Parameters
    ----------
    num_control_points : float
        The number of control points to use for the NASA Ikhana.
    print_results : boolean, optional
        Whether to print out the cluster points, useful for debugging and informational purposes. The default is
            False.

    Returns
    -------
    cos_cluster : array, [float]
        The span fraction array without doubles, 0, or 1

    '''
    span_frac = create_span_fraction_array(num_control_points)
    cos_clust = span_frac_to_cos_cluster(span_frac)

    if print_results:
        print("Ikhana span locations for cos clustering (ie span locations for control points)")
        print(cos_clust)

    return cos_clust
```

B.9 Supporting Code: timing.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jan  5 10:00:21 2022

Pulled from stackoverflow on Wed Jan 5, 2022
https://stackoverflow.com/questions/1557571/how-do-i-get-time-of-a-python-programs-execution
answered Sep 10 '12 at 2:03 by Nicojo

Used to display the runtime for each of the run code files. It also gives a method
for getting the current time with secondsToStr(), which is used to differentiate
the multiple output files generated during each run.
"""

import atexit
from time import time, strftime, localtime
from datetime import timedelta

def secondsToStr(elapsed=None):
    if elapsed is None:
        return strftime("%Y-%m-%d %H:%M:%S", localtime())
    else:
        return str(timedelta(seconds=elapsed))

def log(s, elapsed=None):
    line = "="*40
    print(line)
    print(secondsToStr(), '-', s)
    if elapsed:
        print("Elapsed time:", elapsed)
    print(line)
    print()

def endlog():
    end = time()
    elapsed = end-start
    log("End Program", secondsToStr(elapsed))

start = time()
atexit.register(endlog)
log("Start Program")
```