5-2025

Investigating Stability of Cone-Derived Hypersonic Waverider Vehicles via Design Space Exploration

Adam S. Weaver
Utah State University

INVESTIGATING STABILITY OF CONE-DERIVED HYPERSONIC WAVERIDER VEHICLES VIA DESIGN SPACE EXPLORATION

by

Adam S. Weaver

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

of

MASTER OF SCIENCE

in

Aerospace Engineering

Approved:

Douglas F. Hunsaker, Ph.D. Major Professor	Som Dutta, Ph.D. Committee Member
Stephen A. Whitmore, Ph.D. Committee Member	Kevin G. Bowcutt, Ph.D. Committee Member

D. Richard Cutler, Ph.D.
Vice Provost for Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2025

ABSTRACT

Investigating Stability of Cone-Derived Hypersonic Waverider Vehicles via Design Space Exploration

by

Adam S. Weaver, Master of Science

Utah State University, 2025

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

Waverider vehicles exploit a layer of compressed air created by their own shock to enhance their aerodynamic efficiency at cruise conditions. Among the various types of waveriders, conically-derived geometries are common and have been subject to optimization routines targeting superior L/D ratios. However, conically-derived waveriders rely on stabilizing surfaces to achieve lateral and/or longitudinal stability at basic cruise conditions. To characterize stability, the purpose of this paper is to explore the effects of leading-edge parameters on both stability and aerodynamic efficiency within a preliminary design space. Cone-derived waverider bodies are generated by perturbing leading-edge polynomial parameters as dictated by a exhaustive search at any given user-specified cruise condition. Graphical relationships between leading-edge parameters and aerodynamic traits are curated by the use of a high-speed, low-fidelity impact-method solver. Visual relationships between discrete stability constraints, waverider geometry, aerodynamic characteristics, and leading edge parameters are developed.

(107 pages)

PUBLIC ABSTRACT

Investigating Stability of Cone-Derived Hypersonic Waverider Vehicles via Design Space Exploration

Adam S. Weaver

It may come as no surprise that man-made objects flying at 5, 10, or even 25 the speed of sound (termed *hypersonic*) are difficult to design, inefficient in flight, and expensive to deploy. Most hypersonic vehicles are effectuated in ballistics, which simplify aerodynamic considerations owing to their fixed trajectory. However, *maneuverable* hypersonic vehicles can revolutionize commercial flight, national defense, and space travel. One candidate for such missions are waverider vehicles. Waverider vehicles cruise on a cushion of air generated by its own shock wave, essentially enjoying extra lift without any extra drag penalties (which are usually inherent to extra lift).

Waveriders have been optimized by finding waverider shapes that maximize its lift-to-drag (L/D) ratio at a waverider's normal operating cruise condition. However, such bodies may be unstable in flight and thus, cannot be realized in the real world in their pure form. This thesis aims at investigating what happens when basic flight stability requirements are considered in tandem with L/D maximization. This research generates important correlations that can be helpful to engineers hoping to investigate waverider vehicles in a preliminary stage; thus, this work and its data may prove helpful to the actual realization of maneuverable hypersonic vehicles in our day.

"Higher, Orville, higher!"

ACKNOWLEDGMENTS

I would be amiss to fail to acknowledge my mother for waking me up at 3 AM as a child to watch the shooting stars, my father for his wonderful example in completing graduate education, and my brothers for extending a listening ear and supporting hand in my endeavors. I would like also like to thank my teachers, extended family, and friends for fostering my passions, sharing their knowledge, and leading by example. My dreams were created and made possible by those before me!

I would also like to thank the expertise, encouragement, and advice of my advisor, Dr. Douglas Hunsaker, who is a large reason why I chose to pursue this thesis. I would also like to thank my wife, Lindsey, for making sure I didn't forget the rest of the world.

All praise to my Almighty God for His hand in my life: helping me achieve my goals, giving me hope, and leaving challenges for His children to solve. Many times, I have been led by His great wisdom and taken to heights I could not imagine.

Adam S. Weaver

CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

NOTATIONS

γ	Specific Heat Ratio
β	Shock Angle
δ	Flow Deflection Angle
Δ	Change-in
$\Delta^{'}$	RK4 First 'Implicit' Step derived from Euler's method
$\Delta^{''}$	RK4 Second 'Implicit' Step
$\Delta^{'''}$	RK4 Third 'Implicit' Step
$\Delta^{''''}$	RK4 Fourth 'Implicit' Step
L/D	Lift-to-Drag Ratio
θ	Polar Angle (on zy-plane)
ϕ	Spherical coordinate angle (on zx-plane)
r	Spherical 'Length'
C_{l_β}	Yaw Stability Derivative
C_{n_β}	Roll Stability Derivative
C_{l_α}	Pitch Stability Derivative
V_r	Velocity Parallel to r vector
V_θ	Velocity Parallel to θ, the polar angle
$\dot{V}_r$	Positional Derivative equal to $\frac{dV_r}{d\theta}$
$\ddot{V}_r$	Positional Derivative equal to $\frac{d^2V_r}{d\theta^2}$
M_2	Mach Number directly behind oblique shock
$\mathbf{e}_r$	Unit Vector along r direction
$\mathbf{e}_\theta$	Unit Vector along θ direction
$\mathbf{e}_\phi$	Unit Vector along ϕ direction

ACRONYMS

RK4	Runge-Kutta 4^{th}-Order Numerical Integration Scheme
SHADOW	Stability of Hypersonic Aerodynamic Derivatives for Optimized Waveriders
CBAero	Configuration Based Aerodynamics (NASA Software Package)
.tri	Triangular Information File Format
COBYLA	Constrained Optimization by Linear Approximation
SLSQP	Sequential Least SQuares Programming Optimizer

CHAPTER 1

INTRODUCTION

Ulike subsonic and supersonic flow, which are characterized by their relativity to the local speed of sound, hypersonic flow is defined and dominated by immense aerothermal loads [4] [1], making hypersonic vehicles difficult and expensive to successfully implement. Beyond implementation challenges, geopolitics has played an important role in hypersonic vehicle adoption since its inception in 1949. One promising approach to execute high-speed missions expands on the traditional ballistic approach, enabling hypersonic vehicles to change their trajectory by maneuvering in the atmosphere, requiring new challenges for defense applications [5] and providing promise for hypersonic commercial flight. Albeit greater implementation challenges, vehicles capable of flying these missions, deemed *hypersonic glide vehicles* (HGVs) [6], provide a broad range of benefits for various applications. In this light, HGVs have cemented their position in a multinational arms race and are increasingly popular among research scientists studying high-speed aerodynamics. The intent of this research is to analyze waveriders retrofitted as hypersonic glide vehicles utilizing either a vertical take-off rocket-powered boost-glide configuration or horizontal take-off-and-landing (HTOL) air-breathing configuration.

Specialized hypersonic vehicles, *waveriders*, are uniquely poised to make atmospheric hypersonic travel more efficient and are excellent candidates for HGVs or re-entry vehicles, whether on Earth or on other planets [7]. Waveriders are crafted to perfectly confine a compression layer synthesized from its own oblique shock to the edges of its lower surfaces, preventing any spillage flow from high to low pressure zones. Hence, waveriders enjoy extra lift without inducing drag penalties. The first waverider geometry was developed by Dr. Terrence Nonweiler [8] via the postulation of planar shocks stemming from the leading edge of a delta wing planform, creating a caret wing re-entry vehicle that would capture high

[1] Heuristically, the term "hypersonic" is used for vehicles traveling greater than Mach 5, though this number is subject for debate

pressure air under its body. Since a waverider vehicle's induced shock is attached along its entire leading edge, waverider bodies are dictated by a shock structure at a selected cruise condition. In this light, waverider geometries differ from traditional aircraft because they are *discovered*, rather than just *designed*. Historical research on waverider geometries are well-summarized in Ref. [9].

Waverider geometries can be derived from various supersonic flow conditions, depending on the suitable application. Beyond the development of the first waveriders derived from planar shocks, waverider shapes can also be derived from non-axisymmetric planar flow stemming from star-shaped cross sections [10] or general three-dimensional non-axisymmetric flow [11] [12]. Additional flexibility is achieved by defining waveriders based on multi-shock definitions, like osculating methods [13] [14]. Osculating cone methods are common in waverider design; however, a more general osculating flow method may provide even greater aerodynamic performance [15]. In addition to investigating shock configurations discretely, looking at favorable body configurations and their consequential shock condition may also be beneficial. For example, power law bodies present slight increases in lift-to-drag compared to cone flow [16]. Further, dynamically shaping a waverider geometry could improve efficiency over an entire trajectory [17]. For this present work, a single cone flow for each cruise condition was used to generate waveriders because such flow fields present an excellent blend of fidelity and simplicity. Additionally, axisymmetric flows like cone flows possess particular advantages in volumetric efficiency [18].

After defining a shock(s) to generate a waverider, waverider lower surfaces are crafted by tracing post-shock streamlines from a leading edge geometry. *Streamlines* are instantaneous vector snapshots of a particular velocity field; each streamline is exactly tangent with the velocity vector at each point along a flow. In the case of cone-flow, the streamlines downstream of a cone-shaped shock structure bend towards the generating solid cone, becoming parallel to it, often giving rise to the uniquely beautiful lip-shaped waverider lower surface, as shown in Fig. (1.1). Tracing streamlines in an already known flowfield ensures the flow over the lower surface of a waverider will remain unaltered, cementing adherence to

assumed shock conditions and facilitating simple mathematical computation [19]. To this end, the lower surface of a cone-derived waverider is solely dictated by both the shock cone and leading edge projection.

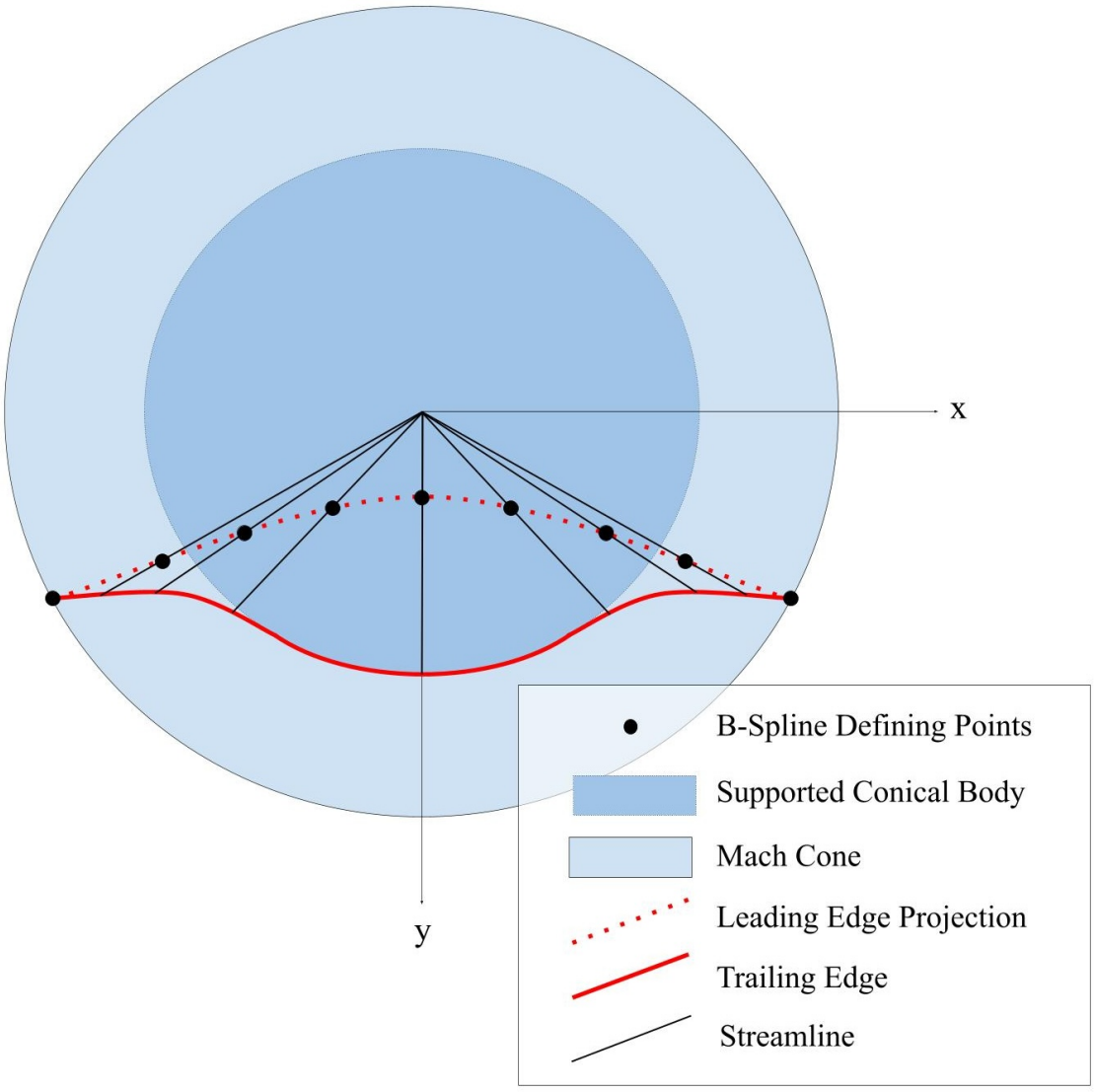


Fig. 1.1: Waverider Development, adapted from [1]

Modern computational power enables scientists to quickly approach the best of the possible waverider shapes via numerical minimization methods or exhaustive search techniques. Targeting waverider geometries that maximize L/D, for example, can produce shapes that

break Dr. Kuchemann's classical L/D 'barrier' [19] [20]. However, design space exploration is only useful within the domain in which it operates – a domain which sometimes produces undesirable or impossible results. For instance, many waverider shapes are laterally unstable in cruise and are longitudinally unstable along the entire speed regime [21], often requiring a deflected aft control surface to achieve longitudinal stability at cruise [22]. Although constraining a waverider geometry to meet realistic standards, like volumetric constraints, might decrease the final aerodynamic efficiency [23], integrating constraints in preliminary design space is better than altering the aircraft *after* an optimization phase [18].

Much of the existing body of research in waverider stability exists in the low-speed regime, where waveriders are inherently off-condition. For example, some papers have successfully characterized longitudinal stability and/or control authority at subsonic speeds [24] [25] [26]. However, some research on longitudinal stability exists at high speeds too – one paper, for example, explores double-swept geometries and their effect on high speed longitudinal stability [27]. Further, a parametric design study concerning a volumetric-constrained waverider-based missile provides additional insights on important design parameters [28]. Another study employed a pitch-stable constraint on various design parameters to optimize a caret-shaped waverider, proving that stability-constrained waveriders are possible [29]. Similar research, although more limited, exists on lateral stability at low [30] [31] and high speeds [32].

Much of the research on waverider stability is fundamentally experimental or does not provide concise, comprehensive insight on how to design waveriders with stability in mind (despite some notable exceptions, like found in Ref. [21]). The purpose of the present study is to enhance the existing body of knowledge by providing a comprehensive look at the waverider design space in its relation to static stability at cruise conditions using exhaustive search methods. Such methods provide concise visual relationships between basic waverider geometry parameters and aerodynamic performance, suggesting the existence of stable, near-optimal waveriders.

CHAPTER 2
THEORY

Modern compressible flow relations, aided by numerical methods, is paramount to understand and visualize supersonic conical flow. In this section, several fundamental theories pertinent to the completion of this research are presented and discussed.

2.1 Coordinate System

To visualize and develop axisymmetric conical flow, it is common to utilize the spherical coordinate system, as outlined in Fig. (2.1). In a spherical coordinate system, a flow field stemming from a right-circular cone at zero angle of attack guarantees that lines at constant ϕ are streamlines and that the flow is axisymmetric (congruent at any θ).

However, mesh file formatting requires waverider geometry points to be delineated in a traditional three-dimensional Cartesian coordinate system. Thus, the following coordinate transformations were used as needed

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} \\ \theta &= \cos^{-1}\left(\frac{z}{\sqrt{x^2 + y^2 + z^2}}\right) \\ \phi &= \tan^{-1}\left(\frac{y}{x}\right) \end{aligned} \tag{2.1}$$

After a waverider geometry is defined, its length, area, and center of gravity dimensional references are utilized to ensure the aerodynamic forces and moments are non-dimensionalized and reported correctly. It is common to define a waverider reference area as the planform area projected on the xz-plane and the reference length as the waverider length [22] [33]. Because the geometry development code strictly creates and analyzes *pure* waverider geometries, the center of gravity is ambiguous because the weight and positions of the propulsion systems, control surfaces, fuel, and internal structures are not known.

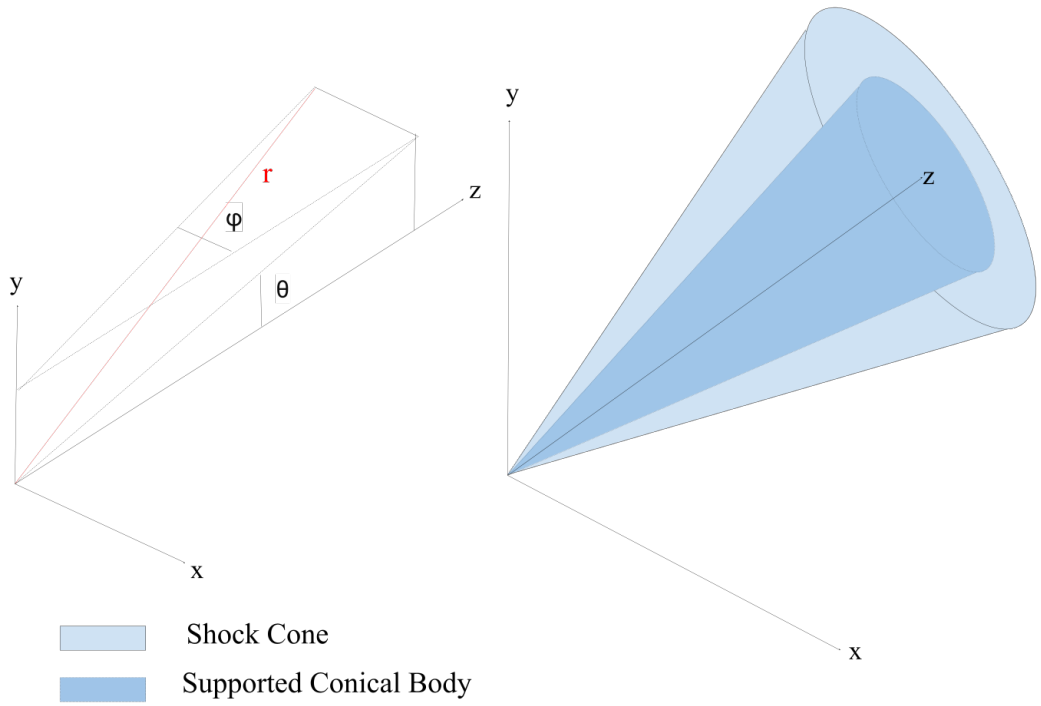


Fig. 2.1: Coordinate system of the conical flow field and associated waverider

Rather than designing and processing weight and mass statements for each vehicle, which is outside of the scope of this conceptual research, empirical data from previous supersonic aeroplanes – the North American XB-70, Lockheed Martin SR-71, and the NASA X-15. Not coincidentally, each of these vehicles reported a center of gravity at or near 0.25 of the mean aerodynamic chord (MAC) [34] [35] [36]. Assuming the waveriders generated in this study are lifting bodies, the entire body can be treated as a wing, simplifying the MAC integration. The mean aerodynamic chord is defined purely geometrically, as

$$\overline{c}_{mac} = \frac{2}{S}\int_{x=0}^{b/2} c^2 dx \tag{2.2}$$

where S is planform area projection, b is base lateral length, and c is chord length – a function of x. Using the lengths of discretely defined top-surface streamlines (described in detail later in this text) along the top surface, Eq. (2.2) was numerically integrated to obtain the mean aerodynamic chord. After determining the MAC, the point corresponding to center of gravity was defined by tracing 0.75 MAC upstream from the base surface (along the center line of the waverider).

2.2 Taylor-Maccoll Cone Flow

Unlike two-dimensional wedge flow, where flow properties are mathematically accessible, three-dimensional cone flow introduces additional freedom for the flow [37], preventing exact, closed form solutions in the present day. A common way to derive axisymmetric three-dimensional flow is the hypersonic small disturbance theory, which approximates the flow via a thickness parameter and mach number [38], combined in a similarity parameter which can relate flow properties [39]. Small disturbance theory approximates body pressures well [38], especially with slender bodies flying at high altitudes [40]. However, more deliberate cone flow solutions can be solved by the Taylor-Maccoll equation [41], which formulates compressible, conical flow via differential equations. Hence, the present study utilizes modern numerical analysis to evaluate the Taylor-Maccoll equation.

2.2.1 Taylor-Maccoll Equation Derivation

To begin, it is important to understand that the spherical coordinates can be simplified by the axisymmetric flow assumption that ∇V_ϕ does not change the flow. Accordingly, cone flow is consistent at a given theta at *any* ϕ, as given by

$$\begin{aligned} \frac{\partial}{\partial \phi} &= 0 \\ \frac{\partial}{\partial r} &= 0 \end{aligned} \tag{2.3}$$

The Taylor-Maccoll equation is derived from the continuity equation with regards to spherical coordinates, as described by

$$\nabla \cdot \rho(\mathbb{V}) = \frac{1}{r^2}\frac{\partial}{\partial r}(r^2 \rho V_r) + \frac{1}{r\sin\theta}\frac{\partial}{\partial \theta}(\rho V_\theta \sin\theta) + \frac{1}{r\sin\theta}\frac{\partial(\rho V_\phi)}{\partial \phi} = 0 \tag{2.4}$$

After applying the product rule to evaluate the derivatives, the spherically derived continuity equation becomes

$$\frac{1}{r^2}\left[r^2\frac{\partial(\rho V_r)}{\partial r} + \rho V_r(2r)\right] + \frac{1}{r\sin\theta}\left[\rho V_\theta \cos\theta + \sin\theta\frac{\partial(\rho V_\theta)}{\partial \theta}\right] + \frac{1}{r\sin\theta}\frac{\partial(\rho V_\theta)}{d\phi} = 0 \tag{2.5}$$

With only one remaining ∂ operator remaining, it is suitable to evaluate the partial derivative. Further, axisymmetric flow conditions are applied using Eq. (2.3)

$$\frac{2\rho V_r}{r} + \frac{\rho V_\theta}{r}\cot\theta + \frac{1}{r}\left(\rho\frac{\partial V_\theta}{\partial \theta} + V_\theta\frac{\partial \rho}{\partial \theta}\right) = 0 \tag{2.6}$$

Then, the equation can be simplified by multiplying by r

$$2\rho V_r + \rho V_\theta \cot\theta + \rho\frac{\partial V_\theta}{\partial \theta} + V_\theta\frac{\partial \rho}{\partial \theta} = 0 \tag{2.7}$$

Crocco's theorem [42] relates enthalpy and entropy to pressure, vorticity, and velocity as given by

$$T\nabla s = \nabla h_0 - \nabla V \times (\nabla \times \mathbb{V}) \tag{2.8}$$

Since the edge of a conical shock is straight, the change in entropy across the flow field is constant ($\Delta s = 0$). Further, we can assume that the flow is adiabatic and steady, meaning that there is no enthalpy change across the flow ($\Delta h_0 = 0$). Utilizing these assumptions, Crocco's theorem is simplified, as expressed by

$$\nabla \times \mathbb{V} = 0 \tag{2.9}$$

An arbitrary coefficient, $\frac{1}{r^2 \sin\theta}$, is utilized to aid with future simplification. Additionally, the cross product can be expanded to yield a matrix

$$\nabla \times \mathbb{V} = \frac{1}{r^2 \sin\theta} \begin{vmatrix} \mathbf{e}_r & r\mathbf{e}_\theta & (r\sin\theta)\mathbf{e}_\phi \\ \frac{\partial}{\partial r} & \frac{\partial}{\partial \theta} & \frac{\partial}{\phi} \\ V_r & rV_\theta & (r\sin\theta)V_\phi \end{vmatrix} = 0 \tag{2.10}$$

Evaluating the cross product by Surrus's rule yields an expanded equation

$$\begin{aligned} \nabla \times \mathbb{V} = \frac{1}{r^2 \sin\theta} \Bigg\{ & \mathbf{e}_r \left[\frac{\partial}{\partial \theta}(rV_\phi \sin\theta) - \frac{\partial}{\partial \phi}(rV_\theta) \right] \\ & - r\mathbf{e}_\theta \left[\frac{\partial}{\partial r}(rV_\phi \sin\theta) - \frac{\partial}{\partial \phi}(V_r) \right] \\ & + (r\sin\theta)\mathbf{e}_\phi \left[\frac{\partial}{\partial r}(rV_\theta) - \frac{\partial V_r}{\partial \theta} \right] \Bigg\} \\ = 0 \end{aligned} \tag{2.11}$$

Since each component of this curl operation must be zero, a helpful relation can be developed by looking specifically at the component tied to the $\mathbf{e}_\phi$ unit vector

$$\frac{1}{r^2 \sin\theta}(r\sin\theta)\mathbf{e}_\phi \left[\frac{\partial}{\partial r}(rV_\theta) - \frac{\partial V_r}{\partial \theta} \right] = 0 \tag{2.12}$$

Evaluating the ∂ operator using the product rule, the equation is better represented by

$$\frac{1}{r^2 \sin\theta}(r\sin\theta)\mathbf{e}_\phi [r\frac{\partial V_\theta}{\partial r} + V_\theta \frac{\partial r}{\partial r} - \frac{\partial V_r}{\partial \theta}] = 0 \tag{2.13}$$

Using the equations for steady, axisymmetric cone flow found in Eq. (2.3), Eq. (2.13) is reduced to a simple, powerful relation

$$V_\theta = \frac{\partial V_r}{\partial \theta} \tag{2.14}$$

Euler's equation is the differential equation that captures momentum conservation. In vector form, Euler's equation can be written as follows

$$\rho \frac{D\mathbf{V}}{Dt} = -\nabla p + \rho \mathbf{f} \tag{2.15}$$

Since the flow is time independent and irrotational, Euler's equation is simplified, expressed as

$$-dp = \frac{1}{2}\rho V d(V)^2 \tag{2.16}$$

Or alternatively, the equation can be simplified further

$$dp = -\rho V dV \tag{2.17}$$

However, since $V^2 = V_r^2 + V_\theta^2$, Euler's equation can be represented in polar coordinates as follows

$$dp = -\rho(V_r dV_r + V_\theta dV_\theta) \tag{2.18}$$

To gain further understanding of this equation, isentropic flow relations can be explored

$$\frac{dp}{d\rho} = \left(\frac{\partial p}{\partial \rho}\right)_s = a^2 \tag{2.19}$$

Note that Eq. (2.19) represents Mach number as a, and constant entropy as the subscript s. Combining Eq. (2.19) and Eq. (2.18) yields an important relation, shown by

$$\frac{dp}{\rho} = -\frac{1}{a^2}(V_r dV_r + V_\theta dV_\theta) \tag{2.20}$$

It is helpful to represent enthalpy in terms of velocity. To do this, it is common to set V_{max} as the velocity at which the flow has expanded until it essentially is at zero temperature

$$V_{max} = \sqrt{2h_0} \tag{2.21}$$

For a calorically perfect gas, the specific enthalpy becomes

$$\begin{aligned} \frac{a^2}{\gamma-1}+\frac{V^2}{2} &= \frac{V_{max}^2}{2} \\ a^2 &= \frac{\gamma-1}{2}(V_{max}^2-V^2) \\ &\frac{\gamma-1}{2}(V_{max}^2-V_r^2-V_\theta^2) \end{aligned} \tag{2.22}$$

Subbing Eq. (2.22) into Eq. (2.20)

$$\frac{d\rho}{\rho} = -\frac{2}{\gamma-1}\left(\frac{V_r dV_r + V_\theta dV_\theta}{V_{max}^2 - V_r^2 - V_\theta^2}\right) \tag{2.23}$$

Eq. (2.7) can be rearranged by dividing by ρ

$$2V_r + V_\theta \cot\theta + \frac{dV_\theta}{d\theta} + \frac{V_\theta}{\rho}\frac{d\rho}{d\theta} = 0 \tag{2.24}$$

Further, Eq. (2.23) can be divided by $d\theta$ to get the following differential equation

$$-\frac{2\rho}{\gamma-1}\left(\frac{V_r\frac{dV_r}{d\theta} + V_\theta\frac{dV_\theta}{d\theta}}{V_{max}^2 - V_r^2 - V_\theta^2}\right) \tag{2.25}$$

Combining Eq. (2.25) into Eq. (2.24), the equation can be represented as a differential equation set equal to zero

$$\frac{\gamma-1}{2}(V_{max}^2 - V_r^2 - V_\theta^2)\left(2V_r + V_\theta\cot\theta + \frac{dV_\theta}{d\theta}\right) - V_\theta\left(V_r\frac{dV_r}{d\theta} + V_\theta\frac{dV_\theta}{d\theta}\right) \tag{2.26}$$

Using the relation in Eq. (2.14) and its derivative, the Taylor-Maccoll equation is successfully derived and represented in the following equation

$$\frac{\gamma-1}{2}\left[V_{max}^2 - V_r^2 - \left(\frac{dV_r}{d\theta}\right)^2\right]\left[2V_r + \frac{dV_r}{d\theta}\cot\theta + \frac{d^2V_r}{d\theta^2}\right] - \frac{dV_r}{d\theta}\left[V_r\frac{dV_r}{d\theta} + \frac{dV_r}{d\theta}\left(\frac{d^2V_r}{d\theta^2}\right)\right] = 0 \tag{2.27}$$

Representing the equation's spatial derivatives with a superscript dot, the Taylor-Maccoll equation is represented as

$$\frac{\gamma-1}{2}[1-V_r^2-(\dot{V}_r)^2][2V_r+\dot{V}_r\cot\theta+\ddot{V}_r]-\dot{V}_r[V_r\dot{V}_r+\dot{V}_r\ddot{V}_r]=0 \tag{2.28}$$

The Taylor-Maccoll equation is impossible to solve analytically, necessitating the use of numerical integration techniques. To solve the Taylor-Maccoll equation numerically, it is common to split the equation into a system of equations by first setting $\dot{V}_\theta = \ddot{V}_r$ (as derived above). To solve for the set of differential equations, an arbitrary constant can be used

$$A=\frac{\gamma-1}{2}[1-V_r^2-(\dot{V}_r)^2] \tag{2.29}$$

Algebraically, $\dot{V}_\theta$ can be isolated by setting it equal to a fraction split by terms added or subtracted to $\dot{V}_\theta$ in the numerator, divided by the terms multiplied by $\dot{V}_\theta$

$$\dot{V}_\theta=-\frac{[A(2V_r+\dot{V}_r\cot(\theta))-\dot{V}_r(V_r\dot{V}_r)]}{A-\dot{V}_r^2} \tag{2.30}$$

The resulting system of differential equations is

$$\begin{bmatrix}\dot{V}_r\\ \dot{V}_\theta\end{bmatrix}=\begin{bmatrix}V_\theta\\ -\frac{[A(2V_r+\dot{V}_r\cot(\theta))-\dot{V}_r(V_r\dot{V}_r)]}{A-\dot{V}_r^2}\end{bmatrix} \tag{2.31}$$

Using shock relationships informed by freestream conditions, like mach number directly upstream of the shock (M_1), flow conditions directly behind the cone shock can be determined. Because of the pressure increase found downstream of any shock structure, flow bends towards the cone axis as soon as it enters the conical shock and is measured by the flow deflection angle δ. Flow deflection angle and other properties, including the mach number directly behind the shock (M_2), can be obtained by oblique shock relations [37].

2.2.2 Computational Scheme

The numerical procedure to solve the Taylor-Maccoll equation is aided by the use of a non-dimensional velocity, given by

$$\frac{V}{V_{max}} = V^{'} = \left[\frac{2}{(\gamma-1)M^2} + 1\right]^{-\frac{1}{2}} \tag{2.32}$$

where M is the mach number directly behind the shock (M_2). Further, the total velocity can be trigonometrically sectionalized into directional components along the r and θ axis, represented by V_r and V_θ

$$\begin{aligned} V_r &= V^{'}\cos(\theta-\delta) \\ V_\theta &= V^{'}\sin(\theta-\delta) \end{aligned} \tag{2.33}$$

These directional velocity components are important to both initialize and solve the Taylor Maccoll differential equations found in Eq. (2.31) via numerical integration. Explicit methods like the fourth-order Runge-Kutta (RK4) technique [43] are widely used for numerical integration due to their versatility and simplicity. The RK4 method is outlined as follows

$$\Delta y = \frac{\Delta^{'} + 2\Delta^{''} + 2\Delta^{'''} + \Delta^{''''}}{6} \tag{2.34}$$

where

$$\begin{aligned} \Delta^{'} &= f(x,y)\Delta x \\ \Delta^{''} &= f\left(x+\frac{\Delta x}{2}, y+\frac{\Delta^{'}}{2}\right)\Delta x \\ \Delta^{'''} &= f\left(x+\frac{\Delta x}{2}, y+\frac{\Delta^{''}}{2}\right)\Delta x \\ \Delta^{''''} &= f\left(x+\Delta x, y+\Delta^{'''}\right)\Delta x \end{aligned} \tag{2.35}$$

and

$$y_{i+1} = y_i + \Delta y \tag{2.36}$$

The vector field representing the flow downstream of the conical shock can be generated by selecting a theoretical conical body angle and integrating *out* to find the angle of the oblique shock generated by such body. Likewise, a shock angle can be selected and the theoretical conical body which supports the selected shock angle is discovered by integrating *in*. Choosing the later for the present study necessitating flipped the signs in the RK4 integration routine

$$\Delta y = \frac{\Delta^{'} - 2\Delta^{''} - 2\Delta^{'''} - \Delta^{''''}}{6} \tag{2.37}$$

where

$$y_{i+1} = y_i - \Delta y \tag{2.38}$$

2.3 Waverider Geometry Definition

It is important to note that the nature of the supersonic flow regime is represented by a *hyperbolic* partial differential equation (PDE), rather than an *elliptical* PDE used to represent subsonic flow. Rather than perturbing disturbances in all directions, disturbances in flow traveling $M_\infty > 1$ propagate only downstream. This discontinuity causes the flow's derivative to become undefined at sonic flow conditions; hence, Mach lines serve as characteristic lines of constant velocity [44] in supersonic flow. Projecting the leading edge on the defining shock cone and subsequent streamline tracing ensures that the mach lines are consistent to the shock cone from which the waverider was defined, guaranteeing the effectuated waverider to create the exact flow in which it was defined from, or as put by Dr. John D. Anderson –

> ..nature will make certain that the shock wave is attached all along the vehicle's leading edge, that is, the vehicle will be a waverider [37]

Confirmation of this theoretical background has been confirmed by both computational fluid dynamics (CFD) and schlieren imaging [45].

2.4 Optimization

Optimization routines iterate the design space until a minima is found, informed by gradient information or other mathematical estimators. Although more exhaustive and informed searches can increase the chances of converging to a global minima, minimization routines are not guaranteed, or likely, to uncover a global minima. Increasing L/D is a significant challenge along high-speed applications; thus, investigating L/D is often paramount to optimization routines along the hypersonic regime [20]. It should be noted that maximum L/D vehicles are tailored for cruise, while minimizing drag produces waveriders with the best acceleration properties [16]. Since the present research is concerning stability at cruise, the present study optimizes for L/D by finding the minimum of the negated L/D ratio.

Optimization routines are defined by functions returning a value whose lowest value represents the best possible solution. The framework inside of a optimization routine is often referred to as a *blackbox*; the numerical minimization method has no information about what defines the design space and is solely informed by input and output values associated with the function. Thus, the defining function and any sub-functions, including their analysis, must be curated deliberately to prevent incorrect or irrelevant results. Outside of the blackbox, various numerical routines exist to find minima of the given function. These, and other, components of optimization can have substantial effects on the success of the optimization routine and its eventual output. In this light, three main elements are predominant in a constrained optimization routine:

1. Minimization Method
2. Degrees of Freedom
3. Constraints

Mathematical theory is important to consider in optimization routines. For example, gradient-based optimization enables a more informed step-wise process, leading to faster convergence time and reduction in memory pressure. However, gradient-based methods are especially prone to converge on locally optimal solutions [46] and usually increase computational needs. Thus, gradient-based methods require the initial guess to be close to an anticipated 'global' optimal solution to produce valid results. Instead of making best-guess attempts, which tarnish the purpose and effectiveness of an optimization routine, an initial search algorithm [47] is often used. Derivative-free (non gradient-based) optimization routines are widely used and applicable in many different circumstances. Derivative-free optimization routines may require more iterations but are versatile for various applications.

Properly constraining an optimizer is important for the results' fidelity and applicability; unconstrained optimizers will produce undesirable and often unachievable results. Constraints limit certain variables as dictated by a user-defined constraint function. As opposed to constraints, *bounds* do not exhibit integrated mathematical iteration-dependencies. Instead, bounds limit optimization solutions by specifically limiting the pool of possible values in the solution vector for a given routine. Usually lacking mathematically functions to determine, bounds are usually more elementary in their implementation and are specified using a $(\min, \max)$ numerical pair. Bounds are especially helpful to discourage optimization routines from exploring invalid regions with a design space, though are not entirely preventative in order to estimate a function's gradient.

Exhaustive-search methods, although not guaranteeing a global minima, ensure a local minima within the selected domain. Unlike traditional optimization routines, exhaustive-search methods provide unique information about a selected design space, providing additional insight on parameterization, like sensitivities, coupling, and domain variable importance.

CHAPTER 3

OBJECTIVES

The aim of this research is to explore trade-offs between aerodynamic efficiency and stability adherence for cone-derived waverider vehicles at cruise conditions. This research goal can be accomplished via geometry development, implementing aerodynamic prediction models, and design space investigation. More specifically, these main tasks can be split into orderly, compounding project objectives:

1. Characterize cone-derived waverider geometries using leading-edge polynomial parameterization at various cruise conditions, unveiling near-optimal geometries
2. Assess the effects of pitch, yaw, and roll stability constraints on waverider geometry
3. Correlate leading-edge polynomial parameterization parameters to near-optimal waveriders, with and without stability considerations

CHAPTER 4

METHODS

This chapter describes the application of mathematical and physical theory to accomplish the plan set forth in the Objectives section.

The figures shown throughout this section are all based on the same default waverider geometry, which is discussed in detail in the Results section. The geometry is shown in the Fig. (4.1).

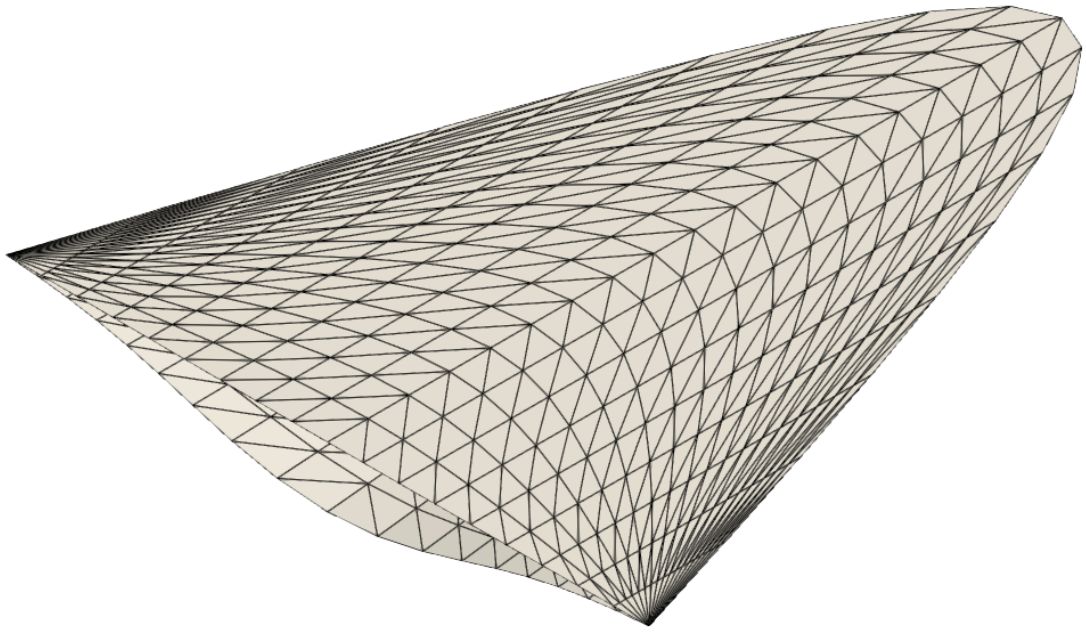


Fig. 4.1: Default Geometry

4.1 Waverider Design and Analysis

The overarching design space exploration is accomplished by the execution of three main scripts: geometry development, aerodynamic prediction deployment, and exhaustive search. These scripts work in tandem to create, analyze, or perturb waverider geometries. For an exhaustive-search design space exploration, optimization is not required. Rather, a test function which catalogs information about each waverider is deployed.

4.1.1 Geometry Development

Python boasts widely supported functionality in addition to being easy to write and debug – the ideal candidate for preliminary code development. Python's large developer community provides ready-to-implement packages like NumPy, SciPy, and MatPlotLib, which were used extensively to develop the waverider geometry code, which was named SHADOW (Stability of Hypersonic Aerodynamic Derivatives Of Waveriders). Further rewrites of the SHADOW code in compiled language like C++ or FORTRAN, which would implement numerical routines significantly faster, presents a potential efficiency boost in the future.

The geometry development portion of the code includes leading edge projection, Taylor-Maccoll flow development, streamline tracing, and mesh construction, as described below and shown in Fig.(4.2).

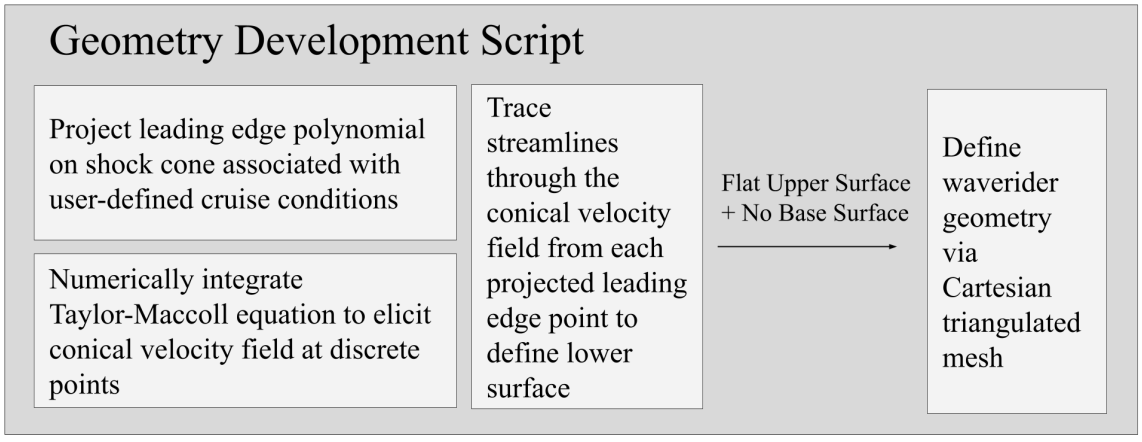


Fig. 4.2: Geometry Development

Leading Edge Projection

Projecting any representative two-dimensional leading edge trace on a three-dimensional shock structure ensures that the resulting three-dimensional leading edge can engender a waverider geometry. By perturbing the two-dimensional leading edge shape, uniquely defined families of waverider geometries can be created at select possible flow conditions. The leading edge is parameterized via simple n^{th} degree polynomials, which also ensure sufficiently smooth leading edges. The three-dimensional shock structure selected for this research is conical, which presents an excellent blend of fidelity and simplicity despite lacking potential for supporting lower-drag body configurations. It should also be noted that axisymmetric flows like cone flows possess particular advantages in volumetric efficiency [18], simplifying the later effectuation of such waverider vehicles.

Leading edge polynomials are defined solely by their shape in Quadrant IV and completed by their reflection across the y-axis. This approach avoids requiring higher order polynomial definitions to approximate vertically symmetric functions. Notwithstanding, any polynomial function must exhibit a smooth reflection across the y-axis in order to avoid a ridge in the waverider's top surface. A smooth reflection across the y-axis requires that the change in slope across the y-axis is zero, mathematically represented as

$$y^{'}(0) = 0 \tag{4.1}$$

Starting with a general second degree polynomial

$$y = Ax^2 + Bx + C \tag{4.2}$$

whose derivative is

$$y^{'} = 2Ax + B \tag{4.3}$$

The boundary condition can be plugged into the derivative, yielding

$$\begin{aligned} y^{'}(0) &= 2A(0) + B = 0 \\ \therefore B &= 0 \end{aligned} \tag{4.4}$$

Repeating this process for any n^{th}-order polynomial will find similar results; the coefficient in front of the linear x term must be zero.

After a leading edge polynomial is curated, the polynomial can be projected on the shock cone of interest by finding the corresponding z coordinate for each set of defining (x, y) leading edge pairs dictated by the defining polynomial. This process is aided by the use of spherical/Cartesian coordinate transformations; knowing the definition for the shock cone angle (θ) in standard spherical coordinates,

$$\theta = \cos^{-1}\left(\frac{z}{\sqrt{x^2 + y^2 + z^2}}\right) \tag{4.5}$$

This equation can be used to isolate z by rearranging to yield

$$\cos\theta = \frac{z^2}{x^2 + y^2 + z^2} \tag{4.6}$$

$$\cos^2\theta(x^2 + y^2 + z^2) = z^2 \tag{4.7}$$

$$z^2(1 - \cos^2\theta) = \cos^2\theta(x^2 + y^2) \tag{4.8}$$

$$z = \sqrt{\frac{\cos^2\theta(x^2 + y^2)}{(1 - \cos^2\theta)}} \tag{4.9}$$

$$\tag{4.10}$$

Each waverider is constrained to a non-dimensional length of 1 from the geometrical intersection of the zy-plane on the waverider body (z_i), enabling consistent scaling to more realistic values for the aerodynamic prediction model. Using Eq. (4.9) to solve for (z_i), it is natural to also derive the coordinate for x using θ and the z coordinate lying one-unit downstream of z_i, which will serve as the bounding end point –

$$z_{\text{end}} = z_i + 1 \tag{4.11}$$

Starting with

$$z^2(1 - \cos^2\theta) = \cos^2\theta(x^2 + y^2) \tag{4.12}$$

The corresponding x coordinate can be obtained by

$$\begin{aligned} \frac{z^2(1 - \cos^2\theta)}{\cos^2\theta} - y^2 &= x^2 \\ x &= \sqrt{\frac{z^2(1 - \cos^2}{\cos^2\theta} - y^2} \end{aligned} \tag{4.13}$$

where z is z_{end} and y is the leading edge polynomial function. Thus, Eq. (4.13) is a multi-dimensional polynomial with various possible "solutions". Geometrically, it is intuitive that the desired solution is the smallest x-intercept of Eq. (4.13), denoted as x_{end}.

Using the newly discovered x_{end}, the distance between the zy-plane and x_{end} becomes a reference length at which to scale the resolution of the leading edge in the x direction, which is controlled by n (number of leading edge points) by the user. Using the resolution information, an equally-spaced x vector neatly defines the leading edge geometry, which form two dimensional (x, y) pairs after employing the defining polynomial function. Eq. (4.9) can be used to project the rest of the leading edge point pairs on the shock cone.

Taylor-Maccoll Flow Field

The Taylor-Maccoll equation is a second-order non-linear differential equation, as developed in Eq. (2.3-2.27), which represents a solution to supersonic flow around a non-inclined right cone. The Taylor-Maccoll equation is initialized by velocity components, V_r and V_θ, at a shock cone edge condition (where θ, the general angle of inclination from the z-axis equals θ_s, which is chosen by the user). By incrementally decreasing θ towards the cone axis until V_θ switches sign, the final θ defines the theoretical conical body (θ_b) supporting the respective oblique shock.

At the conclusion of the integration routine, directional velocity components (V_r and V_θ) are known at increments of θ from the chosen shock angle, θ_s, to θ_b. These discretized elements make up an axisymmetric conical velocity field about the z-axis, where the velocity at a given θ is the same for any range of ϕ and is independent of r. After implementing a Taylor-Maccoll flow solver within the geometry development script, final θ_b values stemming from various θ_s and cruise speed selections were validated against Ref. [2] within 0.1 degree, as shown in Fig. (4.3).

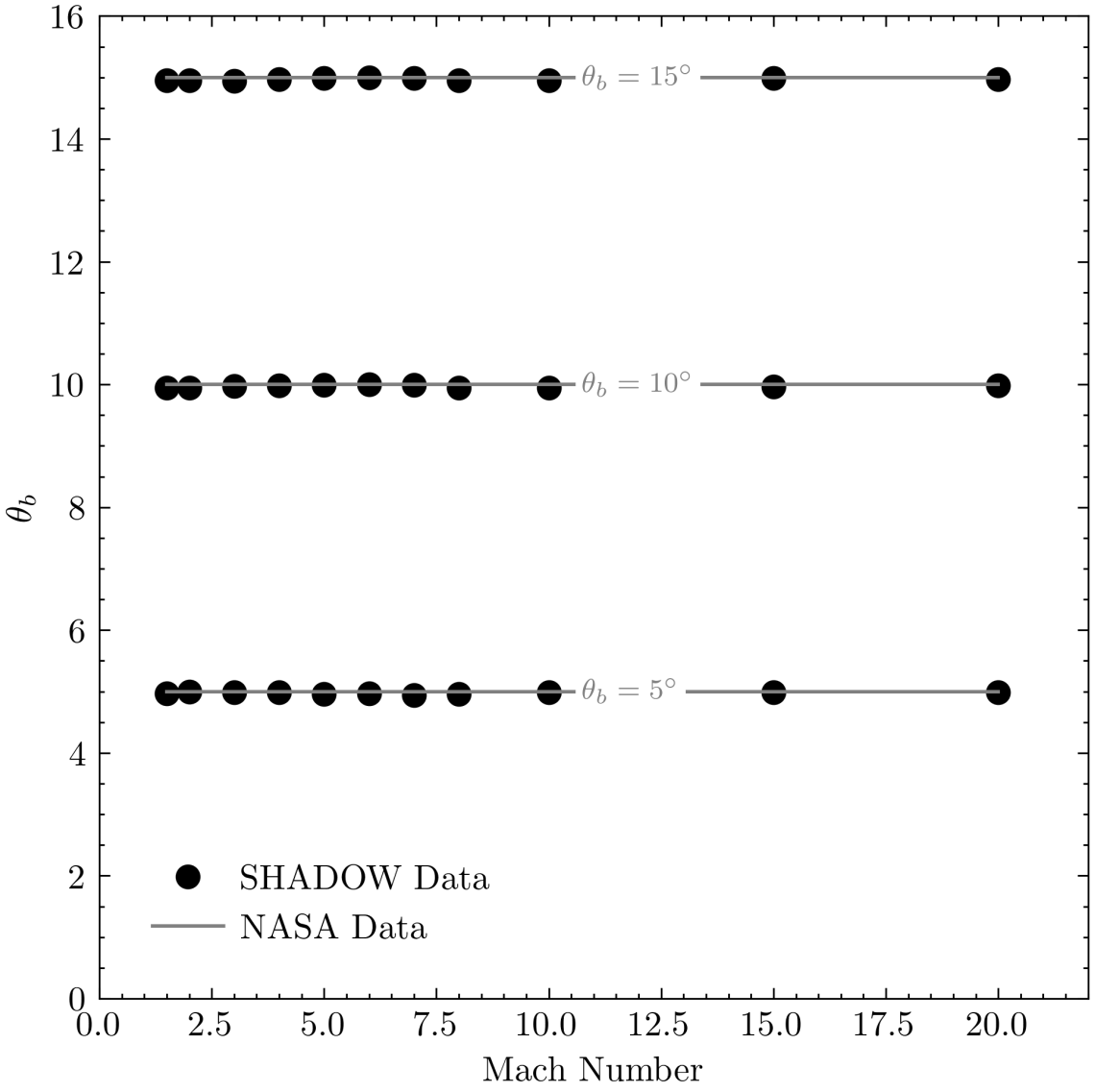


Fig. 4.3: Taylor-Maccoll Verification Cases [2]

Streamline Tracing

The lower surface of a waverider, also known as the *compression* surface, traps the compressed air within the shock wave under its face. Waverider compression surfaces are carved from the desired flow field via streamline tracing [1]. In addition to guaranteeing flow definition adherence, streamline tracing is paramount for propulsion system prediction and development, which is highly coupled to aerodynamics in hypersonic vehicle design.

Streamline tracing is accomplished by integrating a fluid particle's velocity within a flowfield, uncovering its positional information along the flow. The conical flow field in the present study was derived by numerically integrating the Taylor-Maccoll equation; hence, the flow field is only known at *discrete* increments of θ as dictated by the user-input step size. To delineate velocity vectors in between known θ rays to produce a 'smooth' particle path, the present study heavily relies on either forward or backward-difference interpolation between known Taylor-Maccoll solution rays to provide the velocity needed for the streamline integration. The discontinuity in the flow inherent to shock structures prevents extrapolation, forcing flow between the conical shock angle (θ_s) and supporting conical body angle (θ_b). Thus, points outside the shock are forced between the two next closet points within the shock. For steady flow, the positional path at which a fluid particle flows in a spherical coordinate system is as follows

$$\begin{bmatrix} \frac{dr}{dt} \\ \frac{d\theta}{dt} \end{bmatrix} = \begin{bmatrix} V_r \\ \frac{1}{r} d\theta \end{bmatrix} \tag{4.14}$$

Note that in the equations above, t represents an integration step and can be thought of as a *synthetic* time step not actually representative of time, since streamlines may change with time for unsteady flow scenarios. Further, r is generally defined in spherical coordinates as the vector intersecting the θ and ϕ planes. It should be noted that for waveriders whose nose coincide with the shock cone vertex, known as *Class A* waveriders, Taylor-Maccoll flow must be defined at the vertex in order to avoid mathematical singularities

$$\begin{aligned} r &= 0 \\ \therefore \frac{\partial \theta}{\partial t} &= 0 \end{aligned} \tag{4.15}$$

Equation (4.14) is utilized to trace a fluid particle's path from each leading edge point. Since Eq. (4.14) must be solved numerically, a RK4 routine was implemented, which utilizes a synthetic time step to solve Eq. (4.14) until the designated length of the vehicle is reached. The number of synthetic time-steps is consistent for each streamline and is user-specified as n. However, a non-uniform geometry stemming from different leading edge points requires each streamline segment to have a respective unique, precise length to exactly intersect the base surface, which was predefined by the non-dimensional length of 1, after n steps. Hence, the magnitude of each time step fluctuates for each streamline stemming from geometrically unique leading edge points. In the lack of a smooth function to represent each streamline, it is obligatory to use a numerical method to solve for the synthetic step size required to integrate each streamline. The Secant Method is a numerical method well suited to functions whose derivative information is not known, which predicts streamline time steps via

$$x_i = \frac{x_{i-2} f(x_{i-1}) - x_{i-1} f(x_{i-2})}{f(x_{i-1}) - f(x_{i-2})} \tag{4.16}$$

where f is the result of the RK4 integration after n steps at the present step size and i is the secant iteration counter. To initialize the Secant Method, the difference between the leading edge z point and the z_{end} point is calculated, divided by n, and then divided again by $\cos\theta_s$ to account for the additional length traveled in the y direction. A central difference approximation provided synthetic derivative information to initialize the secant method. A convergence tolerance of $1e-3$ was used as a blend between accuracy and speed. It should be noted that the secant method is not guaranteed to find a root, which is more common in non-smooth functions: Due to shock wave discontinuities in the flow, the secant method tolerance was relaxed even further if the solver could not find a solution after 3000 iterations. At the conclusion of the lower surface generation, streamlines stemming from

each leading edge point were defined by a collection of discrete velocity points as dictated by the step size and n, the number of steps specified by the user.

Surface Mesh Generation

Research has shown that defining an expansion surface generates marginal performance gains [1], especially at higher mach numbers [48]. However, waverider upper surfaces are not constrained to one certain design. In addition to aerodynamic considerations, upper surface design is predominant in fulfilling volumetric constraints imposed by the unavoidable need for fuel, avionic systems, and in some cases – passengers [49]. In the interest of simplicity and adherence to research goals, a freestream upper surface was generated to serve as the waverider upper surface.

By keeping x and y points the same from the leading edge to the trailing edge, the upper surface is obtained only by adding steps to the z coordinate until the base surface is reached. The number of z steps can be, but is not required to be, synonymous to the number of synthetic time steps, n, used to generate the lower surface. The step size used to mesh the top surface isn't complicated by unique geometries, and therefore is defined as the length of the freestream line-segment divided by the user-specified n points.

At the present phase of the script, the top and bottom surfaces are defined by a three-dimensional Cartesian point cloud. Various computational techniques exist to extrapolate a triangular mesh from the point cloud representative of the waverider geometry. The point cloud was triangulated over the top and bottom surfaces in a counterclockwise manner.

Inherent to a streamline-traced compression surface and a freestream upper surface, the base surface must be parallel to the xy-plane. Thus, the z-coordinate is consistent across the base plate and is equal to z_{end}. Further, the top and bottom surfaces intersect at a perfectly sharp leading edge.

Using the simple triangular loop to close each of the surfaces to a watertight mesh, SHADOW writes the mesh file in TRI (native to NASA's Cart3D [50]) formatting. Popular fluid flow solvers, like NASA's CBAero [51] or Utah State University's HI-Mach [52], use

triangulated files as input to perform panel-based aerodynamic computations. It is important to note that the mesh must be defined as dictated by the right-hand rule to ensure that the surface normals are correct for the later pressure calculations.

4.1.2 Aerodynamic Prediction Model

Aerodynamic information associated with any waverider geometry is needed to inform the iterative process associated with the subsequent implementation of an exhaustive search method. Due to the later effectuation of an exhaustive search method, engineering tools that provide rapid results were preferable. For this reason, Utah State University's HI-Mach code [52] was selected to output aerodynamic information. In contrast to NASA's popular engineering-grade solver, CBAero [51], HI-Mach is platform independent, enabling the overarching design space exploration to be executed via Powershell (Microsoft), zsh (Mac Unix), or bash (Linux) terminals via FORTRAN compilers and Python interpreters. HI-Mach has been validated with available experimental and theoretical data [52].

Geometry Considerations

In an effort to understand general trends, the waverider body was analyzed in a 'clean' configuration (i.e., without subsystems like the propulsion system, landing gear, and control surfaces) with heuristically-derived center of gravity information. Each waverider was dimensionalized by scaling all surfaces to 60 meters, typical of hypersonic transport vehicles [16].

HI-Mach was configured to neglect base drag, which although yielding a less-accurate answer [53], provides extra simplicity for the solver and subsequent design space exploration. Waverider base surfaces are usually analyzed at vacuum conditions, an accurate assumption especially for unpowered glider vehicles [54] [55], or utilizing a Gaubeaud relationship.

Mesh resolution is also closely related to the results stemming from panel based solvers like HI-Mach. As such, a mesh resolution convergence study was performed at various Mach numbers. It was determined that a total number of panels of 682 resulted in the expected

level of accuracy for the present research. As shown in Fig. (4.4), 682 panels provides a blend between accuracy and speed.

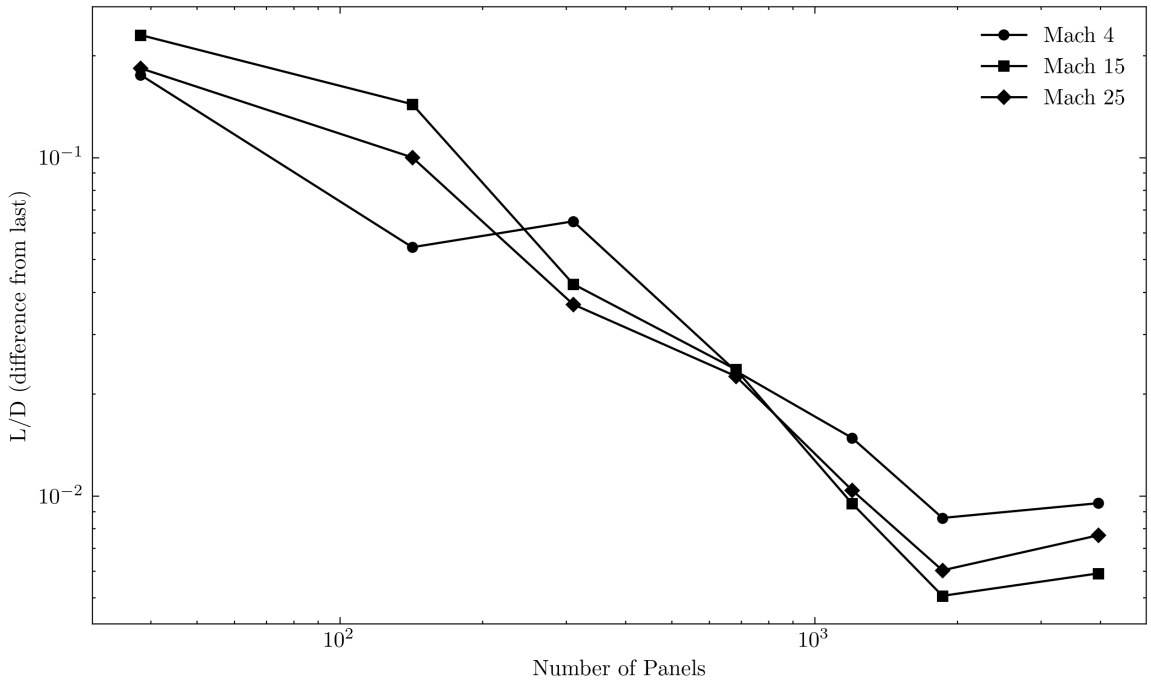


Fig. 4.4: Mesh Resolution Study (Mach 4)

Aerodynamics

For hypersonic flow, panel-based impact solvers like the Tangent Cone and Tangent Wedge approaches provide promising results, especially for lateral stability when analyzed at zero degrees angle of attack [56]. Combining impact methods with shock expansion theory and the modified-Newtonian method also has produced promising results, though not more beneficial than implementing one prediction model alone [57]. Previous research indicates that the tangent cone method under-predicts waverider pressures, while the tangent wedge method over-predicts waverider pressures [3]. Further, the leeward side calculations have been shown to agree with the Prantl-Meyer expansion solution better than freestream impact methods, especially at lower Mach numbers [57].

A blended tangent-cone/tangent-wedge impact solver, set forth in Ref. [3] was utilized for the windward solver within the HI-Mach framework. The blended method exploits the best of both impact methods; tangent wedge solutions are more accurate when analyzing the flow field close to the shock, while tangent cone solutions contrastingly produce accurate results when the flow field is close to the generating conical body. By implementing a quadratic interpolation between the two methods, pressure solutions show outstanding results for *waverider* panel pressures, agreeing well with exact Taylor-Maccoll pressures [3], as shown in Fig. (4.5).

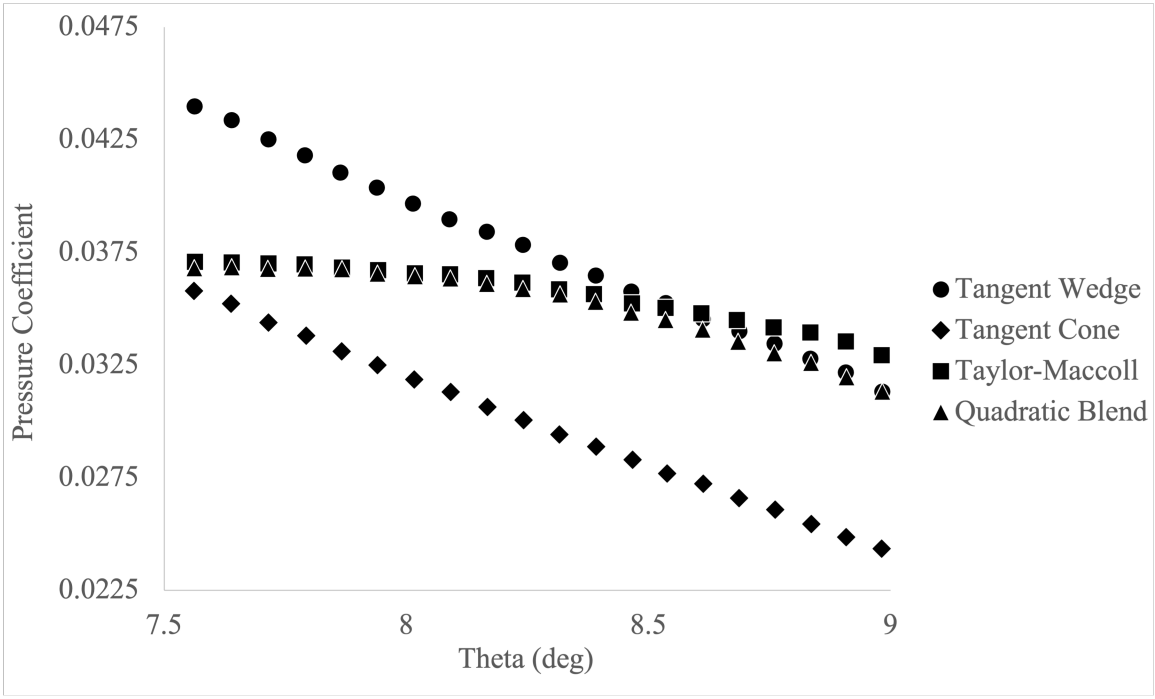


Fig. 4.5: Pressure Solutions for a Mach 6 Waverider with 12° shock, data digitized from [3]

The leeward solver was configured to utilize the Prantl-Meyer expansion. Zero contribution to aerodynamics was assumed for shadowed flow, which lies in line with other panel-based solvers like modified Newtonian method [58], and empirical derivation at high mach numbers [59].

HI-Mach reports non-dimensional aerodynamic coefficients in the body-fixed coordinate frame. To transform body-fixed forces and moments to the aerodynamic (wind) coordinate frame,

$$\begin{aligned} L &= -(N\cos\alpha - A\sin\alpha) \\ D &= -(A\cos\alpha\cos\beta - Y\sin\alpha + N\sin\alpha\cos\beta) \end{aligned} \tag{4.17}$$

Since HI-Mach reports already non-dimensionalized lift and drag coefficients, Eq. (4.17) was altered to use non-dimensional forces, producing C_L and C_D values

$$\begin{aligned} \text{Axial Force} \equiv A &= -C_F|_z \\ \text{Side Force} \equiv Y &= C_F|_x \\ \text{Normal Force} \equiv N &= -C_F|_y \end{aligned} \tag{4.18}$$

To verify that the basic aerodynamic coefficients are reported correctly, a set of near-optimal waverider vehicles were generated, analyzed, and compared against Kuchemann's classical L/D limit, shown in Fig. (4.6). HI-Mach tends to underestimate L/D but agrees well with the trend, which is satisfactory for the present research.

To calculate yawing and rolling moment with respect to sideslip, it was assumed that aerodynamic moments exhibit a linear relation along reasonable sideslip angles. Such an assumption is generally confirmed in the results section and agrees with previous research [21]. To circumvent instabilities around zero degrees, led in part by freestream conditions (for α sweeps) or other impact-based artifacts, larger perturbed cases at ± 5 degrees β were used to provide reference points for the slope estimation for either stability derivative. A similar assumption was made for pitching moment in respect to α, which was perturbed ± 5 degrees.

Further validation on the aerodynamic prediction model, HI-Mach, was performed by comparing to exact CFD solutions provided in Maxwell's paper [60]. The exact mesh, along with some input parameters, used in Maxwell's paper was graciously provided by the author, which was further interrogated with a CAD software to extract the remaining

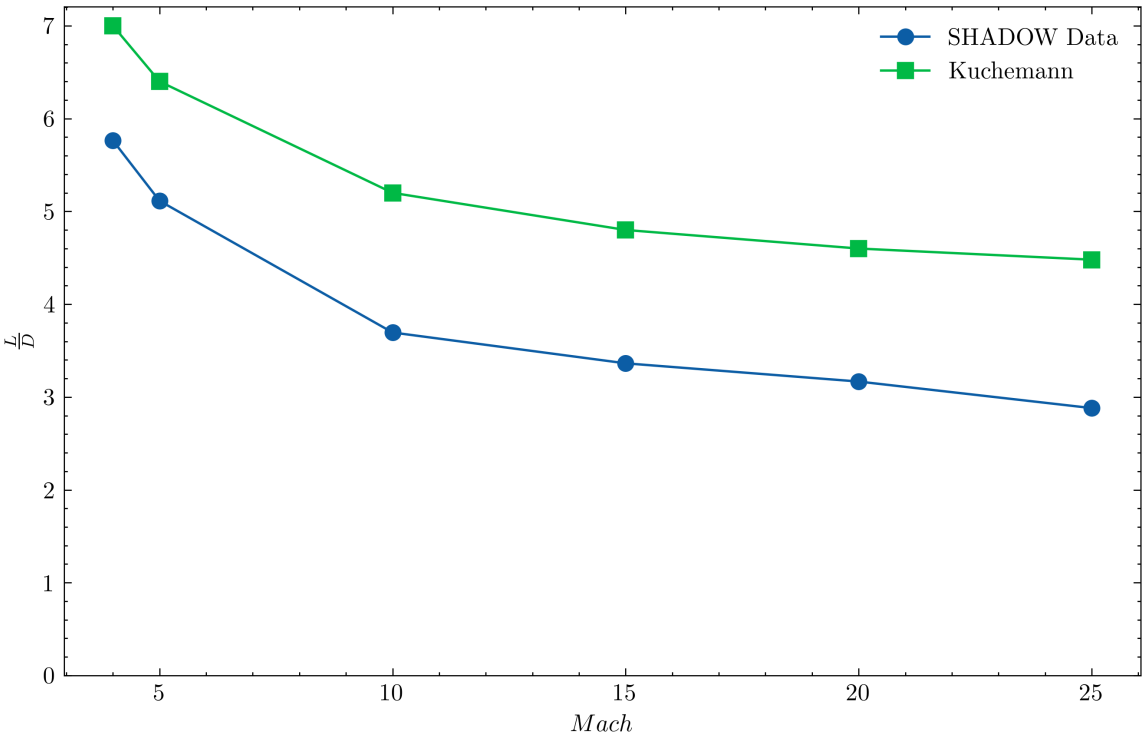


Fig. 4.6: Lift to Drag Ratio Against Mach

necessary information to input to HI-Mach. Maxwell's waverider mesh was analyzed using HI-Mach in the SHADOW code along a range of α and compared against digitized data from Maxwell's paper [60]. Results show general agreement for the pitching moment trend, as shown in Fig. (4.7). HI-Mach is configured to neglect base-drag, which conforms with Maxwell's aerodynamic shadow assumption.

Viscous Modeling

In agreement with mid to low-fidelity expectations for the present research's rapid-aerodynamic analysis, various methods and assumptions were considered to provide good results without extra time expense. For example, the widely-regarded 'Chi-bar' affect, which classifies boundary layer effects, was disregarded in the elementary heating model used in the HI-Mach framework, largely because the scale of the waveriders' length ($60m$), is proven [61] to be classified as a weak interaction.

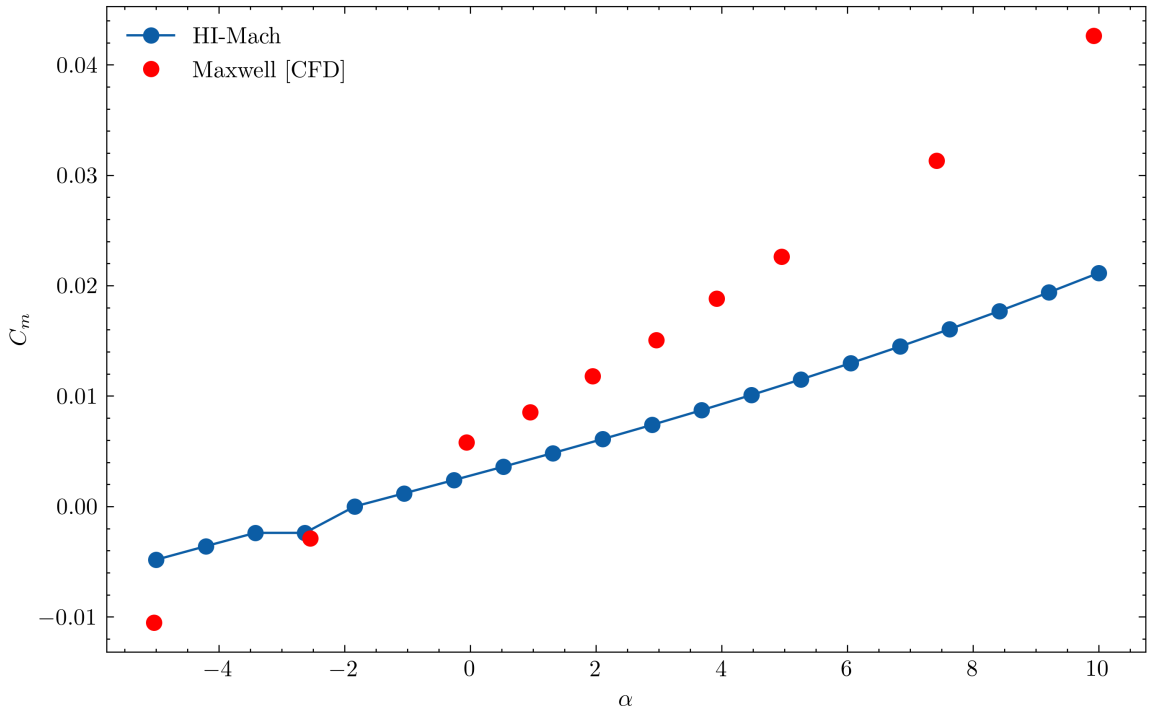


Fig. 4.7: Moment Validation Case

Boundary layer and turbulence modeling has been a topic of active research for decades, and various methods exist to predict both boundary layer behavior and its influence on aerodynamics. Enthalpy relations are set forth by Meador and Smart [62] and Eckert [63].

In an effort to match previous research on waveriders [20] [16], Eckert's reference temperature method, informed by White's relations [64], was used. Eckert's reference temperature corrects Eq. (4.19) and Eq. (4.20), traditionally incompressible equations, for compressible flow by analyzing the Reynolds number at the reference temperature, denoted by the asterisk. Thus, the laminar skin friction coefficient is computed as

$$c_{f_l} = \frac{0.664}{\sqrt{Re_x^*}} \tag{4.19}$$

and for turbulent flow, as given by White's method [64]

$$c_{f_t} = \frac{0.0592}{(Re_x^*)^{0.2}} \tag{4.20}$$

A turbulent boundary layer produces heavier aerodynamic heating than laminar boundary layers [65], which occur while cruising at higher altitudes [66]. More comprehensive transitional methods, which predict the transition between laminar and turbulent models, provide accurate results for waverider vehicles [67] [1]. Further, instead of considering the transition from laminar to turbulent as abrupt, the transitional Reynolds number can be set forth by a blending function and associated γ function given in Ref. ([1]).

$$c_{f_{tr}} = (1-\gamma)c_{f_l} + \gamma c_{f_t} \tag{4.21}$$

where γ is a function of Reynolds number along the length of the vehicle

$$\gamma(x) = 1 - \exp -3\exp\left[\frac{\ln 2}{5x_{ti}}(Re_x)_{ti}^{0.2}(x - x_{ti})\right]^2 \tag{4.22}$$

Fluid conditions play a major role in viscous aerodynamics; gas properties for fluids at various conditions are provided by a gas table, synonymous with a table used by Ref. [51]. Fluid conditions directly influence Reynold's number, thus making Reynold's number an indirect constitute of L/D ratio. As altitude increases, the flow becomes more viscous-dominated, usually telling of laminar flow – decreasing L/D ratios [68]. Data for elevations and wall temperatures was taken directly, or roughly interpolated, from Corda's work [16], which agrees well with Ref. [69].

4.1.3 Design Space Exploration

In the present study, waverider vehicles are parameterized using any leading edge polynomial projection. In addition, SHADOW allows for any n^{th}-order polynomial definition, opening up possibilities to experiment with multi-DOF studies that may discover waverider geometries with a desirable blend of aerodynamic traits. Polynomials are numerically discretized by an odd number of points which exhibit symmetry about the y-axis: A set of 19 points was projected on the xy-plane; one point in the middle, and 9 reflected points across the y-axis. Polynomial coefficients were generalized using the following relation

$$A_3x^3 + A_2x^2 + A_1x + A_0 \tag{4.23}$$

where A_1 is held steady to zero (see Eq. (4.4) and A_0 is the y-intercept.

Exhaustive searches across a discretized sweep of possible polynomials provide a guaranteed optimal solution within the bounds in which it operates, providing excellent fidelity for sound conclusions in the selected design space. Holding the shock angle to an 'optimal' value as dictated by Ref. [1], combinations of possible A_2 and A_0 values are analyzed, uncovering polynomials which produce max L/D waveriders. Limited-third order cases were also executed by holding the y-intercept constant in addition to the shock angle.

The uniquely defining constraint of the present research is discrete longitudinal and lateral stability requirements. To determine the discrete affects of each stabilizing constraint removed from any coupling effects, each stability constraint was overlaid on the L/D contours independently, uncovering polynomials which produce stable waveriders.

$$\begin{aligned} C_{n_\beta} &> 0 \\ C_{l_\beta} &< 0 \\ C_{m_\alpha} &< 0 \end{aligned} \tag{4.24}$$

To investigate the effect of shock angle selection in addition to polynomial parameterization, contour plots at Mach 4 were repeated at several shock angles. Hence, SHADOW may unveil where to find near-optimal waverider *geometries* flying in various *fluid conditions*.

CHAPTER 5

RESULTS

Looking at conically-derived, polynomial-parameterized waveriders, the following results visualize the strength of association between polynomial parameters and aerodynamic values, including the possibility for stability in waverider design space.

5.1 Stability Derivative Investigation

Using the same parameters to generate the waverider found in Fig. (4.1), waverider vehicles were generated at various Mach numbers using their respective shock cones. Figs. (5.1)-(5.3) outline aerodynamic moments associated with these waverider vehicles as a function of β (for C_l and C_n) and α (for C_m) through a range of cruise speeds. Fig. (5.1) shows that lateral stability increases with increasing Mach number. Interestingly, the sign of pitch stability does not change with changes in cruise speed.

It is important to note that the trends shown in Figs. (5.1)-(5.3) disagree with historical, albeit limited, data on a hypersonic vehicle along mach. This disagreement stems from the waverider shape's dependance on cruise speed. To prevent SHADOW from generating waveriders from shock cones associated with a designated Mach number, the default waverider found in Fig. (4.1) was run along increasing speed without changing the vehicle design. This decoupling proved beneficial, producing correct trends as shown in Fig. (5.4).

5.2 Waverider Design Space Exploration

In this section, selected coefficients define two-dimensional space, with additional dimensionality associated with aerodynamic information contoured within the design space. To avoid hidden consequences of stability coupling embedded into the results, each stability coefficient is overlaid discretely of one another. Further, each of the results are reported at several cruise speeds to visualize the effect of cruise speed on waverider design space.

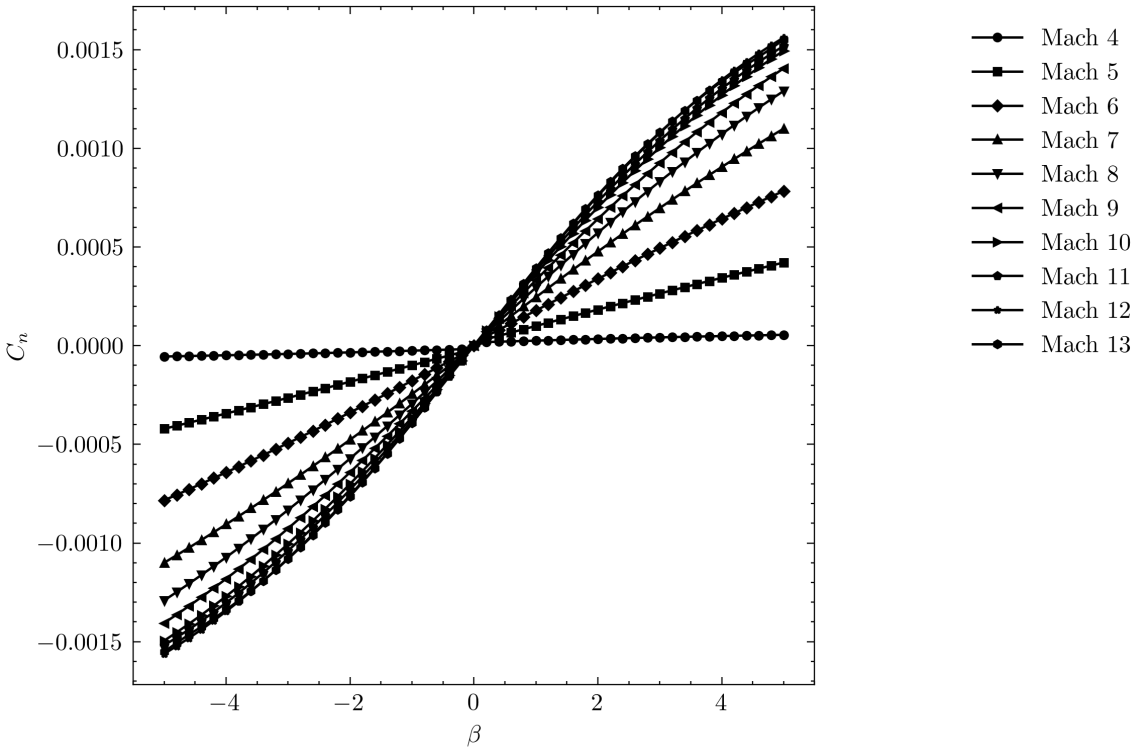


Fig. 5.1: Yawing Moment as a function of β Over a Range of Cruise Speeds

5.2.1 Second-Order

An exhaustive search of second-order polynomial possibilities was performed by perturbing A_2 from -2 to 2 along 80 points, and A_0 from -1 to -0.01 along 20 points. Exhaustive searches were performed identically at Mach 4, 10, 15, 20, and 25, with the exception of shock angle definition. In this subsection, shock angles were held constant at values as shown in Table 5.1. Further investigation on shock angles sensitivity is discussed in subsequent subsections.

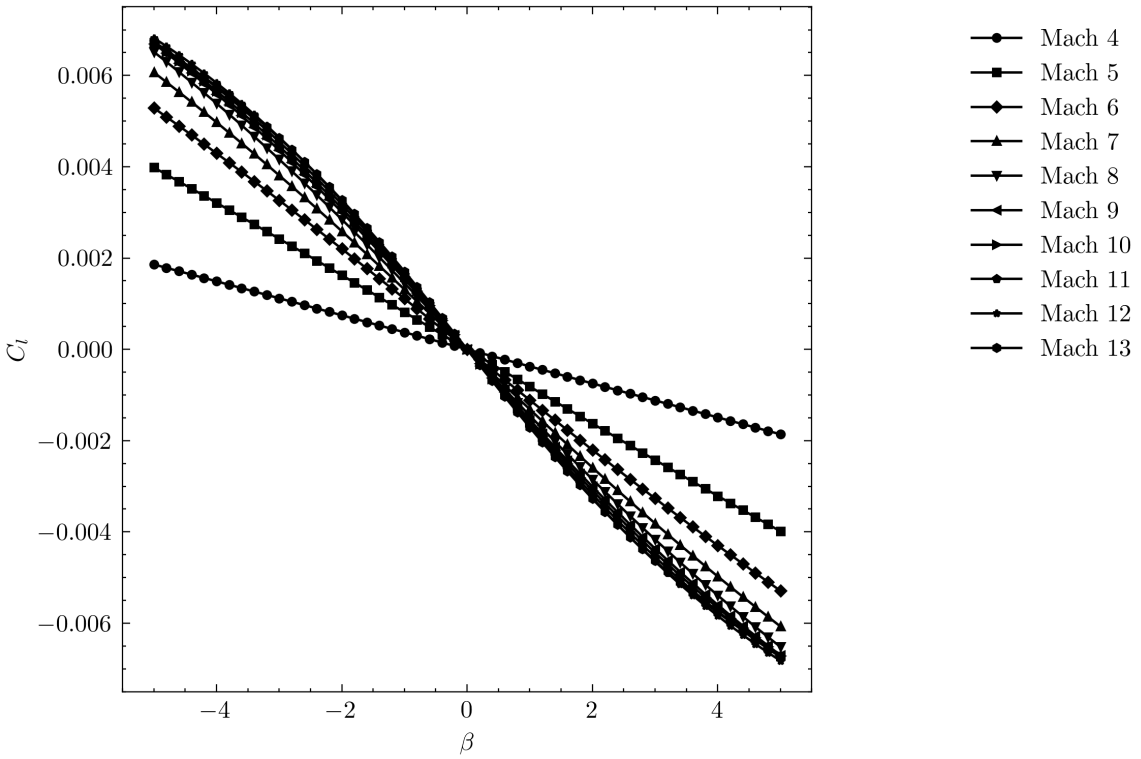


Fig. 5.2: Rolling Moment as a function of β Over a Range of Cruise Speeds

Table 5.1: Shock Angles at Various Cruise Speeds

Cruise Speed	Shock Angle (θ_s)
Mach 4	16°
Mach 10	9°
Mach 15	9°
Mach 20	7°
Mach 25	9°

Figure (5.5) show contours of L/D and regions of stability as a function of A_0 and A_2 at Mach 4. Fig. (5.5) demonstrates that near-optimal L/D waveriders are generated at or near $A_2 = 0$. As A_2 becomes more negative, outboard surfaces curve downwards, increasing yaw stability. Contrastingly, as A_2 becomes more positive, outboard surfaces curve upwards. However, positive quadratics (especially with higher magnitudes) might

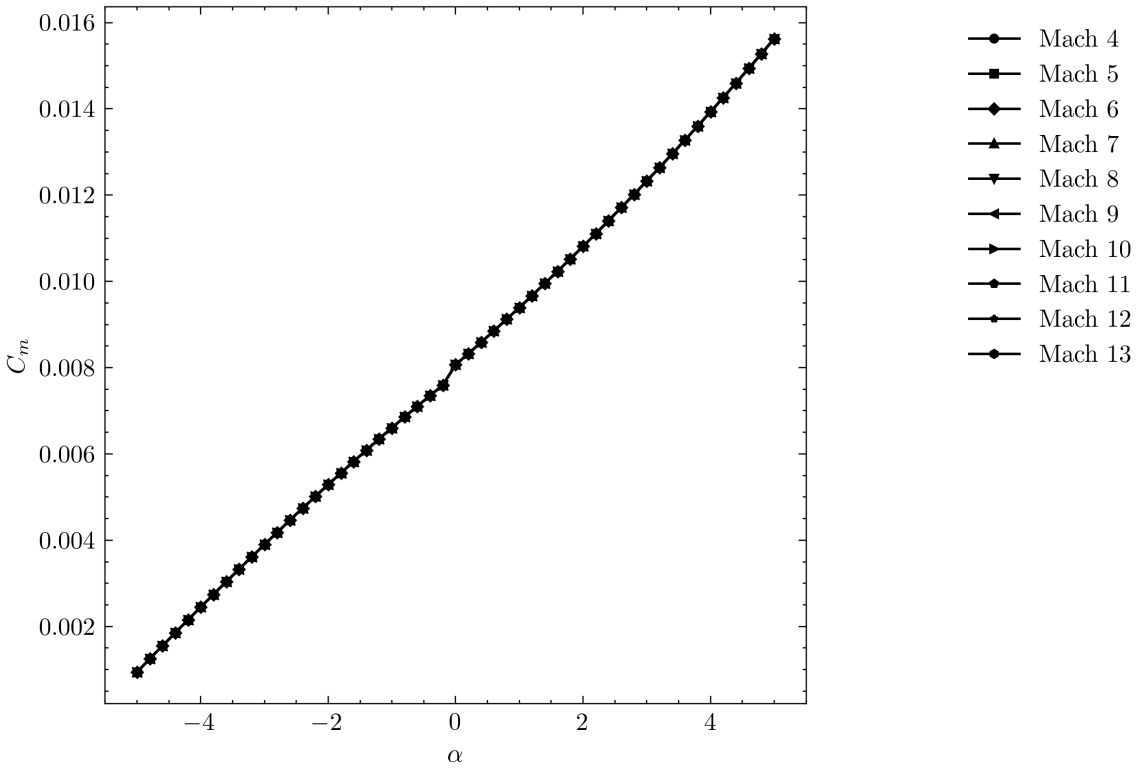


Fig. 5.3: Pitching Moment as a function of α Over a Range of Cruise Speeds

cause the leading edge to move upstream. Hence, portions of the waverider become longer in outboard regions than the center line, which was fixed as the non-dimensional length. These polynomials create a multi-vertex body, as shown in Fig. (5.6c). Despite their reminiscence to the TIE Interceptor from Star Wars, these geometries are unconventional and unsurprisingly produce low L/D values.

The most significant region of Fig. (5.5) is near the top, where y-intercept is small and associated waveriders exhibit pitch stability. A waverider residing in this region is visualized in Fig. (5.6a), which shows a unique lower-surface geometry. Because the y-intercept is small, the streamline tracing quickly reaches the theoretical solid body cone (θ_b), notwithstanding the polynomial shape. The quick convergence on the theoretical solid body cone produces a conical extrusion on the lower surface of waveriders in this region, which may be attributed to the favorable pitch stability.

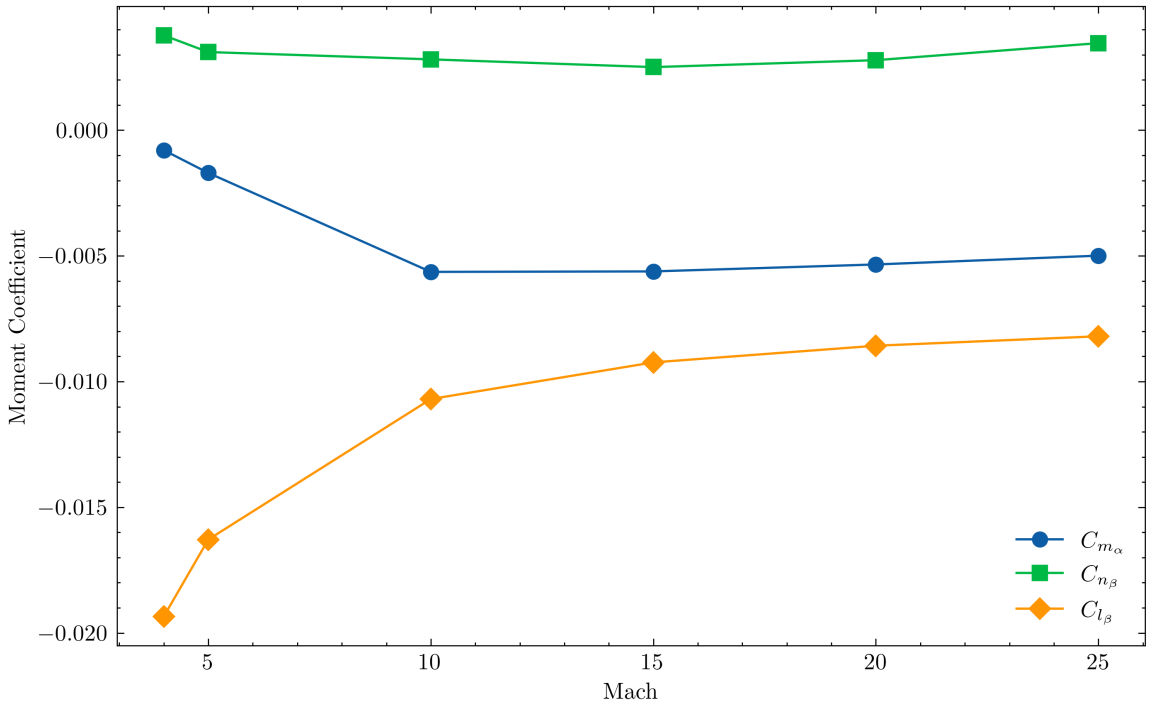


Fig. 5.4: Moment Coefficients along Increasing Speed

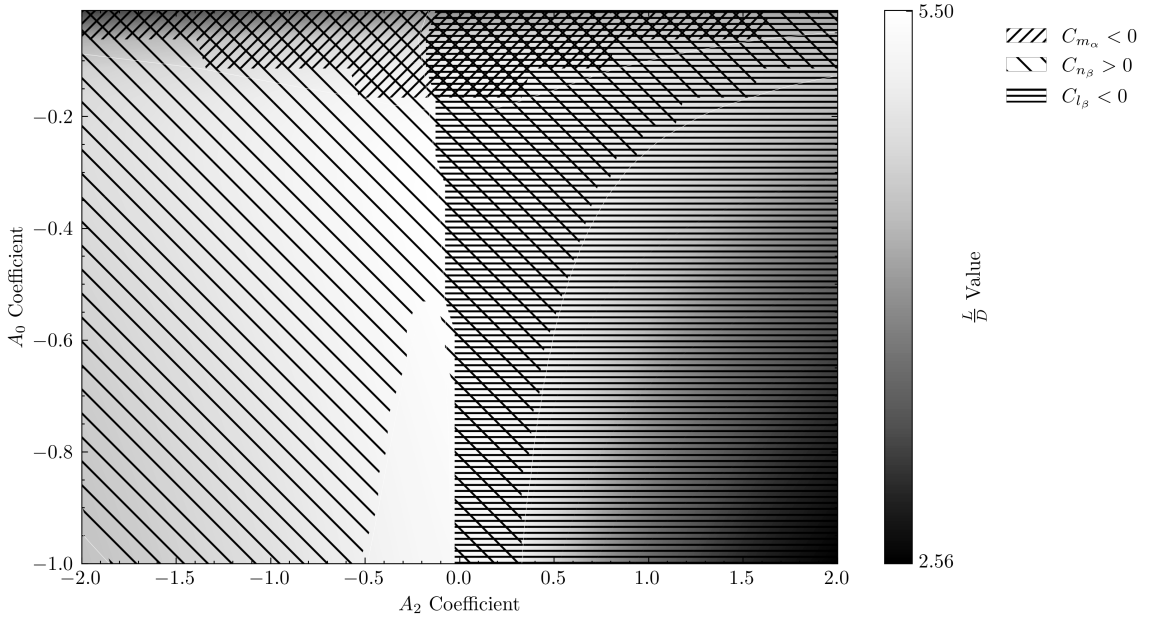
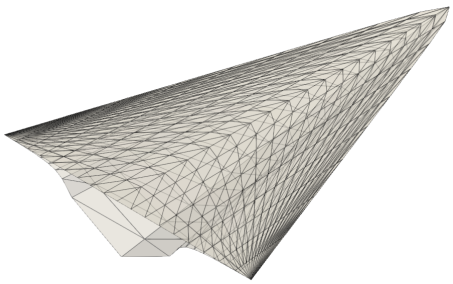


Fig. 5.5: Design Space Exploration with Stability Overlays at Mach 4

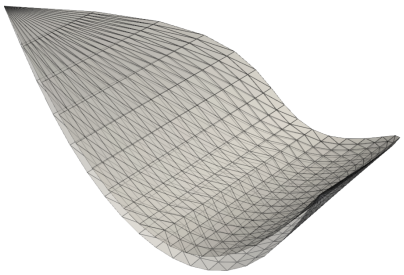
Figures (5.6a)-(5.6d) show waveriders generated at Mach 4 with the parameters found in Table 5.2.

Table 5.2: Waverider Geometry Demonstrations at Mach 4

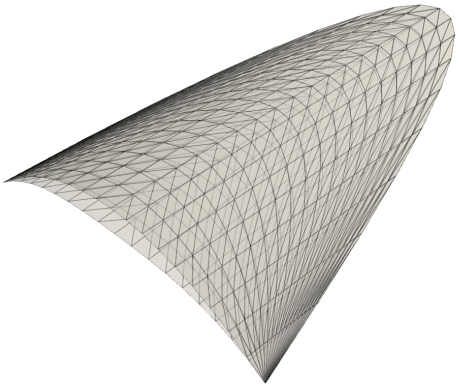
Figure Reference	Description	A_2	A_0
5.6a	Fully Stable	0	-0.02
5.6b	Roll Stable	1	-0.6
5.6c	Yaw Stable	-1	-0.5
5.6d	Near-Optimal L/D	0.1	-0.3



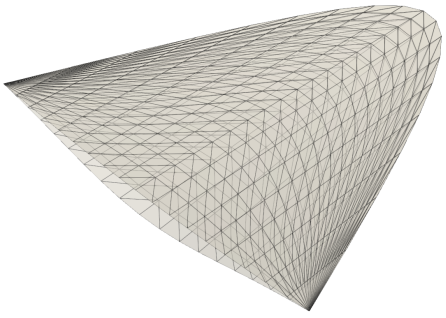
(a) Fully Stable Waverider at Mach 4



(b) Roll Stable Geometry at Mach 4



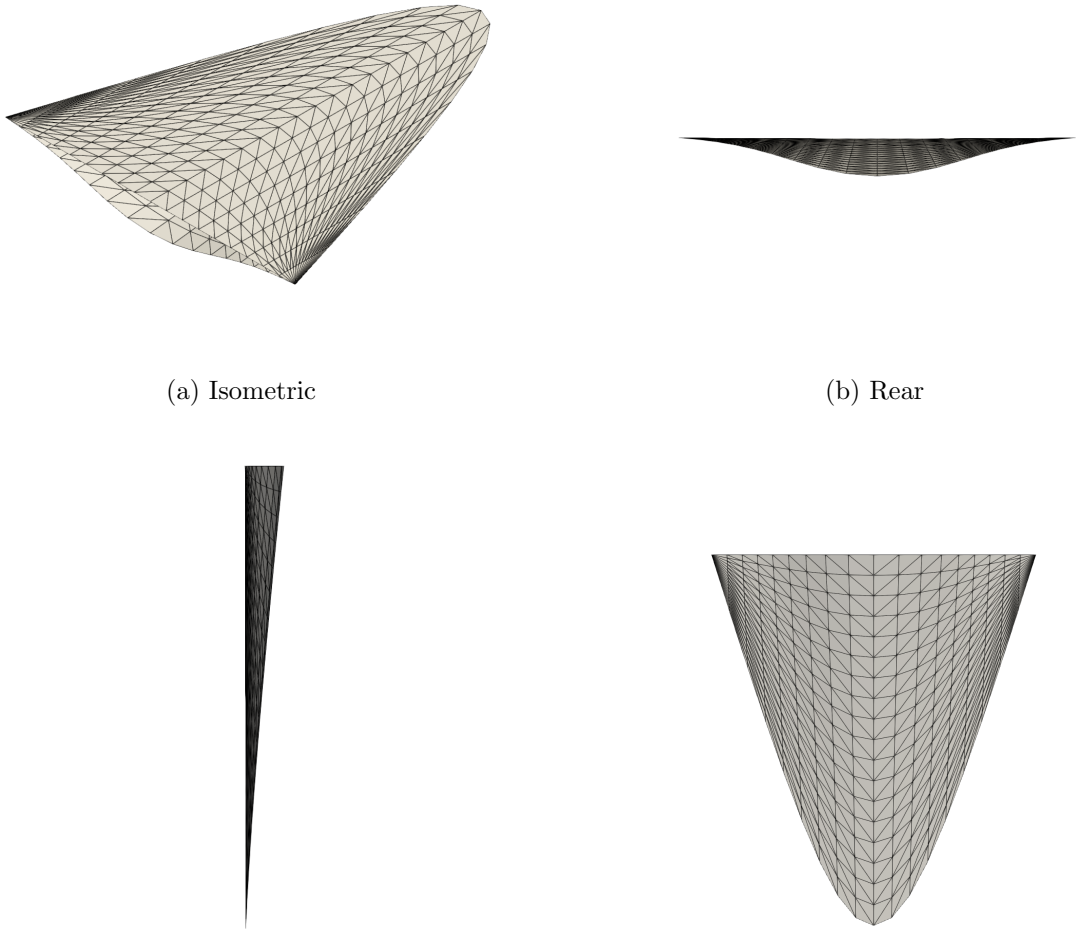
(c) Yaw Stable Waverider at Mach 4



(d) Max L/D Waverider at Mach 4

Fig. 5.6: Comparison of Different Waverider Configurations at Mach 4

Because very small y-intercept values produce smaller L/D values, increasing the y-intercept to the boundary of pitch stability produces the most optimal, fully stable geometry within the limited design space investigated at Mach 4. Generating this waverider by setting $A_2 = 0.01$ and $A_0 = -0.19$, the geometry is shown in Fig. (5.7).



(a) Isometric

(b) Rear

(c) Side

(d) Top

Fig. 5.7: Most Optimal, Fully-Stable Mach 4 Waverider Geometry

Additional contour plots for Mach 10, 15, 20 and 25 vehicles are provided in Figs. (5.8)-(5.11). In the limited design space exploration, near-optimal waverider shapes were shown to be independent of cruise speed. Further, pitch stability appears to be directly tied to

the shock angle at which the waverider was defined from. For example, Fig. (5.9) and Fig. (5.11), which were defined at the same shock angle, have very similar pitch stability regions despite having a different cruise speed. Decreasing the shock angle, which is demonstrated in the Mach 20 case found in Fig. (5.10) decreases the potential for pitch stability. Lateral stability exhibits similar dependencies, but perhaps not as evident due to the larger stable regions across the entire hypersonic regime.

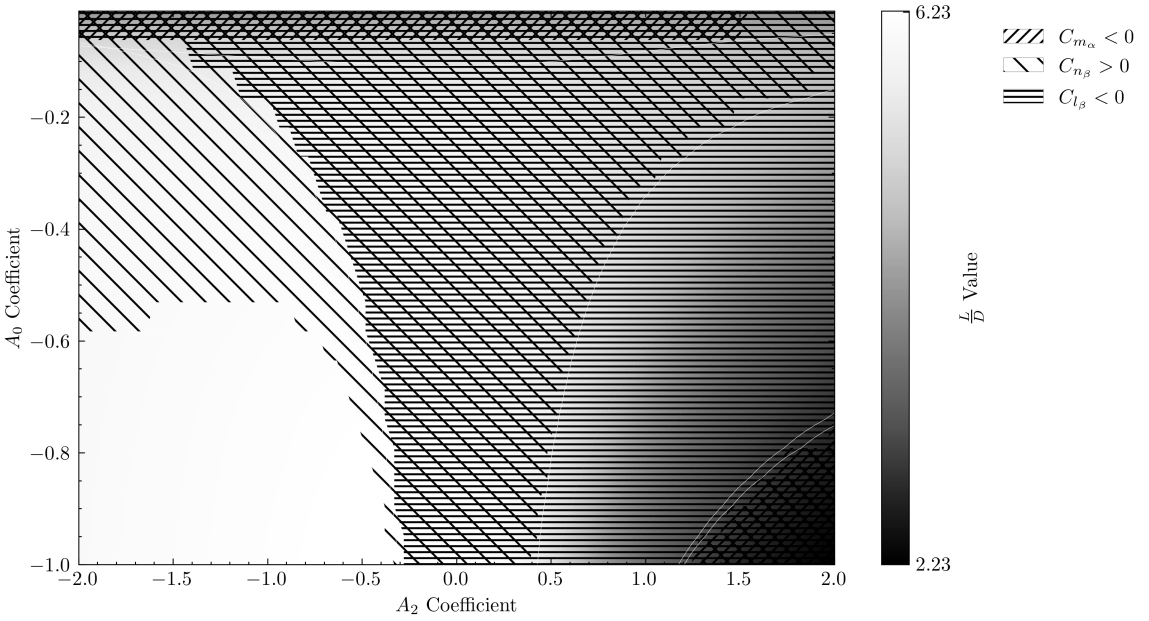


Fig. 5.8: Second-Order Design Space Exploration with Stability Overlays at Mach 10

5.2.2 Shock Angle Optimization

As previously discussed, the shock angle from which the waverider is defined can have significant effects on aerodynamic output. As such, the second-order Mach 4 exhaustive search was repeated at the following shock angles in Figs. (5.12)-(5.14).

1. 14.5°
2. 20°
3. 25°

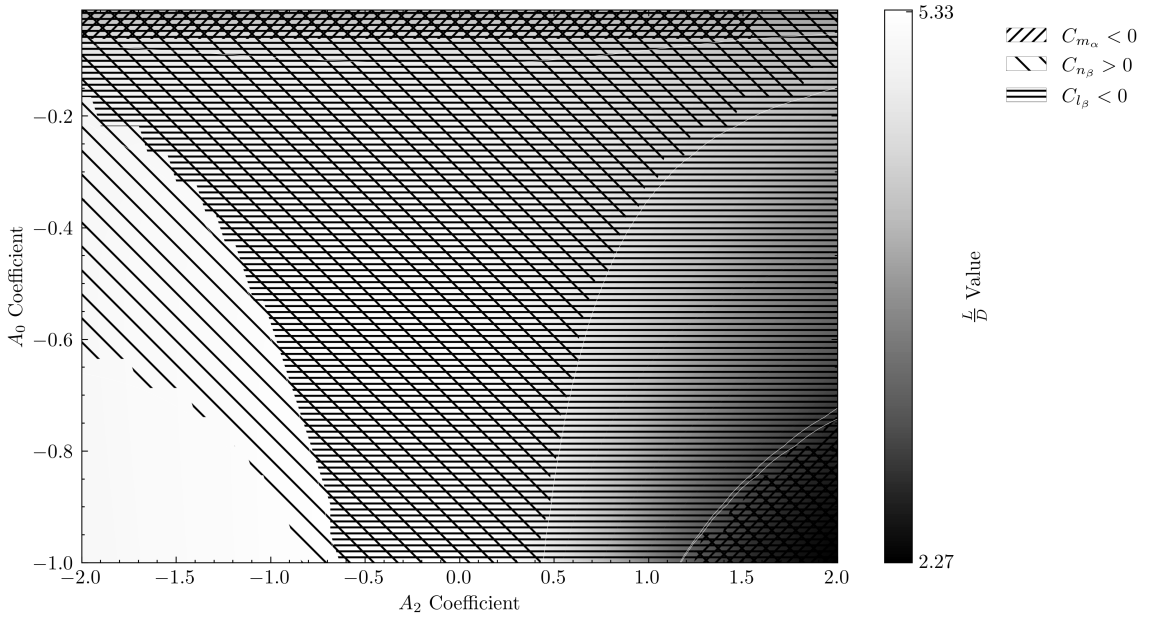


Fig. 5.9: Second-Order Design Space Exploration with Stability Overlays at Mach 15

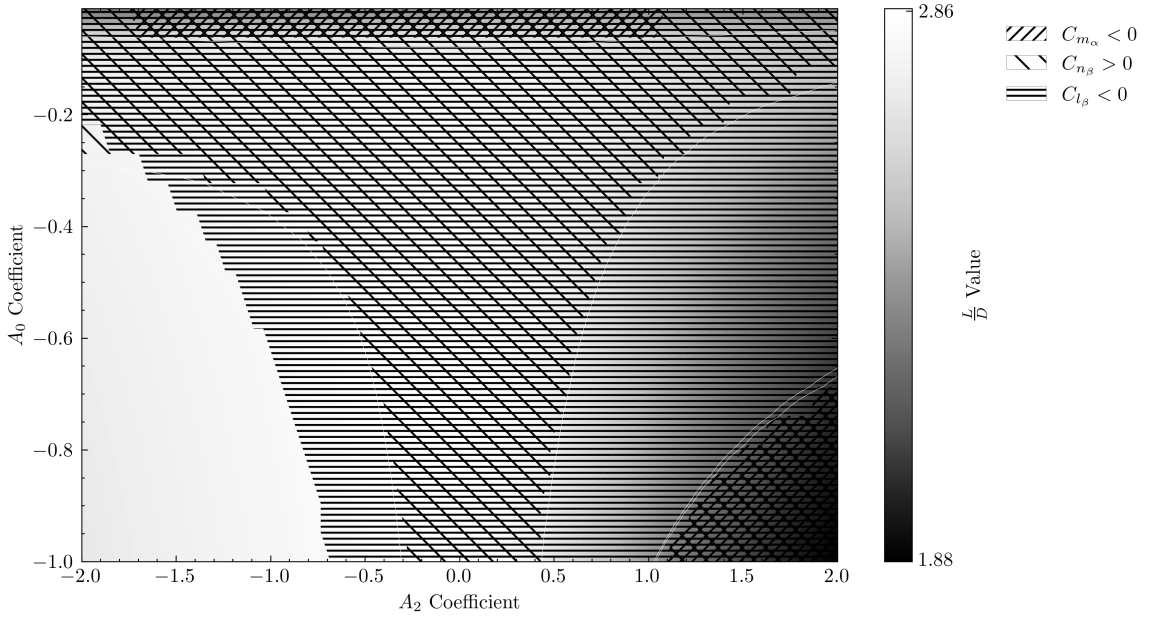


Fig. 5.10: Second-Order Design Space Exploration with Stability Overlays at Mach 20

Figures (5.12)-(5.14) possess similar stability trends to Figs. (5.5) - (5.11). There is, however, a growth in pitch stability along increasing shock angle.

Beyond stability considerations, results at 14.5° show poor L/D values. Different 'max' L/D values for each shock angle definition suggest that aerodynamic efficiency is directly

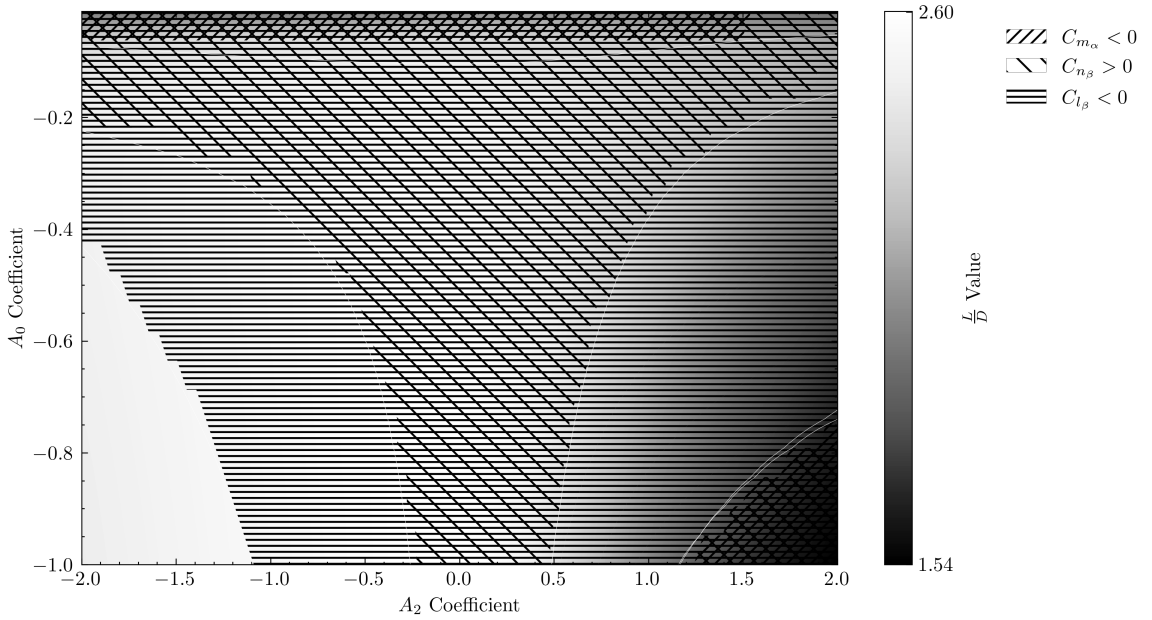


Fig. 5.11: Second-Order Design Space Exploration with Stability Overlays at Mach 25

related to the shock at which the waverider was derived from, agreeing with historical data [1]. Hence, looking at various possible shock structures could yield benefits in the preliminary waverider design process.

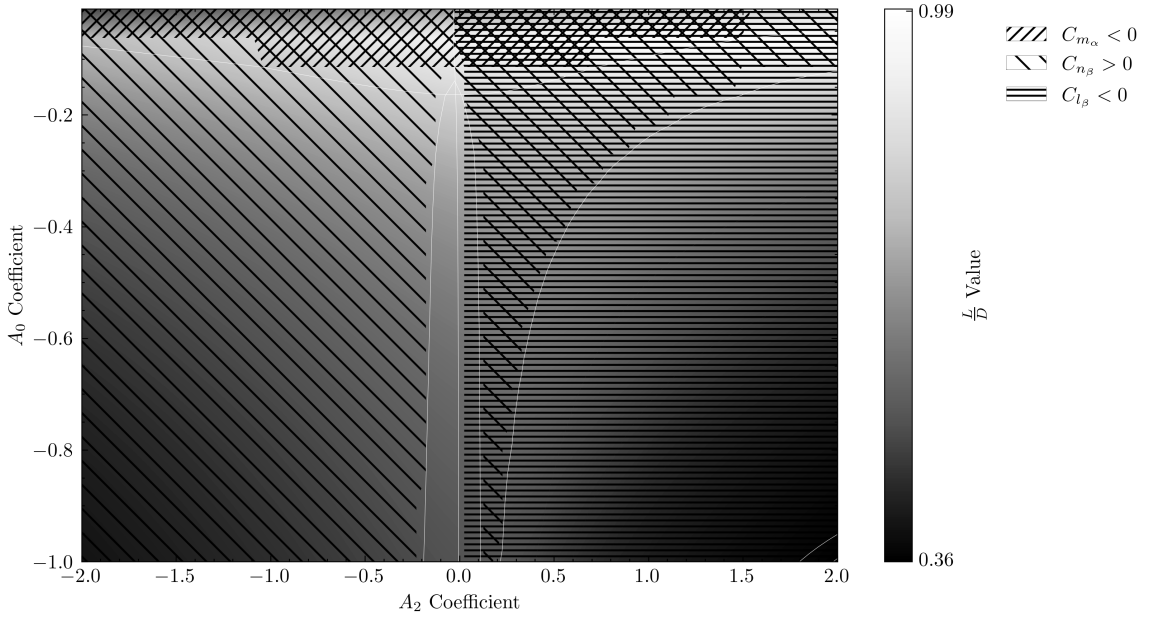


Fig. 5.12: Design Space Exploration At Mach 4 at $\theta_s = 14.5°$

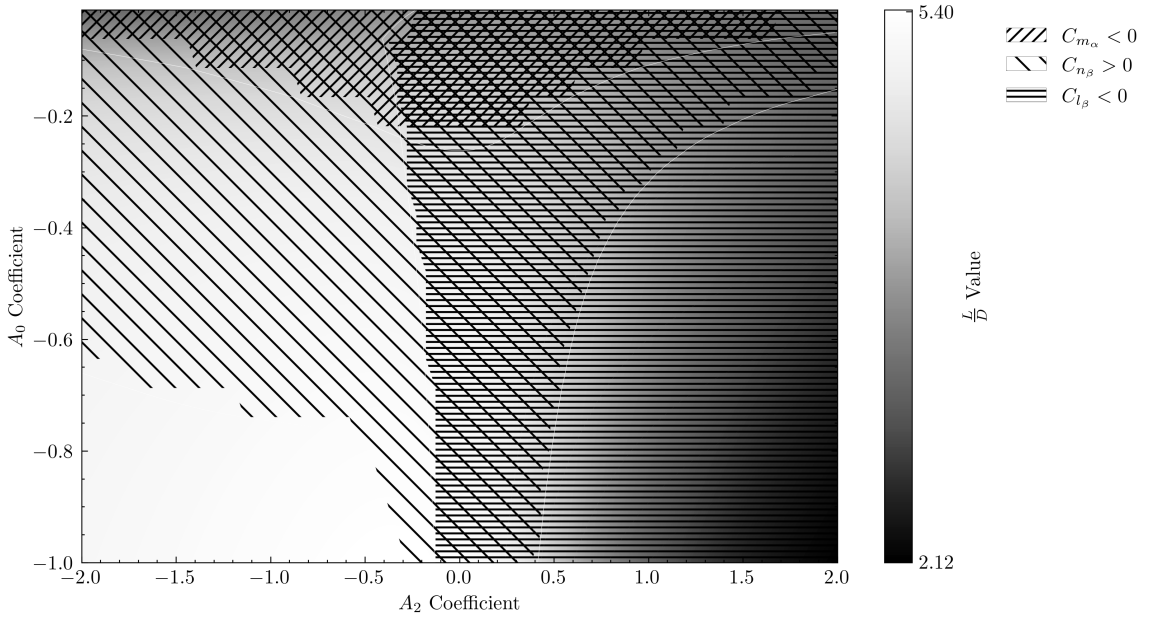


Fig. 5.13: Design Space Exploration At Mach 4 at $\theta_s = 20°$

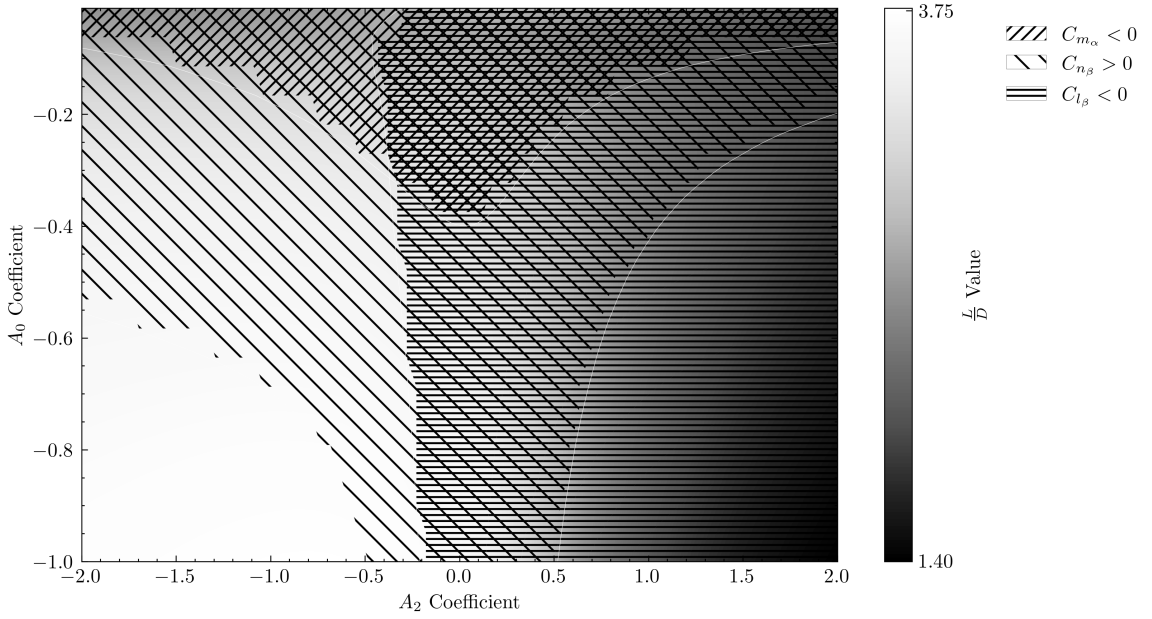
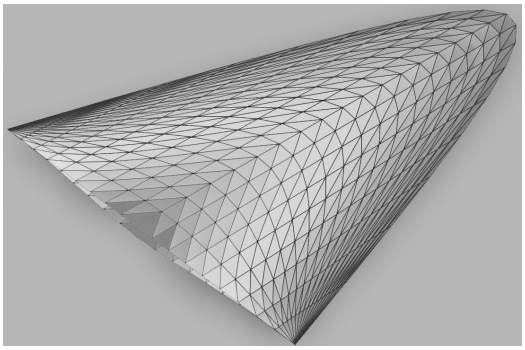


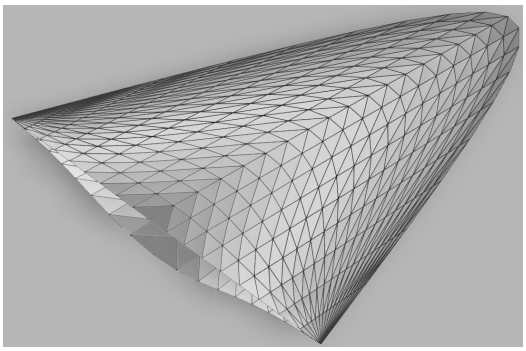
Fig. 5.14: Design Space Exploration At Mach 4 at $\theta_s = 25°$

Additional insight is derived by perturbing the shock used to generate the default geometry (shown in Fig. (5.6a), for which nominal stability values for increasing speed are

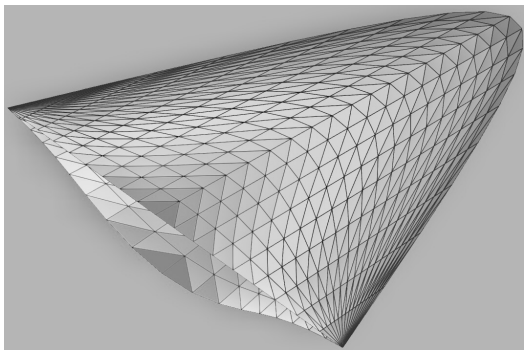
shown in Fig. (5.4). Keeping all other parameters congruent, the shock angle was perturbed $\pm 2^{\circ}$ along with the nominal value, generating the waveriders shown in Fig. (5.15).



(a) Default Waverider Generated with 14° Conical Shock



(b) Default Waverider Generated with 16° Conical Shock, congruent to the Default Geometry found in the Methods Section



(c) Default Waverider Generated with 18° Conical Shock

Fig. 5.15: Visual Effects of Perturbing Shock Angle on Waverider Shape

Plotting the stability coefficients associated with the perturbed waverider geometries against the nominal values contained in Fig. (5.4), Fig. (5.16) importantly deduces that lateral stability is little changed by shock angle, while longitudinal stability is significantly influenced by changes in shock angle. Interestingly, roll stability, in contrast to the other stability parameters, demonstrates an inverse relationship with shock angle.

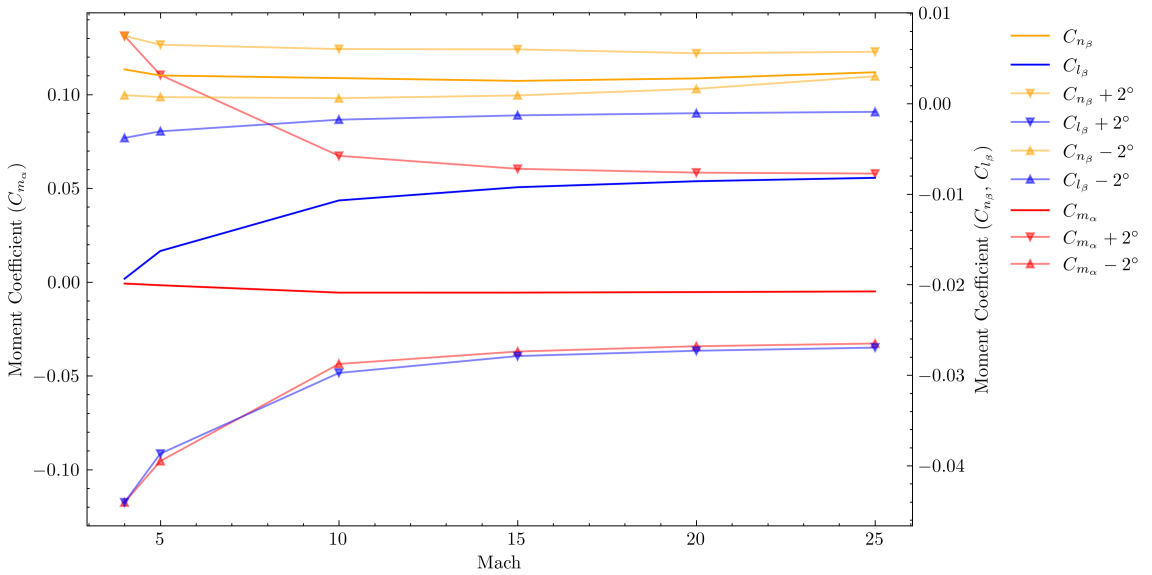


Fig. 5.16: Moment Coefficients along Increasing Speed, with perturbed shock angle

5.2.3 Third Order

To discover if higher-order polynomials opened flexibility to achieve more desirable waverider bodies, exhaustive searches of limited third-order polynomials was performed at opposite ends of the speed regime found in this research: Mach 25 and Mach 4. Holding the shock angles as dictated in Table 5.1, A_0 was held consistent to -0.1. For Mach 25, A_3 was swept from -100 to 100 in 100 steps, while A_2 was swept for -100 to 100 in 100 steps. For Mach 4, A_3 was swept from -45 to 10 in 100 steps, while A_2 was swept for -2.5 to 10 in 100 steps.

Since polynomials whose coefficients were nearly zero were shown to have max L/D values in the second-order investigations, it is unremarkable that aerodynamic efficiency did not significantly increase with third-order polynomials found in Fig. (5.17). However, the third-order design space demonstrates that adding complexity to the leading edge shape provides new flexibility in the design space for stable regions, like as shown in Fig. (5.18). As such, increasing the polynomial order may provide resiliency for stability requirements.

The limited Mach 25 study provides very different stability regions than heretofore reported. Generating a waverider with the parameters shown by the yellow dot in Fig. (5.17),

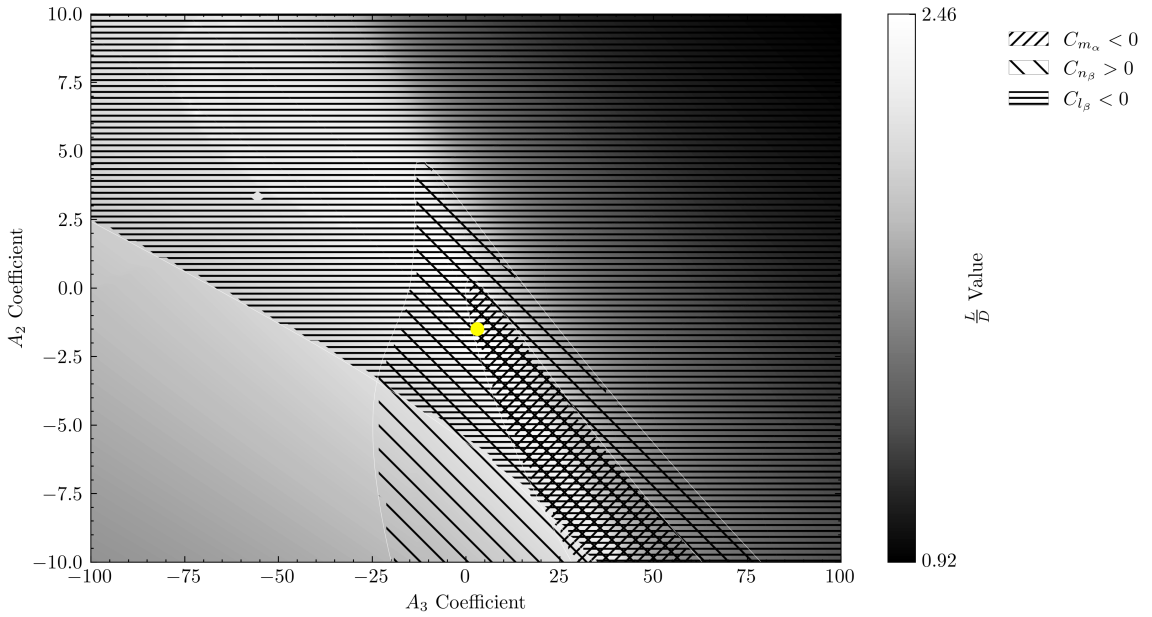


Fig. 5.17: Third-Order Design Space Exploration with Stability Overlays at Mach 25 with a y-intercept of -0.1

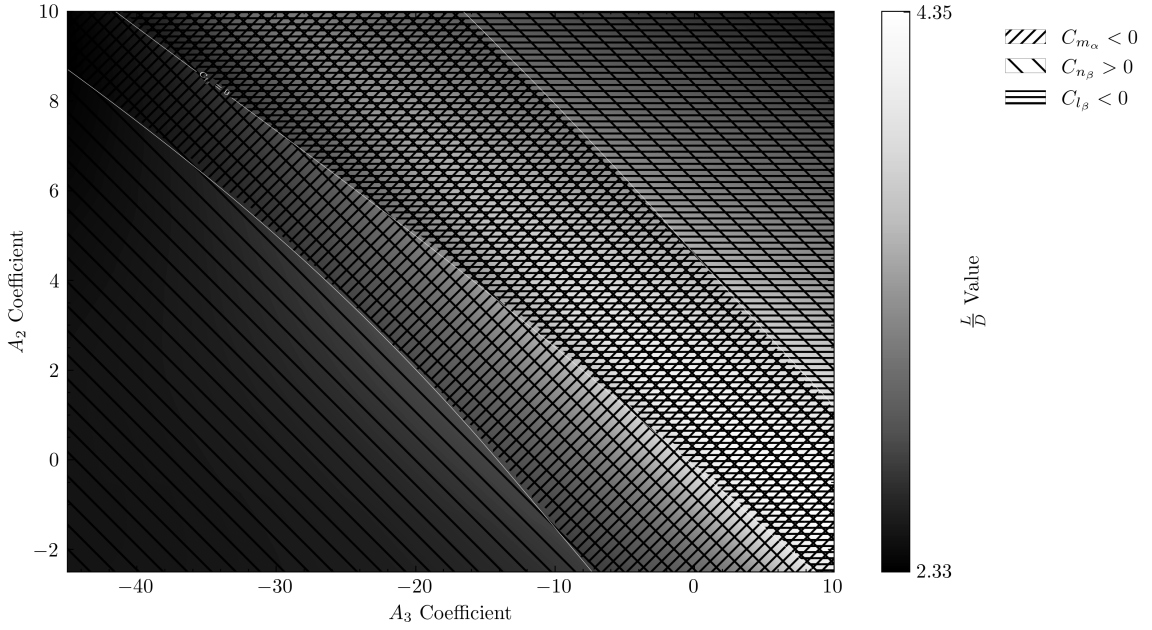


Fig. 5.18: Third-Order Design Space Exploration with Stability Overlays at Mach 4 with a y-intercept of -0.1

a stable, high-efficiency waverider is shown in Fig. (5.19). This near-optimal waverider exhibits slightly upturned edges, reminiscent of vertical stabilizers.

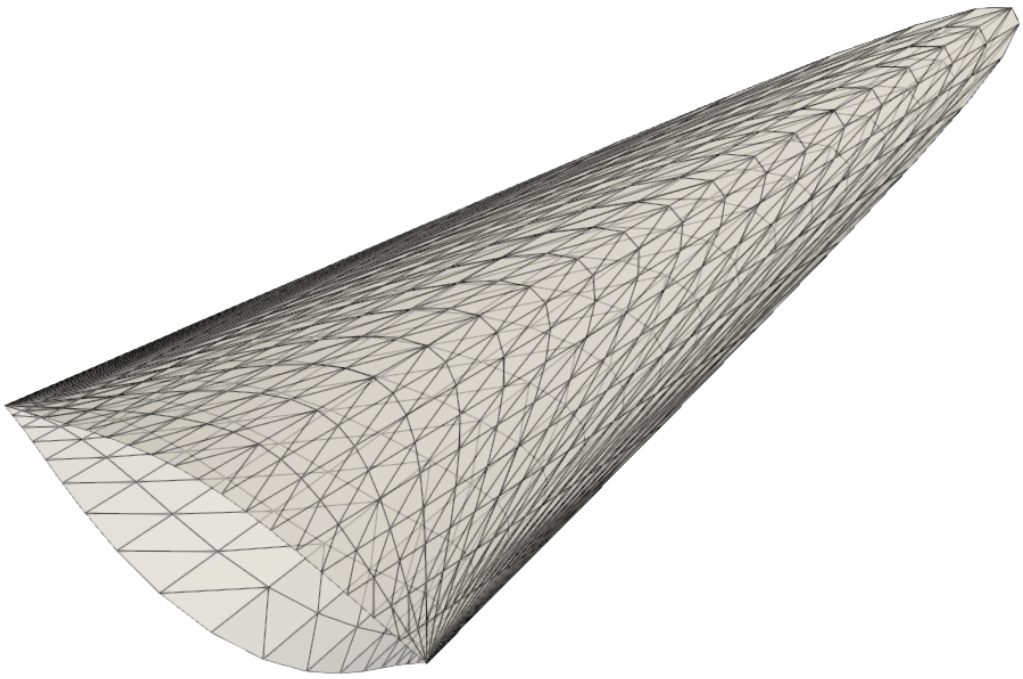


Fig. 5.19: Near-Optimal, Stable Waverider at Mach 25

CHAPTER 6

CONCLUSION

Waveriders exhibit outstanding potential to make hypersonic travel more effective and efficient. However, a successful flight must also meet stability and control requirements. Investigating waverider design spaces via polynomial leading-edge parameterization uncovered relationships between aerodynamic stability and efficiency at cruise conditions.

This research unveiled important information about preliminary design for conically-derived waveriders:

1. Investigating stability in preliminary design is important because waverider design space, has potential for blending stability and aerodynamic efficiency, potentiality simplifying and enhancing the design and analysis of waverider concepts

2. Additional leading-edge shape complexity only makes nominal differences in waverider efficiency. Flat polynomials were found to be the best for aerodynamic efficiency, independent of Mach number. However, waverider stability is more sensitive to changes in leading-edge parameters

3. The shock angle at which a conically-derived waverider is derived from can have significant effects on waverider efficiency and stability

4. The y-intercept of the polynomial projection is highly important to waverider stability

There are various ways to refine and build on this research. In addition to stability requirements, waverider vehicles must meet take-off and landing requirements and volumetric constraints in order to perform a successful mission. These requirements can also have significant effects on waverider design space, complimenting the present research. In addition to traditional conically-derived waveriders, design space exploration and stability requirements could be considered for various types of waverider vehicles. Shock considerations, as discussed above, have a large effect on waverider aerodynamic performance.

This thesis and associated SHADOW tool can be used as a preliminary design tool to get 'back-of-envelope' estimates on desired waverider shapes. Hopefully, this research can serve as a starting point for further waverider design space investigation and visualization. Understanding these design spaces and associated aerodynamic information will help pave the way for efficient hypersonic travel.

REFERENCES

[1] K. G. Bowcutt, "Optimization of hypersonic waveriders dervied from cone flows - including viscous effects," PhD Dissertation, University of Maryland, College Park, MA, July 1986.

[2] J. L. Sims, *Tables for supersonic flow around right circular cones at zero angle of attack*. Office of Scientific and Technical Information, National Aeronautics and . . . , 1964, vol. 3004.

[3] A. Grantz, "Calibration of aerodynamic engineering methods for waverider design," in *32nd Aerospace Sciences Meeting and Exhibit*, 1994, p. 382.

[4] M. Albano, D. Micheli, G. Gradoni, R. B. Morles, M. Marchetti, F. Moglie, and V. M. Primiani, "Electromagnetic shielding of thermal protection system for hypersonic vehicles," *Acta Astronautica*, vol. 87, pp. 30–39, 2013.

[5] E. R. Bartusiak, M. A. Jacobs, M. W. Chan, M. L. Comer, and E. J. Delp, "Predicting hypersonic glide vehicle behavior with stochastic grammars," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 60, no. 1, pp. 1208–1223, 2023.

[6] K.-Y. Hwang and H. Huh, "Research and development trends of a hypersonic glide vehicle (hgv)," *Journal of the Korean Society for Aeronautical & Space Sciences*, vol. 48, no. 9, pp. 731–743, 2020.

[7] P. E. Rodi, "Evaluation of the capsule/waverider concept for mars entry, descent, and landing," in *AIAA AVIATION 2021 FORUM*, 2021, p. 2508.

[8] T. Nonweiler, "Aerodynamic problems of manned space vehicles," *The Aeronautical Journal*, vol. 63, no. 585, pp. 521–528, 1959.

[9] D. A. Lunan, "Waverider, a revised chronology," in *20th AIAA International Space Planes and Hypersonic Systems and Technologies Conference*, 2015, p. 3529.

[10] G. Maikapar, "On the wave drag of non anixymmetric bodies at supersonic speeds," *Journal of Applied Mathematics and Mechanics*, vol. 23, no. 2, pp. 528–531, 1959.

[11] K. D. Jones, H. Sobieczky, A. R. Seebass, and F. C. Dougherty, "Waverider design for generalized shock geometries," *Journal of Spacecraft and Rockets*, vol. 32, no. 6, pp. 957–963, 1995.

[12] N. Takashima and M. Lewis, "Waverider configurations based on non-axisymmetric flow fields for engine-airframe integration," in *32nd Aerospace Sciences Meeting and Exhibit*, 1994, p. 380.

[13] P. E. Rodi, "Improved waverider vehicle optimization with volumetric and lift constraints for shaped sonic booms," in *AIAA SCITECH 2023 Forum*, 2023, p. 2100.

[14] K. Bowcutt, G. Kuruvila, T. Grandine, and E. Cramer, "Advancements in multidisciplinary design optimization applied to hypersonic vehicles to achieve performance closure," in *15th AIAA international space planes and hypersonic systems and technologies conference*, 2008, p. 2591.

[15] P. Rodi and D. Genovesi, "Engineering-based performance comparisons between osculating cone and osculating flowfield waveriders," in *37th AIAA Fluid Dynamics Conference and Exhibit*, 2007, p. 4344.

[16] S. Corda and J. ANDERSON, JR, "Viscous optimized hypersonic waveriders designed from axisymmetric flow fields," in *26th aerospace sciences meeting*, 1988, p. 369.

[17] J. R. Maxwell, "Shapeable hypersonic waverider entry vehicles," in *53rd AIAA/SAE/ASEE Joint Propulsion Conference*, 2017, p. 4880.

[18] N. Takashima and M. Lewis, "Engine-airframe integration on osculating cone waverider-based vehicle designs," in *32nd Joint Propulsion Conference and Exhibit*, 1996, p. 2551.

[19] D. Küchemann, *The aerodynamic design of aircraft*. American Institute of Aeronautics and Astronautics, Inc., 2012.

[20] K. G. Bowcutt, J. D. Anderson, and D. Capriotti, "Viscous optimized hypersonic waveriders," in *25th AIAA Aerospace Sciences Meeting*, 1987, p. 272.

[21] C. E. Cockrell Jr, L. D. Huebner, and D. B. Finley, "Aerodynamic characteristics of two waverider-derived hypersonic cruise configurations," NASA, Tech. Rep., 1996.

[22] H. Etoh, N. Tsuboi, Y. Maru, and K. Fujita, "Aerodynamic characteristics of simplified waveriders," *TRANSACTIONS OF THE JAPAN SOCIETY FOR AERONAUTICAL AND SPACE SCIENCES, AEROSPACE TECHNOLOGY JAPAN*, vol. 12, no. ists29, pp. Pg_25–Pg_31, 2014.

[23] M. A. Lobbia, "Multidisciplinary design optimization of waverider-derived crew reentry vehicles," *Journal of Spacecraft and Rockets*, vol. 54, no. 1, pp. 233–245, 2017.

[24] T. Bykerk, D. Verstraete, and J. Steelant, "Low speed longitudinal aerodynamic, static stability and performance analysis of a hypersonic waverider," *Aerospace Science and Technology*, vol. 96, p. 105531, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1270963819323387

[25] R. Miller and B. Argrow, "Subsonic aerodynamics of an osculating cones waverider," in *35th Aerospace Sciences Meeting and Exhibit*, 1997, p. 189.

[26] A. P. Pisano and C. A. Whitfield, "Characterization of a hypersonic waverider in low-speed flight," in *AIAA SCITECH 2024 Forum*, 2024, p. 2330.

[27] C. Liu, R. Liu, X. Meng, and P. Bai, "Experimental investigation on off-design performances of double-swept waverider," *AIAA Journal*, vol. 61, no. 4, pp. 1596–1607, 2023.

[28] R. P. Starkey and M. J. Lewis, "Critical design issues for airbreathing hypersonic waverider missiles," *Journal of Spacecraft and Rockets*, vol. 38, no. 4, pp. 510–519, 2001.

[29] C. Tarpley and M. Lewis, "Sensitivity of engine-integrated waverider performance to static margin constraint," in *International Aerospace Planes and Hypersonics Technologies*, 1995, p. 6142.

[30] D. E. Hahne, "Evaluation of the low-speed stability and control characteristics of a mach 5.5 waverider concept," NASA, Tech. Rep., 1997.

[31] D. Strohmeyer, "Lateral stability derivatives for osculating cones waveriders in sub-and transonic flow," in *8th AIAA International Space Planes and Hypersonic Systems and Technologies Conference*, 1998, p. 1618.

[32] W. Liu, C.-A. Zhang, X.-P. Wang, J.-J. Li, and F.-M. Wang, "Parametric study on lateral–directional stability of hypersonic waverider," *AIAA Journal*, vol. 59, no. 8, pp. 3025–3042, 2021.

[33] L. Chen, Z. Guo, X. Deng, Z. Hou, and W. Wang, "Waverider configuration design with variable shock angle," *IEEE Access*, vol. 7, pp. 42 081–42 093, 2019.

[34] T. R. Moes and K. Iliff, "Stability and control estimation flight test results for the sr-71 aircraft with externally mounted experiments," NASA, Dryden Flight Research Center, Edwards CA, USA, Technical Publication NASA/TP-2002-210718, June 2002.

[35] E. J. Saltzman and D. J. Garringer, "Summary of full-scale lift and drag characteristics of the x-15 airplane," NASA, Flight Research Center, Edwards CA, USA, Technical Note NASA/TN D-3343, March 1966.

[36] C. H. Wolowicz and R. B. Yancey, "Summary of stability and control characteristics of the xb-70 airplane," NASA, Dryden Flight Research Center, Edwards CA, USA, Technical Memorandum NASA/TM X-2933, June 2002.

[37] J. Anderson, *Modern Compressible Flow: With Historical Perspective*, ser. Aeronautical and Aerospace Engineering Series. McGraw-Hill Education, 2003. [Online]. Available: https://books.google.com/books?id=woeqa4-a5EgC

[38] M. D. Van Dyke, "A study of hypersonic small-disturbance theory," NASA, Tech. Rep., 1954.

[39] H.-S. Tsien, "Similarity laws of hypersonic flows," *Journal of Mathematics and Physics*, vol. 25, no. 1-4, pp. 247–251, 1946. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sapm1946251247

[40] D. J. Harney and A. F. F. D. L. W.-P. A. O. W.-P. AFB, *A Simple Aerodynamic Rule for Hypersonic Small Disturbance Flows*. Air Force Flight Dynamics Laboratory, Air Force Systems Command, Wright ..., 1968.

[41] G. I. Taylor and J. Maccoll, "The air pressure on a cone moving at high speeds.—ii," *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, vol. 139, no. 838, pp. 298–311, 1933.

[42] v. L. Crocco, "Eine neue stromfunktion für die erforschung der bewegung der gase mit rotation." *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, vol. 17, no. 1, pp. 1–7, 1937.

[43] W. Kutta, *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. Teubner, 1901.

[44] H. Liepmann and A. Roshko, *Elements of Gasdynamics*, ser. Dover Books on Aeronautical Engineering Series. Dover Publications, 2001. [Online]. Available: https://books.google.com/books?id=6zcolViQHIYC

[45] M. A. Lobbia and K. Suzuki, "Experimental investigation of a mach 3.5 waverider designed using computational fluid dynamics," *AIAA Journal*, vol. 53, no. 6, pp. 1590–1601, 2015.

[46] V. T. Ramamoorthy, E. Özcan, A. J. Parkes, L. Jaouen, and F.-X. Bécot, "Multi-objective topology optimisation for acoustic porous materials using gradient-based, gradient-free, and hybrid strategies," *The Journal of the Acoustical Society of America*, vol. 153, no. 5, pp. 2945–2955, 05 2023. [Online]. Available: https://doi.org/10.1121/10.0019455

[47] S. Obayashi and T. Tsukahara, "Comparison of optimization algorithms for aerodynamic shape design," *AIAA journal*, vol. 35, no. 8, pp. 1413–1415, 1997.

[48] P. E. Rodi, "On using upper surface shaping to improve waverider performance," in *2018 AIAA Aerospace Sciences Meeting*, 2018, p. 0554.

[49] C. Liu, J. Tian, P. Bai, and Q. Shang, "Effect of upper surface shape on waverider performances," *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, vol. 236, no. 6, pp. 1239–1250, 2022.

[50] M. J. Aftosmis, J. E. Melton, and M. J. Berger, "Software for automated generation of cartesian meshes," NASA, Tech. Rep., 2006.

[51] D. Kinney, "Aero-thermodynamics for conceptual design," in *42nd AIAA Aerospace Sciences Meeting and Exhibit*, 2004, p. 31.

[52] J. K. Goates, L. B. Freeman, and D. F. Hunsaker, "Theory and implementation of a rapid hypersonic impact mehtod (hi-mach) part i: Aerodynamics," in *AIAA Scitech 2025 Forum*. AIAA, 2025.

[53] C. Bliamis, C. Menelaou, and K. Yakinthos, "Implementation of various-fidelity methods for viscous effects modeling on the design of a waverider," *Aerospace Science and Technology*, vol. 133, p. 108141, 2023.

[54] D. Igra, "Extension of the simple analytical model for waverider design," *Journal of Spacecraft and Rockets*, vol. 55, no. 4, pp. 1024–1027, 2018.

[55] R. P. Starkey and M. J. Lewis, "Analytical off-design lift-to-drag-ratio analysis for hypersonic waveriders," *Journal of Spacecraft and Rockets*, vol. 37, no. 5, pp. 684–691, 2000.

[56] J. L. Pittman and G. D. Riebe, "Experimental and theoretical aerodynamic characteristics of two hypersonic cruise aircraft concepts at mach numbers of 2.96, 3.96, and 4.63," NASA, Tech. Rep., 1980.

[57] M. A. Lobbia, "Rapid supersonic/hypersonic aerodynamics analysis model for arbitrary geometries," *Journal of Spacecraft and Rockets*, vol. 54, no. 1, pp. 315–322, 2017.

[58] A. E. Gentry, D. N. Smyth, and M. D. C. L. B. C. D. A. DIV, "Hypersonic arbitrary-body aerodynamic computer program mark iii version. volume 2. program formulation and listings. no," DAC-61552-VOL-2. MCDONNELL DOUGLAS CORP LONG BEACH CA DOUGLAS AIRCRAFT DIV, Tech. Rep., 1968.

[59] D. Kinney, "Aerothermal anchoring of cbaero using high fidelity cfd," in *45th AIAA Aerospace Sciences Meeting and Exhibit*, 2007, p. 608.

[60] J. R. Maxwell, "Efficient design of viscous waveriders with cfd verification and off-design performance analysis," in *53rd AIAA/SAE/ASEE Joint Propulsion Conference*, 2017, p. 4879.

[61] J. ANDERSON, JR, J. Chang, and T. MCLAUGHLIN, "Hypersonic waveriders-effects of chemically reacting flow and viscous interaction," in *30th Aerospace Sciences Meeting and Exhibit*, 1992, p. 302.

[62] W. E. Meador and M. K. Smart, "Reference enthalpy method developed from solutions of the boundary-layer equations," *AIAA journal*, vol. 43, no. 1, pp. 135–139, 2005.

[63] E. Eckert, "Engineering relations for heat transfer and friction in high-velocity laminar and turbulent boundary-layer flow over surfaces with constant pressure and temperature," *Transactions of the American Society of Mechanical Engineers*, vol. 78, no. 6, pp. 1273–1283, 1956.

[64] F. M. White and J. Majdalani, *Viscous fluid flow*. McGraw-Hill New York, 2006, vol. 3.

[65] A. WHITEHEAD, JR, "Nasp aerodynamics," in *National Aerospace Plane Conference*, 1989, p. 5013.

[66] M. J. Lewis and A. D. McRonald, "Design of hypersonic waveriders for aeroassisted interplanetary trajectories," *Journal of Spacecraft and Rockets*, vol. 29, no. 5, pp. 653–660, 1992.

[67] Z. Hao, C. Yan, Y. Qin, and L. Zhou, "Improved γ-reθt model for heat transfer prediction of hypersonic boundary layer transition," *International Journal of Heat and Mass Transfer*, vol. 107, pp. 329–338, 2017.

[68] S. Jain, N. Sitaram, and S. Krishnaswamy, "Effect of reynolds number on aerodynamics of airfoil with gurney flap," *International Journal of Rotating Machinery*, vol. 2015, no. 1, p. 628632, 2015.

[69] W. H. Heiser and D. T. Pratt, *Hypersonic airbreathing propulsion.* Aiaa, 1994.

APPENDICES

My code development was greatly influenced and helped by generative AI, and Jeremy Goates and Logan Freeman (who wrote portions of my HI-Mach usage code, in addition to their development of HI-Mach). The code in these Appendices, while not all used, could be helpful for curious readers.

APPENDIX A

GEOMETRY DEVELOPMENT

```python

"""
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>

                                     DEVELOPED BY:
        ADAM WEAVER | Graduate Researcher at USU AeroLab | adamweaver2000@gmail.com
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
"""

import math
import numpy as np
import matplotlib.pyplot as plt
import os
import tkinter
from matplotlib.backends.backend_tkagg import (
    FigureCanvasTkAgg, NavigationToolbar2Tk)
# Implement the default Matplotlib key bindings.
from matplotlib.backend_bases import key_press_handler
from matplotlib.figure import Figure
import numpy as np
import scipy

# Helper Functions ---------------------------------------------------------------------


def fit_poly(points, order):
    x = [p[0] for p in points]
    y = [p[1] for p in points]
    return np.polyfit(x, y, order)

def get_leadingedge_from_plot():
    points = []
    root = tkinter.Tk()
    root.wm_title("Embedding in Tk")

    fig = Figure(figsize=(5, 4), dpi=100)
    t = np.arange(0, 3, .01)
    ax = fig.add_subplot(111)
    # Add axis limits
    ax.set_xlim(0, 0.5)
    ax.set_ylim(-1, 0)

    canvas = FigureCanvasTkAgg(fig, master=root)  # A tk.DrawingArea.
    canvas.draw()
    canvas.get_tk_widget().pack(side=tkinter.TOP, fill=tkinter.BOTH, expand=1)

    toolbar = NavigationToolbar2Tk(canvas, root)
    toolbar.update()
    canvas.get_tk_widget().pack(side=tkinter.TOP, fill=tkinter.BOTH, expand=1)


    def on_button_press(event):
        print("You pressed:", event.xdata, event.ydata)
```

```
54          if event.inaxes is not None:
55              points.append((event.xdata, event.ydata))
56              ax = event.inaxes
57              ax.plot(event.xdata, event.ydata, 'ro')
58              #ax.plot(event.xdata, event.ydata, 'ro')
59              canvas.draw()
60          key_press_handler(event, canvas, toolbar)
61
62
63      canvas.mpl_connect("button_press_event", on_button_press)
64
65
66      def _quit():
67          root.quit()     # stops mainloop
68          root.destroy()  # this is necessary on Windows to prevent
69                          # Fatal Python Error: PyEval_RestoreThread: NULL tstate
70
71
72      button = tkinter.Button(master=root, text="Quit", command=_quit)
73      button.pack(side=tkinter.BOTTOM)
74
75      tkinter.mainloop()
76
77      polyfit = fit_poly(points, 2)
78
79      return polyfit
80
81  def rk4_array(theta, y0, func, dtheta):
82
83          half_step = dtheta/2
84
85          k1 = []
86          k2 = []
87          k3 = []
88          k4 = []
89
90          k1 = func(y0[1],                          y0[0],                          theta)
91          k2 = func(y0[1] - (dtheta*(k1[1]/2)), y0[0] - (dtheta*((k1[0]/2))), theta-
    half_step)
92          k3 = func(y0[1] - (dtheta*(k2[1]/2)), y0[0] - (dtheta*((k2[0]/2))), theta-
    half_step)
93          k4 = func(y0[1] - (dtheta*(k3[1])),   y0[0] - (dtheta*((k3[0]))),   theta-
    dtheta)
94
95          ans = [0,0]
96          ans[0] = y0[0] - ((k1[0] + k2[0]*2 + k3[0]*2 + k4[0])*(dtheta/6))
97          ans[1] = y0[1] - ((k1[1] + k2[1]*2 + k3[1]*2 + k4[1])*(dtheta/6))
98
99          return ans
100
101 def rk4_streamline(y0, func, dt):
102
103         k1 = []
104         k2 = []
105         k3 = []
106         k4 = []
107
108         k1 = func(y0[1],                  y0[0])
109         k2 = func(y0[1] + (dt*(k1[1]/2)), y0[0] + (dt*((k1[0]/2))))
110         k3 = func(y0[1] + (dt*(k2[1]/2)), y0[0] + (dt*((k2[0]/2))))
111         k4 = func(y0[1] + (dt*(k3[1])),   y0[0] + (dt*((k3[0]))))
112
113         ans = [0,0]
114         ans[0] = y0[0] + ((k1[0] + k2[0]*2 + k3[0]*2 + k4[0])*(dt/6))
115         ans[1] = y0[1] + ((k1[1] + k2[1]*2 + k3[1]*2 + k4[1])*(dt/6))
```

```
116
117         return ans
118
119 def interpolate(pair1, pair2, X):
120
121     Y10 = pair1[1]
122     Y20 = pair2[1]
123     X10 = pair1[0]
124     X20 = pair2[0]
125
126     Y11 = pair1[2]
127     Y21 = pair2[2]
128     X11 = X10
129     X21 = X20
130
131     Y0 = Y10+(((Y20-Y10)/(X20-X10))*(X-X10))
132     Y1 = Y11+(((Y21-Y11)/(X21-X11))*(X-X11))
133
134     return Y0, Y1
135
136 def sphere_to_cart(r, theta, phi):
137
138
139     st = math.sin(theta)
140     ct = math.cos(theta)
141     sp = math.sin(phi)
142     cp = math.cos(phi)
143
144     x = r*cp*st
145     y = r*sp*st
146     z = r*ct
147
148     return [x, y, z]
149
150 def cart_to_sphere(x, y, z):
151
152     r = math.sqrt(x**2+y**2+z**2)
153     if x == 0 and y == 0 and z == 0:
154         theta = 0
155     else:
156         theta = math.acos(((z)/(math.sqrt(x**2+y**2+z**2))))
157     phi = math.atan2(y, x)
158
159     return [r, theta, phi]
160
161 def triangulate(point_cloud, filename):
162
163     import trimesh
164     mesh = trimesh.Trimesh(point_cloud)
165     mesh.export(str(filename)+'.tri')
166
167 def find_closest_point(input_list, desired_point):
168
169     input_list = input_list[input_list[:, 1].argsort()]
170     differences = [abs(arr[0] - desired_point) for arr in input_list]
171
172     # Find the index of the minimum absolute difference
173     index = differences.index(min(differences))
174
175     if index + 1 == len(input_list):
176         index_point1 = index
177         index_point2 = index - 1
178     else:
179
180         if input_list[index][0] < desired_point:
```

```
            index_point1 = index
            index_point2 = index + 1
        else:
            index_point1 = index
            index_point2 = index - 1


    return [[input_list[index_point1][0], input_list[index_point1][1], input_list[index_point1][2]], [input_list[index_point2][0], input_list[index_point2][1], input_list[index_point2][2]]]

def plot_3d(upper_surface, lower_surface, base_plate):

    data = np.concatenate((upper_surface, lower_surface))

    datab = base_plate

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    ax.set_xlim(-3,3)
    ax.set_ylim(-3,3)
    ax.set_zlim(-3,3)

    for i in range(len(data)):
        color = np.random.rand(3,)
        for j in range(len(data[i])-1):
            pairx = [data[i][j][0], data[i][j+1][0]]
            pairy = [data[i][j][1], data[i][j+1][1]]
            pairz = [data[i][j][2], data[i][j+1][2]]
            ax.plot(pairx, pairy,zs=pairz, color=color)

    for i in range(len(datab)):
        color = np.random.rand(3,)
        for j in range(len(datab[i])-1):
            pairx = [datab[i][j][0], datab[i][j+1][0]]
            pairy = [datab[i][j][1], datab[i][j+1][1]]
            pairz = [datab[i][j][2], datab[i][j+1][2]]
            ax.plot(pairx, pairy,zs=pairz, color=color)

    plt.show(block=False)

def get_z_point(xylist, theta):

    zlist = []
    for i in range(len(xylist)):
        x = xylist[i][0]
        y = xylist[i][1]
        z = math.sqrt(((((x**2)*(math.cos(theta))**2))+((y**2)*(math.cos(theta))**2))/(1-(math.cos(theta))**2))
        zlist.append(z)

    return zlist

def surface_mesh(top_surface, lower_surface):

    leadingedge = []
    top_surface_mesh = []
    for i in range(len(top_surface) - 1):
        for j in range(len(top_surface[i]) - 1):
            if i == 0:
                top_surface_mesh.append([0, j+1, j+2])
```

```
243                 if j == 0:
244                     leadingedge.append(0)
245 
246             elif i != 0 and i != len(top_surface) - 2:
247                 top_surface_mesh.append([(len(top_surface[0])*(i-1))+1+j, ((len(
top_surface[0]))*(i))+1+j, (len(top_surface[0])*(i))+2+j])
248                 top_surface_mesh.append([(len(top_surface[0])*(i-1))+1+j, (len(
top_surface[0])*(i-1))+2+j, (len(top_surface[0])*(i))+2+j])
249                 if j == 0:
250                     leadingedge.append((len(top_surface[0])*(i-1))+1+j)
251                     if i == len(top_surface) - 3:
252                         leadingedge.append((len(top_surface[0])*(i))+1+j)
253 
254             elif i == len(top_surface) - 2:
255                 top_surface_mesh.append([(len(top_surface[0])*(i))+1, (len(
top_surface[0])*(i-1))+1+j, (len(top_surface[0])*(i-1))+2+j])
256                 if j == 0:
257                     leadingedge.append((len(top_surface[0])*(i))+1)
258 
259     lower_surface_mesh = []
260     trailingedge = []
261     last_index = (len(top_surface[0])*(i))+1
262     for i in range(len(lower_surface) - 1):
263         for j in range(len(lower_surface[i]) - 2):
264             if i == 0:
265                 lower_surface_mesh.append([0+last_index, j+1+last_index, j+2+
last_index])
266                 if j == len(lower_surface[i]) - 3:
267                     lower_surface_mesh.append([0+last_index, j+2+last_index,
last_index-1])
268                 if j == 0:
269                     trailingedge.append(0+last_index)
270 
271             elif i != 0 and i != len(lower_surface) - 2:
272                 lower_surface_mesh.append([(len(lower_surface[0])*(i-1))+(3-i)+j+
last_index-1, (len(lower_surface[0])*(i))+(3-i)+j+last_index-1-1, (len(lower_surface
[0])*(i))+(3-i)+j+last_index-1])
273                 lower_surface_mesh.append([(len(lower_surface[0])*(i-1))+(3-i)+j+
last_index-1, (len(lower_surface[0])*(i-1))+(3-i)+j+1+last_index-1, (len(
lower_surface[0])*(i))+(3-i)+j+last_index-1])
274                 if j == len(lower_surface[i]) - 3:
275                     lower_surface_mesh.append([(len(lower_surface[0])*(i-1))+(3-i)+j+
last_index, (len(lower_surface[0])*(i))+(3-i)+j+last_index-1, (len(top_surface[i])*((
len(top_surface)-2)-i))])
276                     lower_surface_mesh.append([(len(lower_surface[0])*(i-1))+(3-i)+j+
last_index, (len(top_surface[i])*((len(top_surface)-1)-i)), (len(top_surface[i])*((
len(top_surface)-2)-i))])
277                 if j == 0:
278                     trailingedge.append((len(lower_surface[0])*(i-1))+(3-i)+j+
last_index-1)
279                     if i == len(lower_surface) - 3:
280                         trailingedge.append((len(lower_surface[0])*(i))+(3-i)+j+
last_index-2)
281 
282             elif i == len(lower_surface) - 2:
283                 lower_surface_mesh.append([0, last_index+1+(len(lower_surface[i])-1)
*(len(lower_surface)-3)+1+j-1, last_index+1+(len(lower_surface[i])-1)*(len(
lower_surface)-3)+1+j+1-1])
284                 if j == len(lower_surface[i]) - 3:
285                     lower_surface_mesh.append([0, last_index+1+(len(lower_surface[i])
-1)*(len(lower_surface)-3)+1+j, len(top_surface[i])])
286                 if j == 0:
287                     trailingedge.append(0)
288 
289     base_mesh = []
```

```
290
291     trailingedge.pop(0)
292     trailingedge.pop(-1)
293
294     trailingedge = [trailingedge[i] for i in range(len(trailingedge)-1, -1, -1)]
295
296     for i in range(len(trailingedge)-1):
297         base_mesh.append([trailingedge[i], leadingedge[i+1], leadingedge[i+2]])
298         base_mesh.append([trailingedge[i], trailingedge[i+1], leadingedge[i+2]])
299
300     base_mesh.append([leadingedge[0], leadingedge[1], trailingedge[0]])
301     base_mesh.append([leadingedge[-1], leadingedge[-2], trailingedge[-1]])
302
303     triangle_list = top_surface_mesh + lower_surface_mesh + base_mesh
304
305     return triangle_list
306
307 def mesh(top_surface, lower_surface):
308
309     leadingedge = []
310     top_surface_mesh = []
311     for i in range(len(top_surface) - 1):
312         for j in range(len(top_surface[i]) - 1):
313             if i == 0:
314                 top_surface_mesh.append([0, j+1, j+2])
315                 if j == 0:
316                     leadingedge.append(0)
317
318             elif i != 0 and i != len(top_surface) - 2:
319                 top_surface_mesh.append([(len(top_surface[0])*(i-1))+1+j, ((len(
top_surface[0]))*(i))+1+j, (len(top_surface[0])*(i))+2+j])
320                 top_surface_mesh.append([(len(top_surface[0])*(i))+2+j, (len(
top_surface[0])*(i-1))+2+j, (len(top_surface[0])*(i-1))+1+j])
321                 if j == 0:
322                     leadingedge.append((len(top_surface[0])*(i-1))+1+j)
323                     if i == len(top_surface) - 3:
324                         leadingedge.append((len(top_surface[0])*(i))+1+j)
325
326             elif i == len(top_surface) - 2:
327                 top_surface_mesh.append([(len(top_surface[0])*(i-1))+2+j,(len(
top_surface[0])*(i-1))+1+j, (len(top_surface[0])*(i))+1])
328                 if j == 0:
329                     leadingedge.append((len(top_surface[0])*(i))+1)
330
331     lower_surface_mesh = []
332     trailingedge = []
333     last_index = (len(top_surface[0])*(i))+1
334     for i in range(len(lower_surface) - 1):
335         for j in range(len(lower_surface[i]) - 2):
336             if i == 0:
337                 lower_surface_mesh.append([0+last_index, j+1+last_index, j+2+
last_index])
338                 if j == len(lower_surface[i]) - 3:
339                     lower_surface_mesh.append([0+last_index, j+2+last_index,
last_index-1])
340                 if j == 0:
341                     trailingedge.append(0+last_index)
342
343             elif i != 0 and i != len(lower_surface) - 2:
344                 lower_surface_mesh.append([(len(lower_surface[0])*(i-1))+(3-i)+j+
last_index-1, (len(lower_surface[0])*(i))+(3-i)+j+last_index-1-1, (len(lower_surface
[0])*(i))+(3-i)+j+last_index-1])
345                 lower_surface_mesh.append([(len(lower_surface[0])*(i))+(3-i)+j+
last_index-1,(len(lower_surface[0])*(i-1))+(3-i)+j+1+last_index-1, (len(lower_surface
[0])*(i-1))+(3-i)+j+last_index-1])
```

```
346                 if j == len(lower_surface[i]) - 3:
347                     lower_surface_mesh.append([(len(lower_surface[0])*(i-1))+(3-i)+j+
last_index, (len(lower_surface[0])*(i))+(3-i)+j+last_index-1, (len(top_surface[i])*((
len(top_surface)-2)-i))])
348                     lower_surface_mesh.append([(len(top_surface[i])*((len(top_surface
)-2)-i)), (len(top_surface[i])*((len(top_surface)-1)-i)), (len(lower_surface[0])*(i
-1))+(3-i)+j+last_index])
349                 if j == 0:
350                     trailingedge.append((len(lower_surface[0])*(i-1))+(3-i)+j+
last_index-1)
351                     if i == len(lower_surface) - 3:
352                         trailingedge.append((len(lower_surface[0])*(i))+(3-i)+j+
last_index-2)
353 
354             elif i == len(lower_surface) - 2:
355                 lower_surface_mesh.append([last_index+1+(len(lower_surface[i])-1)*(
len(lower_surface)-3)+1+j+1-1, last_index+1+(len(lower_surface[i])-1)*(len(
lower_surface)-3)+1+j-1, 0])
356                 if j == len(lower_surface[i]) - 3:
357                     lower_surface_mesh.append([len(top_surface[i]), last_index+1+(len
(lower_surface[i])-1)*(len(lower_surface)-3)+1+j, 0])
358                 if j == 0:
359                     trailingedge.append(0)
360 
361     base_mesh = []
362 
363     trailingedge.pop(0)
364     trailingedge.pop(-1)
365 
366     trailingedge = [trailingedge[i] for i in range(len(trailingedge)-1, -1, -1)]
367 
368     for i in range(len(trailingedge)-1):
369         base_mesh.append([leadingedge[i+2], leadingedge[i+1], trailingedge[i]])
370         base_mesh.append([trailingedge[i], trailingedge[i+1], leadingedge[i+2]])
371 
372     base_mesh.append([trailingedge[0], leadingedge[1], leadingedge[0]])
373     base_mesh.append([leadingedge[-1], leadingedge[-2], trailingedge[-1]])
374 
375     # REMOVE BASE LIST!
376     #triangle_list = top_surface_mesh + lower_surface_mesh + base_mesh
377     triangle_list = top_surface_mesh + lower_surface_mesh
378 
379     return triangle_list
380 
381 def mesh_2(top_surface, lower_surface):
382 
383     leadingedge = []
384     top_surface_mesh = []
385     for i in range(len(top_surface) - 1):
386         for j in range(len(top_surface[i]) - 1):
387             if i <= (((len(top_surface)-1)/2)-1):
388                 if i == 0:
389                     top_surface_mesh.append([0, j+1, j+2])
390                     if j == 0:
391                         leadingedge.append(0)
392 
393                 elif i != 0 and i != len(top_surface) - 2:
394                     top_surface_mesh.append([(len(top_surface[0])*(i-1))+1+j, ((len(
top_surface[0]))*(i))+1+j, (len(top_surface[0])*(i))+2+j])
395                     top_surface_mesh.append([(len(top_surface[0])*(i))+2+j, (len(
top_surface[0])*(i-1))+2+j, (len(top_surface[0])*(i-1))+1+j])
396                     if j == 0:
397                         leadingedge.append((len(top_surface[0])*(i-1))+1+j)
398                         if i == len(top_surface) - 3:
399                             leadingedge.append((len(top_surface[0])*(i))+1+j)
```

```
400
401                 elif i == len(top_surface) - 2:
402                     top_surface_mesh.append([(len(top_surface[0])*(i-1))+2+j,(len(
top_surface[0])*(i-1))+1+j, (len(top_surface[0])*(i))+1])
403                     if j == 0:
404                         leadingedge.append((len(top_surface[0])*(i))+1)
405             else:
406                 if i == 0:
407                     top_surface_mesh.append([0, j+1, j+2])
408                     if j == 0:
409                         leadingedge.append(0)
410
411                 elif i != 0 and i != len(top_surface) - 2:
412                     top_surface_mesh.append([((len(top_surface[0]))*(i))+1+j, (len(
top_surface[0])*(i))+2+j, (len(top_surface[0])*(i-1))+2+j])
413                     top_surface_mesh.append([(len(top_surface[0])*(i-1))+2+j, (len(
top_surface[0])*(i-1))+1+j, ((len(top_surface[0]))*(i))+1+j])
414                     if j == 0:
415                         leadingedge.append((len(top_surface[0])*(i-1))+1+j)
416                         if i == len(top_surface) - 3:
417                             leadingedge.append((len(top_surface[0])*(i))+1+j)
418
419                 elif i == len(top_surface) - 2:
420                     top_surface_mesh.append([(len(top_surface[0])*(i-1))+2+j, (len(
top_surface[0])*(i-1))+1+j, (len(top_surface[0])*(i))+1])
421                     if j == 0:
422                         leadingedge.append((len(top_surface[0])*(i))+1)
423
424     lower_surface_mesh = []
425     trailingedge = []
426     last_index = (len(top_surface[0])*(i))+1
427     for i in range(len(lower_surface) - 1):
428         for j in range(len(lower_surface[i]) - 2):
429             if i <= (((len(lower_surface)-1)/2)-1):
430                 if i == 0:
431                     lower_surface_mesh.append([0+last_index, j+1+last_index, j+2+
last_index])
432                     if j == len(lower_surface[i]) - 3:
433                         lower_surface_mesh.append([0+last_index, j+2+last_index,
last_index-1])
434                     if j == 0:
435                         trailingedge.append(0+last_index)
436
437                 elif i != 0 and i != len(lower_surface) - 2:
438                     lower_surface_mesh.append([(len(lower_surface[0])*(i-1))+(3-i)+j+
last_index-1, (len(lower_surface[0])*(i))+(3-i)+j+last_index-1-1, (len(lower_surface
[0])*(i))+(3-i)+j+last_index-1])
439                     lower_surface_mesh.append([(len(lower_surface[0])*(i))+(3-i)+j+
last_index-1,(len(lower_surface[0])*(i-1))+(3-i)+j+1+last_index-1, (len(lower_surface
[0])*(i-1))+(3-i)+j+last_index-1])
440                     if j == len(lower_surface[i]) - 3:
441                         lower_surface_mesh.append([(len(lower_surface[0])*(i-1))+(3-i
)+j+last_index, (len(lower_surface[0])*(i))+(3-i)+j+last_index-1, (len(top_surface[i
])*((len(top_surface)-2)-i))])
442                         lower_surface_mesh.append([(len(top_surface[i])*((len(
top_surface)-2)-i)), (len(top_surface[i])*((len(top_surface)-1)-i)), (len(
lower_surface[0])*(i-1))+(3-i)+j+last_index])
443                     if j == 0:
444                         trailingedge.append((len(lower_surface[0])*(i-1))+(3-i)+j+
last_index-1)
445                         if i == len(lower_surface) - 3:
446                             trailingedge.append((len(lower_surface[0])*(i))+(3-i)+j+
last_index-2)
447
448                 elif i == len(lower_surface) - 2:
```

```
449                     lower_surface_mesh.append([last_index+1+(len(lower_surface[i])-1)*(len(lower_surface)-3)+1+j+1-1, last_index+1+(len(lower_surface[i])-1)*(len(lower_surface)-3)+1+j-1, 0])
450                     if j == len(lower_surface[i]) - 3:
451                         lower_surface_mesh.append([len(top_surface[i]), last_index+1+(len(lower_surface[i])-1)*(len(lower_surface)-3)+1+j, 0])
452                     if j == 0:
453                         trailingedge.append(0)
454             else:
455                 if i == 0:
456                     lower_surface_mesh.append([0+last_index, j+1+last_index, j+2+last_index])
457                     if j == len(lower_surface[i]) - 3:
458                         lower_surface_mesh.append([0+last_index, j+2+last_index, last_index-1])
459                     if j == 0:
460                         trailingedge.append(0+last_index)
461 
462                 elif i != 0 and i != len(lower_surface) - 2:
463                     lower_surface_mesh.append([(len(lower_surface[0])*(i))+(3-i)+j+last_index-1-1, (len(lower_surface[0])*(i))+(3-i)+j+last_index-1, (len(lower_surface[0])*(i-1))+(3-i)+j+1+last_index-1])
464                     lower_surface_mesh.append([(len(lower_surface[0])*(i-1))+(3-i)+j+1+last_index-1, (len(lower_surface[0])*(i-1))+(3-i)+j+last_index-1, (len(lower_surface[0])*(i))+(3-i)+j+last_index-1-1])
465                     if j == len(lower_surface[i]) - 3:
466                         lower_surface_mesh.append([(len(lower_surface[0])*(i))+(3-i)+j+last_index-1, (len(top_surface[i])*((len(top_surface)-2)-i)), (len(top_surface[i])*((len(top_surface)-1)-i))])
467                         lower_surface_mesh.append([(len(top_surface[i])*((len(top_surface)-1)-i)), (len(lower_surface[0])*(i-1))+(3-i)+j+last_index, (len(lower_surface[0])*(i))+(3-i)+j+last_index-1])
468                     if j == 0:
469                         trailingedge.append((len(lower_surface[0])*(i-1))+(3-i)+j+last_index-1)
470                         if i == len(lower_surface) - 3:
471                             trailingedge.append((len(lower_surface[0])*(i))+(3-i)+j+last_index-2)
472 
473                 elif i == len(lower_surface) - 2:
474                     lower_surface_mesh.append([last_index+1+(len(lower_surface[i])-1)*(len(lower_surface)-3)+1+j+1-1, last_index+1+(len(lower_surface[i])-1)*(len(lower_surface)-3)+1+j-1, 0])
475                     if j == len(lower_surface[i]) - 3:
476                         lower_surface_mesh.append([len(top_surface[i]), last_index+1+(len(lower_surface[i])-1)*(len(lower_surface)-3)+1+j, 0])
477                     if j == 0:
478                         trailingedge.append(0)
479 
480     base_mesh = []
481 
482     trailingedge.pop(0)
483     trailingedge.pop(-1)
484 
485     trailingedge = [trailingedge[i] for i in range(len(trailingedge)-1, -1, -1)]
486 
487     for i in range(len(trailingedge)-1):
488         base_mesh.append([leadingedge[i+2], leadingedge[i+1], trailingedge[i]])
489         base_mesh.append([trailingedge[i], trailingedge[i+1], leadingedge[i+2]])
490 
491     base_mesh.append([trailingedge[0], leadingedge[1], leadingedge[0]])
492     base_mesh.append([leadingedge[-1], leadingedge[-2], trailingedge[-1]])
493 
494     # REMOVE BASE LIST!
495     #triangle_list = top_surface_mesh + lower_surface_mesh + base_mesh
```

```
496     triangle_list = top_surface_mesh + lower_surface_mesh
497 
498     return triangle_list
499 
500 
501 def parabolic_leading_edge(a, n, xmin, xmax, z):
502     # Note that a must be less than or equal to 4
503     length = xmax-xmin
504     step = length/n
505     xlist = []
506     ylist = []
507     zlist = []
508     for i in range(n+1):
509         x = (i*step)+xmin
510         xlist.append(x)
511         ylist.append(z)
512         zlist.append((a*x**2)+0.1)
513 
514     leading_edge_points = []
515     for i in range(n+1):
516         leading_edge_points.append([xlist[i], ylist[i], zlist[i]])
517 
518     leading_edge_points = np.array(leading_edge_points)
519 
520     return leading_edge_points
521 
522 def define_leading_edge(ConeAngle):
523 
524         LeadingEdgePointsList = [[-0.24,     -0.21],
525                                  [-0.22,     -0.215],
526                                  [-0.2,  -0.22],
527                                  [-0.18,     -0.225],
528                                  [-0.16,  -0.23],
529                                  [-0.14,     -0.23],
530                                  [-0.12,     -0.225],
531                                  [-0.1,  -0.22],
532                                  [-0.08,     -0.215],
533                                  [-0.06,     -0.21],
534                                  [-0.04,     -0.205],
535                                  [-0.02,     -0.2],
536                                  [0,     -0.195],
537                                  [0.02,  -0.2],
538                                  [0.04,  -0.205],
539                                  [0.06,  -0.21],
540                                  [0.08,  -0.215],
541                                  [0.1,   -0.22],
542                                  [0.12,  -0.225],
543                                  [0.14,  -0.23],
544                                  [0.16,  -0.23],
545                                  [0.18,  -0.225],
546                                  [0.2,   -0.22],
547                                  [0.22,  -0.215],
548                                  [0.24,  -0.21]]
549 
550         z_list = get_z_point(LeadingEdgePointsList, ConeAngle)
551 
552         for xy_point, z_value in zip(LeadingEdgePointsList, z_list):
553             xy_point.append(z_value)
554 
555         LeadingEdgePointsList = np.array(LeadingEdgePointsList)
556 
557         return LeadingEdgePointsList
558 
559 def print_clickable_link(filepath, type="csv"):
560 
```

```
561     if type == "csv":
562         print(f"CSV File (containing Taylor-Maccoll solution) created: {filepath}")
563 
564         print(f"\033]8;;file://{filepath}\033\\Open CSV File\033]8;;\033\\")
565     if type == 'tri':
566         print(f"NASA Cart3D TRI File created: {filepath}")
567 
568         print(f"\033]8;;file://{filepath}\033\\Open TRI File\033]8;;\033\\")
569 
570 def execute():
571     import pyfiglet
572 
573     font = "usaflag"
574 
575     figlet = pyfiglet.Figlet(font=font)
576 
577     text = "SHADOW"
578 
579     ascii_art = figlet.renderText(text)
580     ascii_art = ascii_art.rstrip()
581 
582     print(ascii_art)
583     text1 = "\033[1mS\033[0m-tability for \033[1mH\033[0m-ypersonic \033[1mA\033[0m-
    erodyanimc \033[1mD\033[0m-erivatives via \033[1mO\033[0m-ptimizing \033[1mW\033[0m-
    averiders\033[0m"
584     text2 = "Hypersonic Stability Optimization Code in Python3 developed by ADAM
    WEAVER at Utah State University AeroLab"
585     text3 = "v2.1.01 (c)"
586 
587     print(text1)
588     print(text2)
589     print(text3)
590     print()
591 
592 # End Helper Functions
    ----------------------------------------------------------------
593 
594 # WaveRider Geometry Class
595 
596 class WaveRider:
597 
598     wr_vel_list = 0
599     wr_solved_cone_angle = 0
600     wr_upper_surface = 0
601     wr_lower_surface = 0
602     wr_base_plate = 0
603     wr_m2 = 0
604     wr_deflectionangle = 0
605     wr_mn1 = 0
606     wr_m2n = 0
607     wr_mean_aerodynamic_chord = 0
608     wr_integral = 0
609     wr_start_point = 0
610 
611 
612     def __init__(self, mach_number, cone_angle, lead_edge_quant, surface_quant, gamma
    , theta_step,
613                  stream_step, length, leading_edge_order, polyarg, trifilename,
    plotLeadingEdge):
614 
615         # Shock Properities ------------------------------------
616         self.wr_mach_number = mach_number # Operating Mach Number
617         self.wr_cone_angle = cone_angle # Assumed Shock Angle
618         self.wr_gamma = gamma # Specific Heat Ratio
619         # Mesh Propeties ---------------------------------------
```

```
620         self.wr_lead_edge_quant = int(lead_edge_quant) # How many leading edge points
     do you want?
621         self.wr_surface_quant = int(surface_quant) # How many z cuts should the mesh
    use?
622         self.wr_theta_step_size = theta_step # What is the fineness of the RK4
    integration scheme for the Taylor-Maccoll Integration?
623         self.wr_streamline_step = stream_step # What is the fineness of the RK4
    integration scheme for the streamline tracing?
624         self.wr_z_length = length
625         self.wr_length = length
626         self.wr_leading_edge_order = leading_edge_order
627         self.wr_polyarg = polyarg
628         self.wr_trifilename = trifilename
629         self.wr_plotLeadingEdge = plotLeadingEdge
630
631
632     def find_pressures(self, vertices, triangle):
633         panel_pressures = []
634         for i in range(len(triangle)):
635             triangle1 = [vertices[triangle[i][0]]]
636             triangle2 = [vertices[triangle[i][1]]]
637             triangle3 = [vertices[triangle[i][2]]]
638             x_centroid = (triangle1[0][0] + triangle2[0][0] + triangle3[0][0])/3
639             y_centroid = (triangle1[0][1] + triangle2[0][1] + triangle3[0][1])/3
640             z_centroid = (triangle1[0][2] + triangle2[0][2] + triangle3[0][2])/3
641             triangle_centroid = [x_centroid, y_centroid, z_centroid]
642             x = x_centroid
643             y = y_centroid
644             z = z_centroid
645             theta = math.acos(((z)/(math.sqrt(x**2+y**2+z**2))))
646             pairs = find_closest_point(self.wr_vel_list, theta)
647             output = interpolate(pairs[0], pairs[1], theta)
648             vr = output[0]
649             vtheta = output[1]
650             v = math.sqrt(vr**2 + vtheta**2)
651             M1 = math.sqrt((2/(self.wr_gamma-1))*((v**2)/(1-(v**2))))
652             M = math.sqrt(2/((self.wr_gamma - 1)*(((1/v)**2)-1)))
653             pt2opinfty = ((1+(((self.wr_gamma-1)/2)*self.wr_m2**2))**(self.wr_gamma/(
    self.wr_gamma-1)))*(1+(((2*self.wr_gamma)/(self.wr_gamma+1))*((self.wr_mn1**2)-1)))
654             popinfty = pt2opinfty*((1+(((self.wr_gamma-1)/2)*(M**2)))**(-(self.
    wr_gamma)/(self.wr_gamma-1)))
655             pop2 = (1+(((self.wr_gamma-1)/2)*(self.wr_m2**2)))**(self.wr_gamma/(self.
    wr_gamma-1))
656             #p2pinfty = 1+(((2*self.wr_gamma)/(self.wr_gamma+1))*((self.wr_mn1*(math.
    sin(self.wr_cone_angle)**2))-1))
657             p2pinfty = 1+(((2*self.wr_gamma)/(self.wr_gamma+1))*((self.wr_mn1**2)-1))
658             pipo = 1/((1+(((self.wr_gamma-1)/2)*(M**2)))**(self.wr_gamma/(self.
    wr_gamma-1)))
659             #pipo = (2*self.wr_gamma*M**2*((math.sin(self.wr_cone_angle))**2)-(self.
    wr_gamma - 1))/(self.wr_gamma+1)
660             pipinfty = pop2*p2pinfty*pipo
661             pconepinfty = (((1+(((self.wr_gamma-1)/2)*((self.wr_m2)**2)))/(1+(((self.
    wr_gamma-1)/2)*((M)**2))))**(self.wr_gamma/(self.wr_gamma-1)))*(1+(((2*self.wr_gamma)
    /(self.wr_gamma+1))*(((self.wr_mach_number*math.sin(self.wr_cone_angle))**2)-1)))
662             Cp = (2/(self.wr_gamma*self.wr_mach_number**2))*(pipinfty-1)
663             Cp2 = (2/(self.wr_gamma*self.wr_mach_number**2))*(popinfty-1)
664             Cp3 = (2/(self.wr_gamma*self.wr_mach_number**2))*(pconepinfty-1)
665             panel_pressures.append([i,Cp, Cp2, Cp3])
666
667         return panel_pressures
668
669     def leading_edge(self, *args):
670
671         args = list(args)
672         delta_y = args[-1]
```

```
673         args.insert(-1, 0.00)
674         if args[0] == 0.00:
675             args[0] = 0.0001
676 
677         poly_list = []
678 
679         for i in range(self.wr_leading_edge_order):
680             poly_list.append(args[i])
681 
682         poly_list = args
683 
684         leading_edge_poly = np.poly1d(poly_list)
685 
686         x_0 = 0
687         y_0 = delta_y
688         z_0 = math.sqrt(((((x_0**2)*(math.cos(self.wr_cone_angle))**2))+((y_0**2)*(
math.cos(self.wr_cone_angle))**2))/(1-(math.cos(self.wr_cone_angle))**2))
689 
690         self.wr_start_point = z_0
691         end_point = z_0+1
692 
693         self.wr_z_length = end_point
694         if self.wr_leading_edge_order > 1:
695             coefficients = []
696             y_squared = leading_edge_poly*leading_edge_poly
697 
698             coefficients = y_squared.coeffs
699 
700             coefficients[-3] = coefficients[-3]+1
701 
702             right_hand_side = ((end_point**2)*(1-(math.cos(self.wr_cone_angle))**2))
/((math.cos(self.wr_cone_angle))**2)
703             coefficients[-1] = coefficients[-1] - right_hand_side
704 
705             function = np.poly1d(coefficients)
706 
707             roots = function.roots
708 
709         else:
710 
711             print("SHOULDN'T ENTER HERE")
712 
713         x_end = 0.005
714 
715         for root in roots:
716             if np.imag(root) == 0:
717                 if np.real(root) > 0:
718                     x_end = np.real(root)
719                     break
720 
721 
722         delta_x = x_end/((self.wr_lead_edge_quant-1)/2)
723         no_steps = ((self.wr_lead_edge_quant-1)/2)-1
724 
725         positivexpoints = []
726         for i in range(1,int(no_steps)+1):
727             positivexpoints.append(delta_x*i)
728         negativexpoints = []
729         for item in reversed(positivexpoints):
730             negativexpoints.append(-item)
731 
732         xpoints = [-x_end]+negativexpoints+[0]+positivexpoints+[x_end]
733         ypointshalf = []
734         for i in range(1, int(len(xpoints)/2)+1):
735             ypointshalf.append(leading_edge_poly(xpoints[-i]))
```

```
736
737         ypointsotherhalf = [item for item in reversed(ypointshalf)]
738
739         ypoints = ypointshalf +[delta_y]+ ypointsotherhalf
740
741         leading_edge_geom = np.column_stack((xpoints, ypoints)).tolist()
742
743         z_list = get_z_point(leading_edge_geom, self.wr_cone_angle)
744
745         half_z_list = z_list[:int((len(z_list)/2)+1)]
746         half_x_list = xpoints[:int((len(xpoints)/2)+1)]
747         # Alter half z list so that each point is the length of the z_list minus the
    value of the last element
748         half_z_list = [(item - half_z_list[-1])**2 for item in half_z_list]
749
750         integral = scipy.integrate.simps(half_z_list, half_x_list)
751         self.wr_integral = integral
752
753         for xy_point, z_value in zip(leading_edge_geom, z_list):
754             xy_point.append(z_value)
755
756         leading_edge_geom = np.array(leading_edge_geom)
757
758         self.wr_leading_edge_geom = leading_edge_geom
759
760         xroots = leading_edge_poly.roots
761         smallest_x_root = np.min(xroots)
762
763         if smallest_x_root < 0:
764
765             return x_end-smallest_x_root, -(delta_y+0.0000001)
766
767         return -(x_end-smallest_x_root), -(delta_y+0.0000001) # This has to be
    greater than zero!
768
769     def plot_quiver(self):
770
771         theta = self.wr_vel_list[:, 0]
772         v_r = self.wr_vel_list[:, 1]
773         v_theta = self.wr_vel_list[:, 2]
774
775         num_quivers = 50
776
777         ray_indices = np.arange(0, len(theta), 25)
778
779         theta_grid = np.repeat(theta[ray_indices, np.newaxis], num_quivers, axis=1)
780         v_r_grid = np.repeat(v_r[ray_indices, np.newaxis], num_quivers, axis=1)
781         v_theta_grid = np.repeat(v_theta[ray_indices, np.newaxis], num_quivers, axis
    =1)
782
783         radii = np.linspace(0, 1, num_quivers)
784
785         x = radii * np.cos(theta_grid)
786         y = radii * np.sin(theta_grid)
787         vx = v_r_grid * np.cos(theta_grid) - v_theta_grid * np.sin(theta_grid)
788         vy = v_r_grid * np.sin(theta_grid) + v_theta_grid * np.cos(theta_grid)
789
790         plt.figure()
791         plt.quiver(x, y, vx, vy, angles='xy', scale_units='xy', scale=0.0000000001)
792         plt.xlabel('X')
793         plt.ylabel('Y')
794         plt.title('Vector Field')
795         plt.axis('equal')
796         plt.grid()
797         plt.show(block=False)
```

```
798
799     def shock_test(self):
800
801         """
802         THE FOLLOWING EQUATIONS COME FROM PAGE 135-136 FROM MODERN COMPRESSIBLE FLOW:
 Dr. John D. Anderson
803         """
804         # Upstream Flow Normal to Cone (Mach Number)
805         MN1 = self.wr_mach_number*math.sin(self.wr_cone_angle) # Equation 4.7 from
Anderson
806         # Downstream Flow Nomral to Cone (Mach Number)
807         #MN21 = math.sqrt((MN1**2+(2/(self.wr_gamma-1)))/((2*self.wr_gamma/(self.
wr_gamma-1))*MN1**2)-1)) # Equation 4.10 from Anderson
808         MN2 = math.sqrt((1+((self.wr_gamma-1)/2)*MN1**2)/(self.wr_gamma*MN1**2-((self
.wr_gamma-1)/2))) # This is from Dr. Whitmore's slides (same as above)
809
810         # Flow Deflection Angle
811         delta = math.atan2((2*((1/math.tan(self.wr_cone_angle))*(((self.
wr_mach_number**2)*(math.sin(self.wr_cone_angle))**2)-1))),
812                            ((self.wr_mach_number**2)*(self.wr_gamma+math.cos(2*self.
wr_cone_angle)))+2) # Equation 4.17 from Anderson
813         #delta = 11*math.pi/180
814         # Downstream Flow (Mach Number)
815         M2 = MN2/(math.sin(self.wr_cone_angle-delta)) # Equation 4.12 from Anderson
816         self.wr_m2n = MN2
817         self.wr_m2 = M2
818         self.wr_deflectionangle = delta
819     def shock_relations(self):
820
821         """
822         THE FOLLOWING EQUATIONS COME FROM PAGE 135-136 FROM MODERN COMPRESSIBLE FLOW:
 Dr. John D. Anderson
823         """
824         # Upstream Flow Normal to Cone (Mach Number)
825         MN1 = self.wr_mach_number*math.sin(self.wr_cone_angle) # Equation 4.7 from
Anderson
826         self.wr_mn1 = MN1
827         # Downstream Flow Nomral to Cone (Mach Number)
828         #MN21 = math.sqrt((MN1**2+(2/(self.wr_gamma-1)))/((2*self.wr_gamma/(self.
wr_gamma-1))*MN1**2)-1)) # Equation 4.10 from Anderson
829         if (1+((self.wr_gamma-1)/2)*MN1**2)/(self.wr_gamma*MN1**2-((self.wr_gamma-1)
/2)) < 0:
830             V_r = "No Solution"
831             V_theta = "No Solution"
832             return [V_r, V_theta]
833         MN2 = math.sqrt((1+((self.wr_gamma-1)/2)*MN1**2)/(self.wr_gamma*MN1**2-((self
.wr_gamma-1)/2))) # This is from Dr. Whitmore's slides (same as above)
834         self.wr_m2n = MN2
835
836         # Flow Deflection Angle
837         delta = math.atan2((2*((1/math.tan(self.wr_cone_angle))*(((self.
wr_mach_number**2)*(math.sin(self.wr_cone_angle))**2)-1))),
838                            ((self.wr_mach_number**2)*(self.wr_gamma+math.cos(2*self.
wr_cone_angle)))+2) # Equation 4.17 from Anderson
839         # Downstream Flow (Mach Number)
840         M2 = MN2/(math.sin(self.wr_cone_angle-delta)) # Equation 4.12 from Anderson
841
842         V1 = (((2/((self.wr_gamma-1)*M2**2))+1)**(-0.5)) # Equation 10.16 from
Anderson
843         V = math.sqrt((((self.wr_gamma - 1)/2)*M2**2)/(1+((self.wr_gamma - 1)/2)*M2
**2))
844
845         V = 1/math.sqrt((2/((self.wr_gamma-1)*M2**2))+1) # Equation 10.16 from
Anderson
846         V_r = V*math.cos(self.wr_cone_angle-delta) # Basic Trig
```

```
847         V_theta = -V*math.sin(self.wr_cone_angle-delta) # Basic Trig
848 
849         return [V_r, V_theta]
850 
851     def taylor_mccoll(self, vtheta, vr, theta):
852 
853         dvrdtheta = vtheta
854         dvthetadtheta = (vtheta**2*vr-(((self.wr_gamma-1)/2)*(1-vr**2-vtheta**2)*(2*
vr+vtheta*(1/math.tan(theta)))))/((((self.wr_gamma-1)/2)*(1-vr**2-vtheta**2))-vtheta
**2)
855 
856         return dvrdtheta, dvthetadtheta
857 
858     def solve_TayMac(self): # The numerical procedure is outlined in Pages 371-372 in
 Modern Compressible Flow
859         sorry = 0, 0
860         y0 = self.shock_relations()
861         if y0[0] == "No Solution":
862             print("OH NO")
863             sorry = 1, 0
864             return sorry
865         if y0[1] == "No Solution":
866             print("OH NO")
867             sorry = 1, 0
868             return sorry
869 
870         velocity_points = []
871 
872         theta = self.wr_cone_angle
873         first_column = []
874         i = 0
875         while y0[1] < 0:
876             first_column.append(theta)
877             ans = rk4_array(theta, y0, self.taylor_mccoll, self.wr_theta_step_size)
878             theta = theta-self.wr_theta_step_size
879             i += 1
880             y0[0] = ans[0]
881             y0[1] = ans[1]
882             velocity_points.append([y0[0], y0[1]])
883         self.wr_solved_cone_angle = theta
884         if self.wr_solved_cone_angle < 0.5*math.pi/180:
885             sorry = 1, 0.5*math.pi/180 - self.wr_solved_cone_angle
886         first_column = np.array(first_column)
887         velocity_points = np.array(velocity_points)
888         velocity_points = np.concatenate((first_column[:, None], velocity_points),
axis = 1)
889         velocity_points = velocity_points[:-1]  # Remove the last row
890         self.wr_vel_list = velocity_points
891 
892         csv_file_path = 'output.csv'
893 
894         np.savetxt(csv_file_path, self.wr_vel_list, delimiter=',')
895         csv_clickable_path = os.path.abspath(csv_file_path)
896 
897         return sorry
898 
899     def find_end_points(self):
900 
901         possible_points_list = []
902 
903         self.wr_cone_angle
904         self.wr_z_length
905         for i in range(361):
906             possible_points = sphere_to_cart(self.wr_z_length, self.wr_cone_angle, i*
math.pi/180)
```

```
907             possible_points_list.append(possible_points)
908
909         return possible_points_list
910
911     def TayMac_solution(self):
912
913         reflected_list = []
914         # Axisymettric!
915         reflected_list = [[-row[0], *row[1:]] for row in reversed(self.wr_vel_list)]
916         reflected_list = np.array(reflected_list)
917
918         newlist = self.wr_vel_list[::-1]
919
920         axisymettric = np.vstack((reflected_list, newlist))
921
922
923         final_list = []
924         # Translate!
925         for i in range(len(axisymettric)):
926             final_list.append([axisymettric[i][0]+math.pi, axisymettric[i][1],
axisymettric[i][2]])
927
928         self.wr_vel_list = np.array(final_list)
929
930     def get_streamline(self, theta):
931
932         if theta in self.wr_vel_list:
933             for index, row in enumerate(self.wr_vel_list):
934                 if row[0] == theta:
935                     desired_index = index
936             vr = self.wr_vel_list[desired_index, 1]
937             vtheta = self.wr_vel_list[desired_index, 2]
938         else:
939             pairs = find_closest_point(self.wr_vel_list, theta)
940             output = interpolate(pairs[0], pairs[1], theta)
941             vr = output[0]
942             vtheta = output[1]
943
944         return [vr, vtheta]
945
946     def streamline_integration(self, delta_t, n, r, theta, phi):
947
948         # Integrate the streamline n times at a selected single step size
949         lower_streamline_list = []
950
951         for i in range(n):
952
953             x, y, z = sphere_to_cart(r, theta, phi)
954             vr, vtheta = self.get_streamline(theta)
955
956             lower_streamline_list.append([x, y, z])
957             y0 = [r, theta]
958             ans = rk4_streamline(y0, self.streamline_eq, delta_t)
959             r, theta = ans
960
961         return [z, lower_streamline_list]
962
963     def function(self, guess, r, theta, phi):
964         output = self.streamline_integration(guess, self.wr_surface_quant, r, theta,
phi)
965         length = output[0]-self.wr_z_length
966         return length
967
968
969     def secant(self, initial_guess, gradient_step, x, y, z):
```

```
970
971         N = self.wr_surface_quant
972         distance = 1.00
973
974         xn1 = initial_guess - gradient_step
975         xn2 = initial_guess - (2*gradient_step)
976
977         xn1 = initial_guess + gradient_step
978         xn2 = initial_guess - gradient_step
979
980         r, theta, phi = cart_to_sphere(x, y, z)
981
982         xn = 0
983         i = 0
984
985
986         while abs(distance) > 1e-3:
987             i+= 1
988
989             output1 = self.streamline_integration(xn1, N, r, theta, phi)
990             final_length1 = output1[0]
991
992             output2 = self.streamline_integration(xn2, N, r, theta, phi)
993             final_length2 = output2[0]
994
995             fxn1 = final_length1 - self.wr_z_length
996             fxn2 = final_length2 - self.wr_z_length
997
998             if fxn1 == fxn2:
999                 break
1000
1001            xn = (xn2*fxn1-xn1*fxn2)/(fxn1-fxn2)
1002
1003            output = self.streamline_integration(xn, N, r, theta, phi)
1004            final_length = output[0]
1005
1006            fx = final_length - self.wr_z_length
1007
1008            xn2 = xn1
1009            xn1 = xn
1010
1011            distance = fx
1012
1013            if i > 3000:
1014
1015                gradient_step = 1e-8
1016                if abs(distance) < 1e-1:
1017                    return xn
1018
1019        if np.isnan(xn):
1020            print("Nan value encountered in secant method, exiting...")
1021            xn = 0.0
1022
1023        return xn
1024
1025    def streamline_eq(self, theta, r):
1026
1027        # Use Taylor-Maccoll solutions to trace streamlines. If the point lies in
    between solution points, interpolate
1028        vr, vtheta = self.get_streamline(theta)
1029
1030        drdt = vr
1031        if r == 0:
1032            dthetadt = 0
1033        else:
```

```
1034            dthetadt = (1/r)*vtheta
1035
1036        return [drdt, dthetadt]
1037
1038    def lower_surface(self):
1039
1040        streamline_list = []
1041
1042        for i in range(len(self.wr_leading_edge_geom)):
1043            initial_guess = (abs((self.wr_leading_edge_geom[i][2] - self.wr_z_length)
/self.wr_surface_quant))/math.cos(self.wr_cone_angle)
1044            step = 0.0001
1045            needed_step_size = self.secant(initial_guess, step, self.
wr_leading_edge_geom[i][0], self.wr_leading_edge_geom[i][1], self.
wr_leading_edge_geom[i][2])
1046            if needed_step_size == "Nan":
1047                print("Nan value encountered in secant method, exiting...")
1048                randomlist = [0,0]
1049                lower_surface1 = np.array(randomlist)
1050                exit_code = 1
1051                return lower_surface1, exit_code
1052            r, theta, phi = cart_to_sphere(self.wr_leading_edge_geom[i][0], self.
wr_leading_edge_geom[i][1], self.wr_leading_edge_geom[i][2])
1053            vr, vtheta = self.get_streamline(theta)
1054            streamline = self.streamline_integration(needed_step_size, self.
wr_surface_quant, r, theta, phi)
1055            streamline_list.append(streamline[1])
1056
1057        lower_surface = np.array(streamline_list)
1058        self.wr_lower_surface = lower_surface
1059        exit_code = 0
1060
1061        return lower_surface, exit_code
1062
1063    # UPPER SURFACE
----------------------------------------------------------------------
1064
1065    def trace_upper_surface(self, leading_edge_point): # Leading edge point is a 3D
vertex
1066
1067        step = 1/self.wr_surface_quant
1068        upper_surface_streamline = []
1069
1070        for i in range(self.wr_surface_quant):
1071            initial_guess = abs((self.wr_z_length-leading_edge_point[2])/(self.
wr_surface_quant-1))
1072
1073            needed_step_size = initial_guess
1074            z_point = leading_edge_point[2]+(needed_step_size*i)
1075            y_point = leading_edge_point[1]
1076            x_point = leading_edge_point[0]
1077            upper_surface_streamline.append([x_point, y_point, z_point])
1078
1079        np_upper_surface_streamline = np.array(upper_surface_streamline)
1080
1081        return np_upper_surface_streamline
1082
1083    def generate_upper_surface(self): # Leading edge points is a list of 3D vertex
points
1084
1085        upper_surface = []
1086
1087        for i in range(len(self.wr_leading_edge_geom)):
1088            lines = self.trace_upper_surface(self.wr_leading_edge_geom[i])
1089            upper_surface.append(lines)
```

```
1090
1091            upper_surface = np.array(upper_surface)
1092            self.wr_upper_surface = upper_surface
1093            return upper_surface
1094
1095        def base_plate(self):
1096            base_plate_list = []
1097            for i in range(self.wr_lead_edge_quant):
1098                first_point = self.wr_lower_surface[i][self.wr_surface_quant-1]
1099                second_point = self.wr_upper_surface[i][self.wr_surface_quant-1]
1100
1101                pair = [first_point, second_point]
1102
1103                base_plate_list.append(pair)
1104
1105            base_plate_list = np.array(base_plate_list)
1106
1107            self.wr_base_plate = base_plate_list
1108
1109            return base_plate_list
1110
1111        def number_verticies(self):
1112            top_left = []
1113            top_right = []
1114            upper_triangle_list = []
1115            leadingedge = []
1116            for i in range(self.wr_lead_edge_quant-3):
1117                for j in range(self.wr_surface_quant-1):
1118                    one = ((self.wr_surface_quant)*i)+j+1
1119                    two = ((self.wr_surface_quant)*(i+1))+j+1
1120                    three = ((self.wr_surface_quant)*(i+1))+(j+2)
1121                    four = one
1122                    five = one+1
1123                    six = three
1124                    upper_triangle_list.append([one, two, three])
1125                    upper_triangle_list.append([four, five, six])
1126
1127                    if j == self.wr_surface_quant-2:
1128                        leadingedge.append(five)
1129                        if i == self.wr_lead_edge_quant-4:
1130                            leadingedge.append(six)
1131
1132                    if i == 0:
1133                        top_left.append(one)
1134                    if i == self.wr_lead_edge_quant-4:
1135                        top_right.append(two)
1136
1137                    upper_triangle_list.append([one, two, three])
1138                    upper_triangle_list.append([four, five, six])
1139            top_left.append(top_left[-1]+1)
1140            top_right.append(top_right[-1]+1)
1141
1142
1143            last_index = upper_triangle_list[-1][-1]
1144
1145
1146
1147            corner1 = 0
1148            corner2 = last_index+1
1149
1150            trailingedge = []
1151            lower_triangle_list = []
1152            bottom_left = []
1153            bottom_right = []
1154            for i in range(self.wr_lead_edge_quant-3):
```

```
            for j in range(self.wr_surface_quant-1):
                if j == 0:
                    one = ((self.wr_surface_quant)*i)+j+1
                    two = ((self.wr_surface_quant)*(i+1))+j+1
                else:
                    one = ((self.wr_surface_quant)*i)+1-i+j+last_index
                    two = ((self.wr_surface_quant)*(i+1))+(j+2) + (last_index-i-1) -
1


                if i == 0:
                    five = ((self.wr_surface_quant)*i)+j+1+last_index+1


                four = one
                if j == 0:
                    five = last_index+(self.wr_surface_quant*i)+2-i
                else:
                    five = four+1
                three = ((self.wr_surface_quant)*(i+1))+(j+2) + (last_index-i-1)

                six = three

                if j == self.wr_surface_quant-2:
                    trailingedge.append(five)
                    if i == self.wr_lead_edge_quant-4:
                        trailingedge.append(six)

                if i == 0:
                    bottom_left.append(one)

                if i == self.wr_lead_edge_quant-4:
                    bottom_right.append(two)

                lower_triangle_list.append([one, two, three])
                lower_triangle_list.append([four, five, six])
        bottom_left.append(bottom_left[-1]+1)
        bottom_right.append(bottom_right[-1]+1)
        last_index_lower = lower_triangle_list[-1][-1]

        base_triangle_list = []

        for i in range(len(leadingedge)-1):
            if i == 0 :
                a = corner1
                b = leadingedge[i]
                c = trailingedge[i]

                base_triangle_list.append([a, b, c])
            if i == len(leadingedge)-2:
                a = corner2
                b = leadingedge[i+1]
                c = trailingedge[i+1]

                base_triangle_list.append([a, b, c])

            one = trailingedge[i]
            two = trailingedge[i+1]
            three = leadingedge[i+1]
            four = one
            five = leadingedge[i]
            six = three

            base_triangle_list.append([one, two, three])
            base_triangle_list.append([four, five, six])

```

```
1219        corners = []
1220        for i in range(len(top_left)-1):
1221            onel  = corner1
1222            twol = top_left[i]
1223            threel = top_left[i+1]
1224
1225            oner = corner2
1226            twor = top_right[i]
1227            threer = top_right[i+1]
1228
1229            onebl  = corner1
1230            twobl = bottom_right[i]
1231            threebl = bottom_right[i+1]
1232
1233            onebr = corner2
1234            twobr = bottom_left[i]
1235            threebr = bottom_left[i+1]
1236
1237
1238
1239            corners.append([onel, twol, threel])
1240            corners.append([oner, twor, threer])
1241            corners.append([onebl, twobl, threebl])
1242            corners.append([onebr, twobr, threebr])
1243
1244        triangle_list = upper_triangle_list + lower_triangle_list +
    base_triangle_list + corners
1245
1246        for i in range(len(triangle_list)):
1247            if triangle_list[i][0] == triangle_list[i][1]:
1248                print("triangle list at index", i, "has a duplicate value")
1249            if triangle_list[i][1] == triangle_list[i][2]:
1250                print("triangle list at index", i, "has a duplicate value (between
    two and three)")
1251
1252
1253        #
    ------------------------------------------------------------------------------------------
1254
1255        # Combine the vertices into one big list
1256
1257        corner1 = self.wr_upper_surface[0][0].tolist()
1258        corner2 = self.wr_upper_surface[-1][0].tolist()
1259
1260        upper_surface_vertices = [corner1]
1261
1262        for i in range(1, len(self.wr_upper_surface)-1):
1263            for j in range(len(self.wr_upper_surface[i])):
1264                upper_surface_vertices.append(self.wr_upper_surface[i][-(j+1)].tolist
    ())
1265
1266        upper_surface_vertices.append(corner2)
1267
1268        for i in range(1, len(self.wr_lower_surface)-1):
1269            for j in range(len(self.wr_lower_surface[i]) - 1):
1270                upper_surface_vertices.append(self.wr_lower_surface[-(i+1)][-(j+1)].
    tolist())
1271
1272        upper_surface_vis = np.array(upper_surface_vertices)
1273
1274        lengthofvertices = len(upper_surface_vertices)
1275        numberoftriangles = len(triangle_list)
1276
1277        triangle_list = mesh(self.wr_upper_surface, self.wr_lower_surface)
```

```
        triangle_list = mesh_2(self.wr_upper_surface, self.wr_lower_surface)
        for i in range(len(triangle_list)):
            triangle_list[i][0] = triangle_list[i][0]+1
            triangle_list[i][1] = triangle_list[i][1]+1
            triangle_list[i][2] = triangle_list[i][2]+1
        numberoftriangles = len(triangle_list)

        meshfile = self.wr_trifilename

        with open(meshfile, "w") as file:
            file.write(f"{lengthofvertices}\n")
            file.write(f"{numberoftriangles}\n")
            for item in upper_surface_vertices:
                file.write(' '.join(map(str, item)) + '\n')
            for item in triangle_list:
                file.write(' '.join(map(str, item)) + '\n')

        tri_clickable_path = os.path.abspath(meshfile)

    def find_average_area(self):
        half_no = ((self.wr_lead_edge_quant-1)/2)+1
        sum = 0
        for i in range(int(half_no)):
            sum += abs(self.wr_upper_surface[i][0][0])

        s_ref = 2*((sum/half_no)*self.wr_length)

        self.wr_mean_aerodynamic_chord = 2/s_ref*self.wr_integral

        return self.wr_length, s_ref

    def find_cg(self):
        length = len(self.wr_upper_surface)*len(self.wr_upper_surface[0])
        x = 0
        y = 0
        z = 0
        for i in range(len(self.wr_upper_surface)):
            for j in range(len(self.wr_upper_surface[i])):
                x += self.wr_upper_surface[i][j][0]
                y += self.wr_upper_surface[i][j][1]
                z += self.wr_upper_surface[i][j][2]
        x = x/length
        y = y/length
        z = z/length
        z = (self.wr_start_point+1)-(0.75*self.wr_mean_aerodynamic_chord)

        return x, y, z
```

Listing A.1: Sample Python Code

APPENDIX B

HI-MACH USAGE

```
import os
import json
import numpy as np
import shutil
import math

import subprocess as sp
import multiprocessing as mp

from copy import deepcopy

def move_file(source_file_path, destination_directory):
    # Check if the source file exists
    if not os.path.isfile(source_file_path):
        #print(f"Error: The source file {source_file_path} does not exist.")
        return

    # Check if the destination directory exists, if not create it
    if not os.path.exists(destination_directory):
        os.makedirs(destination_directory)

    # Get the base name of the file (e.g., 'example.txt')
    file_name = os.path.basename(source_file_path)

    # Construct the full path for the destination file
    destination_file_path = os.path.join(destination_directory, file_name)

    # Move the file
    try:
        shutil.move(source_file_path, destination_file_path)
        print(f"File moved successfully to {destination_file_path}")
    except Exception as e:
        print(f"Error: {e}")

def run_study(M, alpha, lref, sref, x, y, z, case_name, study_dir, beta, visc, blend,
 cone_angle, shock_angle):
    """Runs a case for given Mach number and mesh density"""

    results_file = study_dir + "results/"+case_name+".vtk"
    report_file = study_dir + "reports/"+case_name+".json"

    altitude = 0
    fttometer = 0.3048
    t_wall = 600
    if M == 4:
        altitude = 80000*fttometer
        t_wall = 725
    if M == 6:
        altitude = 100000*fttometer
        t_wall = 785
    if M == 10:
        altitude = 125000*fttometer
        t_wall = 960
    if M == 14:
```

```
54          altitude = 140000*fttometer
55          t_wall = 1020
56      if M == 20:
57          altitude = 175000*fttometer
58          t_wall = 1200
59       # INTERPOLATION GUESSES -------------
60
61      if M == 5:
62          altitude = 95000*fttometer
63          t_wall = 755
64      if M == 15:
65          altitude = 150000*fttometer
66          t_wall = 1050
67      if M == 25:
68          altitude = 200000*fttometer
69          t_wall = 1350
70
71      if visc == True:
72          input_dict = {
73              "flow":
74              {
75                  "alpha": alpha,
76                  "mach_number" : M,
77                  "gamma" : 1.4,
78                  "beta": beta,
79                  "altitude": altitude
80              },
81              "heating":
82              {
83                  "props_file": "/Users/adamweaver/Thesis/ThesisWaverider/WAVERIDER/HI-
  Mach/common/modified_Earth.props",
84                  "gas_model": 'tabulated',
85                  "use_rad_eq": False,
86                  "t_wall": t_wall,
87                  "boundary_layer_model": "transition",
88                  "xboundary_layer_model": "laminar",
89                  "streamline" :
90                  {
91                      "sparse": False
92                  },
93                  "reference_temperature_method": "eckert",
94                  "use_running_length_offset": False,
95                  "use_mangler_factor": False
96              },
97              "geometry" :
98              {
99                  "file": str(case_name)+'.tri',
100                 "reference":
101                 {
102                     "area": sref,
103                     "length": lref,
104                     "CG": [x, y, z]
105                 },
106                 "nose_axis": "z-",
107                 "pitch_axis": "x+",
108                 "scale": 60.0,
109                 "waverider" :
110                 {
111                     "cone_angle": cone_angle,
112                     "shock_angle": shock_angle
113                 },
114             },
115             "solver":
116             {
117                 "windward_method" : 'blended-waverider',
```

```
                "leeward_method" : 'prandtl-meyer',
                "xbase_pressure" : 'gaubeaud',
                "base_pressure" : 'none',
                "shielding_effects": False
            },
            "output":
            {
                "verbose": False,
                "body_file": results_file,
                "report_file": report_file
            }

        }
    else:
        input_dict = {
            "flow":
            {
                "alpha": alpha,
                "mach_number" : M,
                "gamma" : 1.4,
                "beta": beta,
                "altitude": 42672
            },
            "geometry" :
            {
                "file": str(case_name)+'.tri',
                "reference":
                {
                    "area": sref,
                    "length": lref,
                    "CG": [x, y, z]
                },
                "nose_axis": "z-",
                "pitch_axis": "x+",
                "scale": 1.0,
                "waverider" :
                {
                    "cone_angle": cone_angle,
                    "shock_angle": shock_angle
                },
            },
            "solver":
            {
                "xwindward_method" : 'blended-waverider',
                "windward_method" : 'tangent-cone',
                "leeward_method" : 'prandtl-meyer',
                "xbase_pressure" : 'gaubeaud',
                "base_pressure" : 'none',
                "shielding_effects": False
            },
            "output":
            {
                "verbose": False,
                "body_file": results_file,
                "report_file": report_file
            }

        }


    input_file = study_dir + "input.json"
    write_input_file(input_dict, input_file)
    base = os.path.join(study_dir, case_name+'.tri')
    new = os.path.join(study_dir, 'Hi-Mach')
    move_file(base, new)
```

```
183     report = run_himach(input_file, path=study_dir+'/Hi-Mach', delete_input=False,
    run=True)
184
185     C_F = np.zeros(3)
186     try:
187         C_F[0] = report["total_forces"]["Cx"]
188         C_F[1] = report["total_forces"]["Cy"]
189         C_F[2] = report["total_forces"]["Cz"]
190     except KeyError:
191         C_F[0] = np.nan
192         C_F[1] = np.nan
193         C_F[2] = np.nan
194     except TypeError:
195         C_F[0] = np.nan
196         C_F[1] = np.nan
197         C_F[2] = np.nan
198
199     C_M = np.zeros(3)
200     try:
201         C_M[0] = report["total_moments"]["CMx"]
202         C_M[1] = report["total_moments"]["CMy"]
203         C_M[2] = report["total_moments"]["CMz"]
204     except KeyError:
205         C_M[0] = np.nan
206         C_M[1] = np.nan
207         C_M[2] = np.nan
208     except TypeError:
209         C_M[0] = np.nan
210         C_M[1] = np.nan
211         C_M[2] = np.nan
212
213     try:
214         CL = report["total_forces"]["C_L"]
215         CD = report["total_forces"]["C_D"]
216     except KeyError:
217         CL = np.nan
218         CD = np.nan
219     except TypeError:
220         CL = np.nan
221         CD = np.nan
222
223     A = C_F[2]
224     Y = C_F[0]
225     N = C_F[1]
226
227     CD1 = (A*np.cos(alpha)*np.cos(beta) - Y*np.sin(beta) + N*np.sin(alpha)*np.cos(
    beta))
228     CL1 = (N*np.cos(alpha)-A*np.sin(alpha))
229
230     LD = CL1/CD1
231     Cl = C_M[2]
232     Cm = C_M[0]
233     Cn = C_M[1]
234
235     return LD, Cl, CL1, CD1, Cn, Cm
236
237 def run_himach(input_filename, path, delete_input=True, run=True):
238     """Runs HI-Mach with the given input and returns the report if NewPan generated
    one."""
239     os.chdir(path)
240
241     if run:
242         sp.run(["./himach.exe", input_filename])
243
244     with open(input_filename, 'r') as input_handle:
```

```
245         input_dict = json.load(input_handle)
246     report_file = input_dict["output"].get("report_file")
247     if report_file is not None:
248         try:
249             with open(report_file) as report_handle:
250                 report = json.load(report_handle)
251         except:
252             report = None
253     else:
254         report = None
255 
256     if delete_input:
257         os.remove(input_filename)
258 
259     return report
260 
261 def write_input_file(input_dict, input_filename):
262 
263     """Writes the given input dict to the given file location."""
264 
265     with open(input_filename, 'w') as input_handle:
266 
267         json.dump(input_dict, input_handle, indent=4)
268 
269 def solve_fluid(M, alpha, lref, sref, x, y, z, case_name, study_dir, beta, run_case,
    val, visc, blend, cone_angle, shock_angle):
270 
271     LD1, Cl1, CL1, CD1, Cn1, Cm1 = run_study(M, alpha, lref, sref, x, y, z, case_name
    , study_dir, beta-(5*math.pi/180), visc, blend, cone_angle, shock_angle)
272     LD2, Cl2, CL2, CD2, Cn2, Cm2 = run_study(M, alpha, lref, sref, x, y, z, case_name
    , study_dir, beta+(5*math.pi/180), visc, blend, cone_angle, shock_angle)
273     LD3, Cl3, CL3, CD3, Cn3, Cm3 = run_study(M, alpha+(5*math.pi/180), lref, sref, x,
     y, z, case_name, study_dir, beta, visc, blend, cone_angle, shock_angle)
274     LD4, Cl4, CL4, CD4, Cn4, Cm4 = run_study(M, alpha-(5*math.pi/180), lref, sref, x,
     y, z, case_name, study_dir, beta, visc, blend, cone_angle, shock_angle)
275     LD0, Cl0, CL0, CD0, Cn0, Cm0 = run_study(M, alpha, lref, sref, x, y, z, case_name
    , study_dir, beta, visc, blend, cone_angle, shock_angle)
276 
277     Cl_beta = ((Cl2-Cl1)/(10*math.pi/180))
278     Cn_beta = ((Cn2-Cn1)/(10*math.pi/180))
279     Cm_alpha = ((Cm3-Cm4)/(10*math.pi/180))
280 
281     if run_case == 0:
282         return -LD1
283     if run_case == 1:
284         return -1*(Cl_beta - val)
285     if run_case == 2:
286         return CL1
287     if run_case == 3:
288         return -CD1
289     if run_case == 4:
290         return Cn_beta - val
291     if run_case == 5:
292         return Cl1
293     if run_case == 6:
294         return Cn1
295     if run_case == 10:
296         return Cl_beta
297     if run_case == 11:
298         return Cn_beta
299     if run_case == 12:
300         return CD1
301     if run_case == 13:
302         return Cm_alpha
303     if run_case == 20:
```

```
304             return -LD1/100, Cl_beta, Cn_beta, Cm_alpha
305         if run_case == 21:
306             print("Cl1 is: ", Cl0)
307             print("Cn1 is: ", Cn0)
308 
309             return LD0, Cl_beta, Cn_beta, Cm_alpha
310         if run_case == 22:
311             return LD0, Cl0, Cn0, Cm0
```

Listing B.1: Sample Python Code

APPENDIX C

OPTIMIZATION WRAPPER

```
import scipy as sc
import geom
import HiMach
import time as t
import math
import time
import numpy as np
import matplotlib.pyplot as plt
import os
import tkinter
from matplotlib.backends.backend_tkagg import (
    FigureCanvasTkAgg, NavigationToolbar2Tk)
# Implement the default Matplotlib key bindings.
from matplotlib.backend_bases import key_press_handler
from matplotlib.figure import Figure
import numpy as np

def normalize(x, a, b):
    if b == a == 0:
        return 0
    else:
        return (x - a) / (b - a)

def denormalize(x, a, b):
    return x * (b - a) + a

class Family:

    f_init_bounds = []
    f_init_nonnorm = []

    def __init__(self, mach_number, cone_angle, lead_edge_quant, surface_quant, gamma, theta_step, stream_step,
                 length, plotLeadingEdge, name, path, alpha, beta, vals, visc, blend, constrain, optimtype, initial,
                 method, tol, constraintype):

        # Shock Properities -----------------------------------
        self.f_mach_number = mach_number # Operating Mach Number
        self.f_cone_angle = cone_angle # Assumed Shock Angle
        self.f_gamma = gamma # Specific Heat Ratio
        # Mesh Propeties --------------------------------------
        self.f_lead_edge_quant = lead_edge_quant # How many leading edge points do you want?
        self.f_surface_quant = surface_quant # How many z cuts should the mesh use?
        self.f_theta_step_size = theta_step # What is the fineness of the RK4 integration scheme for the Taylor-Maccoll Integration?
        self.f_streamline_step = stream_step # What is the fineness of the RK4 integration scheme for the streamline tracing?
        self.f_z_length = length
        self.f_length = length
        self.f_plotLeadingEdge = plotLeadingEdge
        self.f_name = name
        self.f_path = path
```

```
50          self.f_alpha = alpha
51          self.f_beta = beta
52          self.f_vals = vals
53          self.f_method = method
54          self.f_tol = tol
55          self.f_alpha_init = self.f_alpha
56          self.f_cone_angle_init = self.f_cone_angle
57          initial += [self.f_cone_angle_init]
58          self.f_init = initial.copy()
59          self.f_visc = visc
60          self.f_blend = blend
61          self.f_constrain = constrain
62          self.f_optimtype = optimtype
63          self.f_constraintype = constraintype
64
65      def callback(self, xk):
66          end_time = time.time()
67          print(f"Iteration time: {end_time - self.start_time} seconds")
68          print(f"Current solution vector: {xk}")
69          self.start_time = end_time
70
71      def geometry(self, variable_vals, case):
72
73          print("WHAT IS", int(len(variable_vals)-1))
74          KEVIN = geom.WaveRider(self.f_mach_number, self.f_cone_angle*math.pi/180,
    self.f_lead_edge_quant, self.f_surface_quant, self.f_gamma, self.f_theta_step_size,
    self.f_streamline_step, self.f_z_length, int(len(variable_vals)), variable_vals,
75                             self.f_name+'.tri', self.f_plotLeadingEdge)
76          if self.f_plotLeadingEdge == True:
77              LeadingEdgePolyPoints = geom.get_leadingedge_from_plot()
78              le_constraint, y = KEVIN.leading_edge(LeadingEdgePolyPoints[0],
    LeadingEdgePolyPoints[1], LeadingEdgePolyPoints[2])
79          else:
80              le_constraint, y = KEVIN.leading_edge(variable_vals[0], variable_vals[1],
     variable_vals[2], variable_vals[3], variable_vals[4])
81          KEVIN.solve_TayMac()
82          k = KEVIN.lower_surface()
83          KEVIN.generate_upper_surface()
84          KEVIN.base_plate()
85          self.f_l_ref, self.f_s_ref = KEVIN.find_average_area()
86          self.f_cg_x, self.f_cg_y, self.f_cg_z = KEVIN.find_cg()
87          KEVIN.number_verticies()
88          if case <= 6:
89              value = HiMach.solve_fluid(self.f_mach_number, self.f_alpha*math.pi/180,
    self.f_l_ref, self.f_s_ref, self.f_cg_x, self.f_cg_y, self.f_cg_z, self.f_name, self.
    f_path, self.f_beta, case, self.f_vals)
90          if case == 7:
91              value = le_constraint
92          if case == 8:
93              value = -y
94          if case == 9:
95              value = KEVIN.wr_solved_cone_angle
96
97          return value
98
99      def gen_geometry(self, variable_vals, case):
100
101         variable_vals_plug = []
102
103         for i in range(len(variable_vals)):
104             variable_vals_plug.append(denormalize(variable_vals[i], self.
    f_init_bounds[i][0], self.f_init_bounds[i][1]))
105
106         variable_vals_plug = np.nan_to_num(variable_vals_plug, nan=0.0, posinf=1e10,
    neginf=-1e10)
```

```
107
108
109         KEVIN = geom.WaveRider(self.f_mach_number, variable_vals_plug[-1]*math.pi
    /180, self.f_lead_edge_quant, self.f_surface_quant,
110                                self.f_gamma, self.f_theta_step_size, self.
    f_streamline_step, self.f_z_length, int(len(variable_vals)-1), variable_vals_plug,
111                            self.f_name+'.tri', self.f_plotLeadingEdge)
112         if self.f_plotLeadingEdge == True:
113             LeadingEdgePolyPoints = geom.get_leadingedge_from_plot()
114             le_constraint, y = KEVIN.leading_edge(LeadingEdgePolyPoints[0],
    LeadingEdgePolyPoints[1], LeadingEdgePolyPoints[2])
115         else:
116             le_constraint, y = KEVIN.leading_edge(*variable_vals_plug[:int(len(
    variable_vals_plug)-1)])
117         b = KEVIN.solve_TayMac()
118         if b[0] == 1:
119             self.f_l_ref, self.f_s_ref = 1,1
120             self.f_cg_x, self.f_cg_y, self.f_cg_z = 0,0,0.5
121             print("Error in geometry")
122             value = 0.1*abs(b[1])
123             return value
124         else:
125             k = KEVIN.lower_surface()
126             KEVIN.generate_upper_surface()
127             KEVIN.base_plate()
128             self.f_l_ref, self.f_s_ref = KEVIN.find_average_area()
129             self.f_cg_x, self.f_cg_y, self.f_cg_z = KEVIN.find_cg()
130             KEVIN.number_verticies()
131             if case <= 6 or case == 13:
132                 if self.f_optimtype == 'mindrag':
133                     value = HiMach.solve_fluid(self.f_mach_number, self.f_alpha*math.
    pi/180, self.f_l_ref,
134                                                self.f_s_ref, self.f_cg_x, self.f_cg_y
    , self.f_cg_z, self.f_name,
135                                                self.f_path, self.f_beta, 12, self.
    f_vals, self.f_visc, self.f_blend,
136                                                KEVIN.wr_solved_cone_angle,
    variable_vals_plug[-1]*math.pi/180)
137                 else:
138                     value = HiMach.solve_fluid(self.f_mach_number, self.f_alpha*math.
    pi/180, self.f_l_ref,
139                                                self.f_s_ref, self.f_cg_x, self.f_cg_y
    , self.f_cg_z, self.f_name,
140                                                  self.f_path, self.f_beta, case, self
    .f_vals, self.f_visc, self.f_blend,
141                                                KEVIN.wr_solved_cone_angle,
    variable_vals_plug[-1]*math.pi/180)
142             if case == 7:
143
144                 value = le_constraint
145             if case == 8:
146                 value = -y
147             if case == 9:
148                 value = KEVIN.wr_solved_cone_angle
149             return value
150
151     def run_geometry(self):
152         self.f_mach_number =25
153         self.f_beta = 0.0
154         self.f_alpha = 0.04363323129985824
155         self.f_l_ref = 1.0
156         self.f_s_ref = 0.22484419297131097
157         self.f_cg_x = 0.0
158         self.f_cg_y = -0.09765280659914805
159         self.f_cg_z = 1.3930102537794176
```

```
160         HiMach.run_study(self.f_mach_number, self.f_alpha*math.pi/180, self.f_l_ref,
    self.f_s_ref, self.f_cg_x, self.f_cg_y, self.f_cg_z, self.f_name, self.f_path, self.
    f_beta*math.pi/180)
161
162     def optimize(self, func, constraint = True):
163
164         bounds = []
165         bounds.extend([(0, 1)] * len(self.f_init))
166         length_of_bounds = len(self.f_init)
167         for i in range(length_of_bounds):
168             self.f_init_bounds.append((0, 1))
169         if len(self.f_init) == 5:
170             self.f_init_bounds[len(self.f_init)-5] = (-1000,1000)
171         if len(self.f_init) == 4:
172             self.f_init_bounds[len(self.f_init)-4] = (-100,100)
173         self.f_init_bounds[len(self.f_init)-3] = (-10,10)
174         self.f_init_bounds[len(self.f_init)-2] = (-3, -0.01)
175         lower_bound = 180/math.pi*math.asin(1/self.f_mach_number)
176         self.f_init_bounds[len(self.f_init)-1] = (lower_bound+0.01, lower_bound+25)
177         self.f_init_nonnorm = self.f_init.copy()
178         f_init_copy = self.f_init.copy()
179         initialize_list = self.f_init.copy()
180
181         for i in range(len(self.f_init)):
182             initialize_list[i] = normalize(f_init_copy[i], self.f_init_bounds[i][0],
    self.f_init_bounds[i][1])
183
184         self.start_time = time.time()
185         if constraint == True:
186             if self.f_constraintype == 'Clb':
187                 cons1 = [{'type':'ineq', 'fun': func, 'args': [1]}]
188                 def cons_func(x):
189                     return self.gen_geometry(x, 1)
190             if self.f_constraintype == 'Cnb':
191                 cons1 = [{'type':'ineq', 'fun': func, 'args': [4]}]
192                 def cons_func(x):
193                     return self.gen_geometry(x, 1)
194             if self.f_constraintype == 'Cma':
195                 cons1 = [{'type':'ineq', 'fun': func, 'args': [13]}]
196                 def cons_func(x):
197                     return self.gen_geometry(x, 1)
198
199             if self.f_method == 'COBYLA':
200                 result = sc.optimize.minimize(func, initialize_list, args = 0.0,
    method = self.f_method,
201                                               constraints = cons1, bounds = bounds, tol
     = self.f_tol,callback = self.callback,
202                                                 options={
203                                                          'maxiter': 10000,
204                                                          'rhobeg': 0.001,
205                                                          'rhoend': 1e-5,
206                                                          'catol': 1e-1
207                                                       })
208             if self.f_method == 'SLSQP':
209                 result = sc.optimize.minimize(func, initialize_list, args = 0.0,
    method = self.f_method,
210                                               constraints = cons1, bounds = bounds, tol
     = self.f_tol)
211                                               #options={'eps': 0.00000001, 'maxiter':
    10000, 'finite_diff_rel_step': 0.025}, callback = self.callback)
212             if self.f_method == 'SHGO':
213                 result = sc.optimize.shgo(func, bounds = bounds, args = (0.0,),
214                                             constraints = cons1, n=200, iters=5,
    sampling_method='simplicial',
```

```
215                                                 options={'qhull_options': 'QJ QbB Qs Qz
', 'n': 100,'local_iter': 5})
216             if self.f_method == 'trust-constr':
217                 from scipy.optimize import NonlinearConstraint
218                 cons1 = NonlinearConstraint(cons_func, 0, np.inf)
219                 result = sc.optimize.minimize(func, initialize_list, args = 0.0,
method = self.f_method,
220                                               constraints = cons1, bounds = bounds,
tol = self.f_tol)
221
222             elif self.f_method == 'DE':
223                 from scipy.optimize import NonlinearConstraint
224                 cons1 = NonlinearConstraint(cons_func, 0, np.inf)
225                 result = sc.optimize.differential_evolution(
226                                                         func,
227                                                         bounds=bounds,
228                                                         constraints=cons1,
229                                                         maxiter=500,
230                                                         tol=0.01,
231                                                         workers=-1
232                                                     )
233             self.f_optimtype = 'LD'
234             LD = self.gen_geometry(result.x, 0)
235             CLBetamin = self.gen_geometry(result.x, 1)
236             Cnbetamin = self.gen_geometry(result.x, 4)
237             CL = self.gen_geometry(result.x, 2)
238             Cl = self.gen_geometry(result.x, 5)
239             Cn = self.gen_geometry(result.x, 6)
240
241         else:
242             cons2 = [{'type':'ineq', 'fun': func, 'args': [8]}]
243             if self.f_method == 'COBYLA':
244                 result = sc.optimize.minimize(func, initialize_list, args = 0.0,
method = self.f_method,
245                                               bounds = bounds, tol = self.f_tol,
246                                                options={
247                                                         'maxiter': 10000,
248                                                         'rhobeg': 0.001,
249                                                         'rhoend': 1e-5,
250                                                         'catol': 1e-1
251                                                     }, callback = self.callback)
252             else:
253                 result = sc.optimize.minimize(func, initialize_list, args = 0.0,
method = self.f_method,
254                                               bounds = bounds, tol = self.f_tol,
callback = self.callback,
255                                               options={'finite_diff_rel_step': 0.05, '
ftol': 1e-3})
256                                              # options={'eps': 0.05, 'ftol': 1e-5, '
maxiter': 400})
257
258             self.f_optimtype = 'LD'
259             LD = self.gen_geometry(result.x, 0)
260             CLBetamin = self.gen_geometry(result.x, 1)
261             Cnbetamin = self.gen_geometry(result.x, 4)
262             CL = self.gen_geometry(result.x, 2)
263             Cl = self.gen_geometry(result.x, 5)
264             Cn = self.gen_geometry(result.x, 6)
265
266         return result, LD, CLBetamin, Cnbetamin, CL, Cl, Cn
267
268     def optimize_noconst(self, func):
269
270
```

```
271         result = sc.optimize.minimize(func, self.f_init, args = 0.0, method = self.
    f_method, tol = self.f_tol, options={'maxiter': 100})
272
273         LD = HiMach.solve_fluid(self.f_mach_number, self.f_alpha*math.pi/180, self.
    f_l_ref, self.f_s_ref, self.f_cg_x, self.f_cg_y, self.f_cg_z, self.f_name, self.
    f_path, self.f_beta*math.pi/180, 0, self.f_vals)
274         CLBetamin = HiMach.solve_fluid(self.f_mach_number, self.f_alpha*math.pi/180,
    self.f_l_ref, self.f_s_ref, self.f_cg_x, self.f_cg_y, self.f_cg_z, self.f_name, self.
    f_path, self.f_beta*math.pi/180, 1, self.f_vals)
275         Cnbetamin = HiMach.solve_fluid(self.f_mach_number, self.f_alpha*math.pi/180,
    self.f_l_ref, self.f_s_ref, self.f_cg_x, self.f_cg_y, self.f_cg_z, self.f_name, self.
    f_path, self.f_beta*math.pi/180, 4, self.f_vals)
276         CL = HiMach.solve_fluid(self.f_mach_number, self.f_alpha*math.pi/180, self.
    f_l_ref, self.f_s_ref, self.f_cg_x, self.f_cg_y, self.f_cg_z, self.f_name, self.
    f_path, self.f_beta*math.pi/180, 2, self.f_vals)
277         Cl = HiMach.solve_fluid(self.f_mach_number, self.f_alpha*math.pi/180, self.
    f_l_ref, self.f_s_ref, self.f_cg_x, self.f_cg_y, self.f_cg_z, self.f_name, self.
    f_path, self.f_beta*math.pi/180, 5, self.f_vals)
278         Cn = HiMach.solve_fluid(self.f_mach_number, self.f_alpha*math.pi/180, self.
    f_l_ref, self.f_s_ref, self.f_cg_x, self.f_cg_y, self.f_cg_z, self.f_name, self.
    f_path, self.f_beta*math.pi/180, 6, self.f_vals)
279
280         return result, LD, CLBetamin, Cnbetamin, CL, Cl, Cn
```

Listing C.1: Sample Python Code