Accelerating Practical Engineering Design Optimization with Computational Graph Transformations

by

Peter D. Sharpe

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Aeronautics and Astronautics
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September 2024

Author . . . .
Department of Aeronautics and Astronautics
August 16, 2024

Certified by . . . .
R. John Hansman
T. Wilson Professor, MIT
Thesis Supervisor

Certified by . . . .
Mark Drela
Terry J. Kohler Professor, MIT
Thesis Supervisor

Certified by . . . .
Karen Willcox
Peter O'Donnell, Jr. Centennial Chair in Computing Systems, UT Austin
Thesis Supervisor

Accepted by . . . .
Jonathan How
Richard Cockburn Maclaurin Professor, MIT
Chair, Graduate Program Committee

Accelerating Practical Engineering Design Optimization with Computational Graph Transformations

by

Peter D. Sharpe

Submitted to the Department of Aeronautics and Astronautics
on August 16, 2024, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Aeronautics and Astronautics

Abstract

Multidisciplinary design optimization has immense potential to improve conceptual design workflows for large-scale engineered systems, such as aircraft. However, despite remarkable theoretical progress in advanced optimization methods in recent decades, practical industry adoption of such methods lags far behind. This thesis identifies the root causes of this theory-to-practice gap and addresses them by introducing a new paradigm for computational design optimization frameworks called *code transformations*. Code transformations encompass a variety of computational-graph-based scientific computing strategies (e.g., automatic differentiation, automatic sparsity detection, problem auto-scaling) that automatically analyze, augment, and accelerate the user's code before passing it to a modern gradient-based optimization algorithm.

This paradigm offers a compelling combination of ease-of-use, computational speed, and modeling flexibility, whereas existing paradigms typically make sacrifices in at least one of these key areas. Consequently, code transformations present a competitive avenue for increasing the adoption of advanced optimization techniques in industry, all without placing the burden of deep expertise in applied mathematics and computer science on end users.

The major contributions of this thesis are fivefold. First, it introduces the concept of code transformations as a possible foundation for an MDO framework and demonstrates their practical feasibility through aircraft design case studies. Second, it implements several common aircraft analyses in a form compatible with code transformations, providing a practical illustration of the opportunities, challenges, and considerations here. Third, it presents a novel technique to automatically trace sparsity through certain external black-box functions by exploiting IEEE 754 handling of not-a-number (NaN) values. Fourth, it proposes strategies for efficiently incorporating black-box models into a code transformation framework through physics-informed machine learning surrogates, demonstrated with an airfoil aerodynamics analysis case study. Finally, it shows how a code transformations paradigm can simplify the formulation of other optimization-related aircraft development tasks beyond just design, exemplified by aircraft system identification and performance reconstruction from minimal flight data.

Taken holistically, these contributions aim to improve the accessibility of advanced optimization techniques for industry engineers, making large-scale conceptual multidisciplinary design optimization more practical for real-world systems.

Thesis Supervisor: R. John Hansman
Title: T. Wilson Professor, MIT

Thesis Supervisor: Mark Drela
Title: Terry J. Kohler Professor, MIT

Thesis Supervisor: Karen Willcox
Title: Peter O'Donnell, Jr. Centennial Chair in Computing Systems, UT Austin

Thesis Reader: Tony Tao
Title: Technical Staff, MIT Lincoln Laboratory

Thesis Reader: Joaquim R. R. A. Martins
Title: Pauline M. Sherman Collegiate Professor, University of Michigan

Acknowledgments

This thesis and my associated research endeavors would not have been possible without the support of so many people, and for that I am deeply thankful.

In particular, I would like to thank my advisor, Prof. John Hansman, for his unwavering support and guidance throughout my time at MIT. I'm also indebted to Prof. Mark Drela, both for his personal mentorship and teaching, as well as his broader professional contributions – much of the work in this thesis would not have been possible without his prolific contributions to the field of aerospace engineering. I would also like to thank Prof. Karen Willcox, who graciously agreed to serve on my doctoral committee and has provided wise and valuable insights throughout our meetings. I am also deeply grateful for Dr. Tony Tao and Prof. Joaquim Martins, both for serving as readers on this thesis and for their contributions to the field.

I would like to thank the open-source scientific computing community for their contributions to this work. In particular, I'd like to thank the developers of NumPy, SciPy, CasADi, JAX, IPOPT, and the Python programming language for their tireless work maintaining tools that have enabled so much of the research in this thesis. The modern day open-source software ecosystem epitomizes the spirit of worldwide collaboration that Newton described so long ago: "If I have seen further, it is by standing on the shoulders of giants." It is a privilege to be a part of this global open-source community, and I am grateful for the opportunity to pay it forward in my own small way.

Various parts of the research in this thesis were funded by the MIT Portugal Program, BAE Systems, and the National Defense Science and Engineering Graduate Fellowship, and I thank them for their support. I would also like to thank the MIT SuperCloud team for supporting my free access to supercomputing resources that enabled parts of this research.

At MIT AeroAstro, I would like to thank my labmates in ICAT for their camraderie and support over the years, especially Matt Vernacchia, Kelly Mathesius, Julia Gaubatz, Annick Dewald, Ara Mahseridjian, Chris Courtin, Estelle Martin, Trevor Long, Juju Wang, Sandro Salgueiro, Marek Travnik, Mina Cezairli, and many others. I've been so fortunate to spend the past few years alongside such a talented and friendly group of people.

Also at AeroAstro, I'd like to thank machine shop staff Todd Billings and Dave Robertson for their friendship and help over the years. When you've been stuck on a hard math problem for days, sometimes there's nothing like cutting some balsa and building a model airplane to clear your head.

Closer to home, I'd like to thank my friends, family, and loved ones – the only thing that really matters. Your support and encouragement means the world to me. Most of all, I'd like to thank my partner and best friend, Marta Manzin, for her seemingly infinite love and support throughout this journey. She has been my rock through the ups and downs of graduate school. I cannot imagine having done this without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Our everyday lives are filled with interactions with complex engineering systems, from the cars we drive, to the electricity grid that powers our cities, to the aircraft we fly in. With every passing year, the design requirements that we place on these systems increase in scope. This inexorable progress towards "doing more with less" has brought staggering benefits in performance, cost, safety, and environmental impact:

as just one example, the fuel economy[1] of both consumer cars and commercial aircraft has roughly tripled over the past 60 years [25, 26].

This progress, however, often comes at the cost of increased complexity in the engineering design process. Satisfying these requirements often involves adding new subsystems or considering new analyses. As the number of these subsystems and analyses grows, the interactions between them also become more numerous, important, and non-intuitive. This presents both challenge and opportunity. The challenge is that, even at the earliest conceptual design stages, the designer must juggle a far wider problem to even begin to meaningfully understand and improve the system's performance. The opportunity is that, by understanding and exploiting these subsystem interactions, the designer can unlock vast new design spaces that were previously inaccessible [27].

Fortunately, computational techniques have evolved to meet the challenge of managing this ever-increasing complexity and navigating these new design spaces in a human-understandable way. These techniques go by many names in various fields, but in the context of aerospace engineering, they are often referred to as *Multidisciplinary Design Optimization* (MDO) [28]. In a typical MDO workflow, an end-user develops a problem-specific *design tool* under the umbrella of some larger MDO *framework*; this framework orchestrates communication between individual disciplinary models. The resulting problem-specific design tool can then be used to explore the design space of the engineering system.

The recognized potential of MDO frameworks to unlock the next generation of performance improvements has motivated a great deal of research in the field. MDO saw rapid progress beginning in the late 1980s, with early works building the mathematical and computer science basis for integrated computational design tools that couple analyses across engineering disciplines [29–31]. Even from these early days, practitioners observed that *industrial* adoption of these tools was not necessarily

[1]Measured in miles per gallon for cars, and fuel burn per passenger-mile for aircraft

straightforward [32, 33]. Often, the reasons for this were not purely technical—the complexity of integrating models into MDO tools, the difficulties of quickly formulating and iterating on design problems, and the fragility of resulting designs in the absence of careful modeling all presented significant practical barriers.

Significant progress in the development of MDO frameworks has been made in the decades since. However, surveys assessing the state of aircraft MDO years later still reveal a persisting gap between MDO's potential and practice [27, 34]. Documented examples of complete built-and-flown aircraft designed with MDO exist, but remain less common than one might otherwise expect, even as many technical barriers have fallen to the exponential rise in computational power in recent years [35].

The core remaining barriers to widespread adoption of conceptual-level MDO relate strongly to practical limitations at the human-optimizer interface, rather than purely on traditional metrics of computational speed [7]. The complexity intrinsic to coordinating many coupled models across disciplines poses inherent difficulties for design interpretation, method credibility, and managing tradeoffs [36]. Modern scientific computing capabilities enable unprecedented problem scale, but push human limitations in complexity management. The weak link is no longer the code itself, but the practical frictions of formulating and interpreting large design problems.

As we develop the next generation of MDO frameworks, the challenge is to make available the fruits of MDO without requiring end-users to be joint experts in applied mathematics and computer science on top of their engineering domain expertise [37]. The path forward for maximizing the benefits of MDO lies in a deliberate focus on the practical human interface with optimization tools, from the perspective of an industrial end-user.

1.1 Project Definition and Thesis Overview

This thesis addresses these challenges by proposing five novel contributions that collectively map out a path towards improved usability and effectiveness of MDO frameworks:

1. **Code Transformation Paradigm** (Chapter 3): First, this thesis conceptually introduces a new computational paradigm for MDO frameworks based on *code transformations*, a related collection of modern scientific computing techniques. The value proposition of this paradigm is that it gives end-users most of the benefits of existing state-of-the-art paradigms, but with fewer of the practical user frictions that impede industry use. To demonstrate this, this thesis provides a proof-of-concept implementation of this paradigm in the open-source *AeroSandbox* [1, 38] framework. In this thesis, this framework is benchmarked against existing frameworks on both technical and non-technical metrics to evaluate the aforementioned value proposition.

    - **Aircraft Design Case Studies** (Chapter 4): The developed framework is then applied to a series of aircraft design problems, many of which supported real-world aircraft development programs. This facilitates more precise discussion about the possible opportunities and challenges of this proposed framework-level MDO approach. These case studies also provide more realistic performance benchmarks, as well as starting points for future applied research.

2. **Traceable Physics Models** (Chapter 5): This chapter provides implementations of several common aerospace physics models that serve as optional, modular plugins into the example MDO framework of Chapter 3. These plugins aim to a) stress-test the code transformation paradigm on common analysis code patterns and b) serve as building blocks to enable end-users to rapidly formulate aircraft design problems.

3. **Sparsity Tracing via NaN-Propagation** (Chapter 6): This chapter conceptually introduces the novel idea of "NaN-propagation" as a technique to trace sparsity through black-box numerical analyses by exploiting floating-point math handling. This opens a path to including black-box models in a code transformations framework while still retaining some (but not all) of the speed advantages over existing options.

4. **Physics-Informed Machine Learning Surrogates for Black-Box Models** (Chapter 7): This chapter explores a strategy to incorporate black-box models into a code transformation framework, where a model is replaced with a learned approximator with desirable mathematical properties. As an example, a physics-informed machine learning surrogate model for airfoil aerodynamics analysis will be presented. This demonstrates that accurate surrogates can be constructed to stand in for complex analyses while retaining compatibility with code transformations.

5. **Aircraft System Identification from Minimal Sensor Data** (Chapter 8): Finally, this chapter demonstrates that a code transformation paradigm enables rapid formulation of optimization problems in applications beyond just design. This can make it attractive to use numerical optimization in contexts where formulation would have otherwise been overly tedious. As an example, the thesis will show an application to aircraft system identification and performance reconstruction from minimal flight data. Here, physics-based corrections are used alongside statistical inference techniques to accurately estimate aircraft performance characteristics from a single short test flight.

Chapter 2

Literature Review of Aircraft Design Optimization

In this chapter, we review the past, present, and future of aircraft design optimization, with the goal of identifying long-standing challenges, recent advances, and new opportunities. Section 2.1 provides historical background on the emergence of aircraft design optimization as its own research field, with the goal of placing the present work in context; the familiar reader is invited to skip it. Section 2.2 evaluates how these past research efforts have translated to practical industry impact, both historically and into the present day. Section 2.3 discusses recent research literature and perspectives on aircraft design optimization from the past decade, particularly those elements that could help operationalize this research into industry. Finally, Section 2.4 presents a forward-looking value proposition for how modern optimization techniques can improve aircraft design workflows; this focuses on specific, tangible industry examples that motivate the broader research direction of this thesis.

2.1 Historical Promises, Predictions, and Pitfalls

> "When an aircraft designer hears that a new program will use multidisciplinary optimization, the reaction is often less than enthusiastic. Over the past 30 years, aircraft optimization at the conceptual and preliminary design levels has often yielded results that were either not believable, or might have been obtained more simply using methods familiar to the engineers. Even 5 to 20 years ago, actual industry application of numerical optimization for aircraft preliminary design was not widespread."
>
> -Ilan Kroo, 1997 [32]

These are the opening lines of a landmark 1997 paper that reviewed the state of the art and future directions in the then-emerging field of aircraft multidisciplinary design optimization (MDO) [32]. The paper, by aircraft designer and MDO pioneer Ilan Kroo, not only reviewed the status of the field's academic research, but also took the key step of assessing whether these research advances had translated to practical industry impact. As a later 2010 review by an MDO technical organizing committee would emphasize, "the ultimate benchmark of a research field's impact is indicated by the realization of its theories into successful products throughout industry." [34]

Despite the paper's stark opening, Kroo's position piece does highlight many reasons for optimism, and it is an interesting window into contemporaneous perspectives at a pivotal moment in the field's history. On one hand, Kroo noted the auspicious progress of aircraft MDO research during the preceding decades by all traditional metrics: problem size, analysis fidelity, runtime speed, and so on. He credited these successes largely to both algorithmic advances and the exponential growth of computational power over time (Moore's law[1]). Extrapolating these trends forward, he concluded that the field is poised to make a significant impact on the aircraft design process across both academia and industry. The promise of MDO is made clear in a remark that many aircraft designers would still agree with today: "In a very real sense, preliminary design is MDO." [32]

On the other hand, Kroo noted that actual industry applications of aircraft MDO remained conspicuously limited as of the paper's 1997 publication, and "many difficulties remain in the routine application of MDO." [32] Other works from the early days of aircraft MDO corroborate this relative dearth of industry adoption. For example, in an AIAA lecture titled *On Making Things the Best – Aeronautical Uses of Optimization*, optimization advocate Holt Ashley noted the "keen disappointment felt by many [optimization] specialists because their theories have received so little practical application." [29] Ashley went further by conducting an exhaustive industry-wide survey to identify "successful practical applications" of aircraft design optimization. The survey received "overwhelming" industry interest and encouragement; however, "the yield of examples which met the criterion of having been incorporated into a vehicle that actually operates in the Earth's atmosphere was painfully, perhaps shockingly small."

Perhaps the reason why Ashley's disappointment was so poignant is that, even at the time, aircraft designers in industry widely recognized the immense potential of MDO tools to fundamentally trans-

[1]"Moore's law" is an empirical observation that chip transistor density (a surrogate for computational power) has tended to double roughly every two years, a trend that has held true for the past half-century.

form engineering design workflows. As two Lockheed engineers stated in 1998 [39], “The technical community knows the power of MDO and not having a cradle-to-grave example has been a continual source of frustration, as voiced by AIAA MDO technical committee members [for] years.” Similarly, a 2002 Boeing paper identifies MDO as one of four key technologies poised to define the next generation of aircraft design[2] [40].

Motivated by this gap between promise and practice, Kroo and other luminaries offered several specific reasons for the lack of industry adoption of MDO technologies. Many of these remain familiar to modern researchers, and are worth revisiting here:

1. **Inadvertently-violated model assumptions:** When optimization is applied to an analysis toolchain, it acts adversarially, disproportionately seeking out the “weakest link” in the analysis chain and exploiting it. Simplified models that are acceptable in a manual sizing study are often unacceptable in an optimization study, because implied assumptions that an engineer would naturally cross-check during manual design are prone to optimizer exploitation. This can lead to unrealistic results from MDO tools that degrade user trust in optimization processes. Kroo contends that the solution to this problem is to implement higher-fidelity models that account for more edge cases, an approach we explore later in Section 2.3.1.

2. **Missing models and constraints:** Critical aircraft design choices are often determined by trade-offs spanning multiple disciplines. If an MDO tool does not incorporate relevant disciplines (e.g., it performs aerodynamic, propulsive, and structural analyses, but the true design driver is noise), then the resulting design will be fundamentally flawed with no indication to the user whatsoever. This can lead to costly design mistakes. Indeed, Kroo suggests that a MDO tool solving an ill-

[2]With the others being computational simulation, small uncrewed aircraft, and newly-emphasized design considerations such as environmental footprint and operations optimization.

specified design problem can lead to more harm than insight, due to the false confidence it can instill.

3. **Challenges of managing complexity:** As computational power grows, MDO tools can incorporate increasingly numerous and higher-fidelity models. To first order, when the number of models N grows, the potential number of cross-discipline couplings to manage tends to scale as $\mathcal{O}(N^2)$. Therefore, in the limit of growing computational power, the practical bottleneck is less about implementing individual analyses, but more about managing this communication overhead and architecting the optimization code itself.

Kroo primarily attributed these early barriers to practical industry adoption to the limited computational power of the era, noting: "This convergence between computational capability and computational requirements for interesting design problems is one of the reasons that MDO is considered to be such a promising technology, despite the limited acceptance of pioneering MDO efforts." A contemporaneous review by Sobieski and Haftka agreed, identifying "very high computational demands" as a "major [obstacle] to realizing the full potential of MDO". [31]

However, even at this time, some works recognized that not all challenges would recede with increasing computational power. A 2002 review paper by a Boeing Technical Fellow in *Journal of Aircraft* [40] cautions: "New [MDO] strategies need to be developed...which take advantage of the assumptions and techniques that airplane designers use, rather than letting a computer churn away and come up with theoretically possible, but practically impossible, configurations." Likewise, Kroo, Sobieski, and Haftka all cite "managing complexity" as a growing challenge of MDO [31, 32, 41], which hints at a human-computer interface problem, rather than merely a need for more CPU cycles. Drela's aptly-titled 1998 work *Pros & Cons of Airfoil Optimization* also demonstrates a similar issue [33]. Here, Drela shows that even seemingly-simple aerospace design optimization problems will only yield practical results if the problem formulation, assumptions, and results are all precisely understood by an expert designer. These works presciently foreshadow modern concerns around MDO interpretability and other user frictions, urging the development of human-centered MDO approaches that synthesize optimization performance with ease-of-use to allow rapid problem iteration.

In summary, while computational limits were a clear barrier in the early history of MDO, foundational challenges around managing complexity and aligning optimization with real-world design constraints were already emerging. These issues would become increasingly central as MDO research rapidly expanded in scope.

2.2 A Retrospective on Aircraft Design Optimization in Industry

Current Status

With the benefit of an additional quarter-century of hindsight since the date of these early MDO assessments, we can begin to assess how these forecasts have played out. In many ways, these predictions by Kroo and others were remarkably accurate. The scale and speed of optimization problems solved today has indeed grown exponentially in the years since. This is not only due to increasing computational power, but also from algorithmic and architectural improvements in MDO and optimization more broadly[3]. Numerous high-quality aircraft MDO case studies and post-hoc design studies have been published in the years since, including the D8 transport aircraft [3, 16], the STARC-ABL aircraft [42], and the Aerion AS2 [43]. Some of these studies leverage relatively-high-fidelity models (e.g., RANS CFD) that would have been computationally-intractable for optimization studies in prior eras.

[3]discussed later in Section 2.3

However, many of the barriers to practical industry use Kroo identified have not disappeared; to the contrary, as computational power and problem scale increase, the challenges of interpretability and managing complexity ring even truer today [7, 35, 44, 45]. As a result, this gap between academic research and practical industry adoption has not closed, and in some ways is wider than ever before. A 2010 review paper [34] concludes that "the actual use of genuine MDO methods within industry at large...is still rather limited." In 2013, Hoburg and Abbeel lamented that "despite remarkable progress in MDO, the complexity and diversity of modern aerospace design tools and teams makes fully coordinated system-level optimization a monumental undertaking." [46] In 2017, a team of Airbus engineers and MDO researchers concurred: "While the field of MDO techniques has tremendously grown since [the 1980s] in the scientific community, its application in industry is still often limited[...] A major challenge remains to apply MDO techniques to industrial design processes." [35] In 2020, van Gent et al. state that "Despite its promise, MDO is not as widely applied as was expected at its birth more than three decades ago." [47] As recently as 2021, Ozturk and Saab noted that "the uptake of new design tools in the aerospace industry has been low due to heavy reliance on legacy design methods and prior experience when faced with risky design propositions." [48]

Despite the prevalence of on-paper design studies using MDO, it remains surprisingly infrequent to see *built-and-flown* airplanes developed with documented, significant use of MDO methodologies. Indeed, the industry norm for aircraft conceptual design remains largely unchanged: expert-driven manual design guided primarily by point analyses and parametric surveys [49–51]. High-quality example aircraft design case studies that implement this expert-driven manual approach successfully are the Joby Aviation S2 eVTOL [52] and Perdix micro-UAV [53] aircraft. Here, the human designer informally fulfills an optimizer-like role, but the requirements are never explicitly translated into a formal mathematical optimization problem[4]. Anecdotally, this remains the most common conceptual design strategy in industry, and the *industry* conceptual designer's computational weapon of choice is more often an Excel spreadsheet than a formalized MDO framework.

It is important to highlight the benefits of this expert-driven manual design strategy. For example, in 2020, a Conceptual Design Principal Engineer at Lockheed-Martin SkunkWorks writes [51]:

> "One of the more successful projects paired one of our most experienced designers with a new-hire who had focused his education more on programming and multidisciplinary optimization than core air vehicle design. The Mentor took the time to help the Mentee understand the value of general aircraft design from an artistic perspective as opposed to a numerical optimization. In order to accomplish this, the Mentor spent a little extra time with the Mentee in the earliest stages of conceptual design, the sketching stage. By forcing the Mentee to quite literally drop the mouse, pick up a pencil and paper, and sketch all the different ways that he could think of that his airplane could look like and be configured, the Mentee had to start thinking about the aircraft as more than just a series of interconnected parametric equations and optimization routines."

This highlights the true requirement for a conceptual MDO framework to provide value in industry: it must be transparent enough to the end-user that it enhances, rather than obscures, a designer's artistic vision. A framework that demands that the end-user have significant expertise in computer science or applied math not only adds a barrier to entry, but also distracts from the ultimate goal of building and flying a real-world physical system. The most successful design processes in industry are those that leverage the strengths of both human and computer, rather than attempting to replace one with the other.

To assess the magnitude of this gulf between academic and industry use of MDO, we surveyed industry literature and design reviews to identify aircraft development programs showing three minimal criteria:

1. A documented design optimization problem for a complete aircraft, coupling at least three

[4]i.e., with a precisely specified objective, defined variables, and enumerated constraints

disciplines (e.g., aerodynamics, structures, and propulsion), that is solved computationally within an MDO framework (as opposed to manual iteration).

2. Evidence that the optimization result (or at least, some insight gained from it) was used to inform the design of aircraft.
3. Evidence that the aircraft was subsequently built and flown.

It is worth pausing to discuss why this final *built-and-flown* aircraft requirement is used, since it accounts for the majority of the aircraft design studies that were ultimately excluded from this survey. Successful optimization on built-and-flown aircraft forces a level of completeness, trust, and durability that may not be present in a paper study. Ashley notes that a practical MDO result "must also survive the gauntlet of ground testing, reliability, demonstration, flight verification, and the like without further special attention". [29] Myriad other practical considerations could be added to this list – certifiability, manufacturability, lifecycle costs, maintainability, robust off-design performance, and the realities of engineering culture (e.g., can the MDO tool give not only a result, but sufficient evidence that it should be believed?), to name only a few. To justify the substantial capital of detail-designing, building, and flying a new aircraft, optimization results must withstand a much higher standard of scrutiny and reviewability than they might otherwise.

Consistent with previous surveys and reviews [29, 31, 32, 34, 35, 41, 54], we find that relatively few programs meet these criteria; here, we make note of some that do. The Lockheed Martin F-22 Raptor program was one of the first programs to leverage an MDO-like process for a complete, flown aircraft design, as reported by Radovcich and Layton [39]. However, the workflow documented in this work differs significantly from how MDO processes are typically envisioned, in that it was remarkably manual – a fusion of traditional and MDO-based design processes. While some disciplines (aerodynamics, structures, and control law design) are coupled computationally, many other disciplines (low-observability, manufacturability, etc.) are coupled in by querying teams of subject-matter experts at each iteration within the optimization loop itself, essentially serving as a black-box function call. When contrasted with more-typical MDO processes where all interdisciplinary communication is computational, this manual approach has some pros and cons. One one hand, intermediate optimization iterates are continuously human-reviewed, which could cause a poorly-posed problem to be identified as such more quickly. On the other hand, it also sharply increases the optimization runtime, which may take months instead of hours. Even allowing for this broader definition of MDO, however, the authors note the rarity of their experience in industry: "Documented experiences of MDO applications during the engineering, manufacturing, and design phases of fighter aircraft programs are not numerous. Documentation is even rarer for aircraft that have flown."

The Lockheed Martin X-59 Quiet Supersonic Transport (QueSST) is another program that leveraged an MDO-based design process throughout aircraft development, unifying aerodynamics, acoustics, and structural analyses for outer mold line design and composite ply scheduling[5] [55–57]. The X-59 program builds upon several decades of successful MDO research for sonic-boom-minimization problems, a topic where computational shape optimization has proven particularly useful due to the non-intuitive and sensitive nature of the design problem [58].

The Airbus *Vahana*, a single-seat eVTOL demonstrator flown in 2018, is perhaps one of the most recent public examples of an MDO-based process used to develop a flown aircraft [59–61]. The initial sizing study considered aerodynamics, structures, propulsion, and cost analysis to drive conceptual trade studies between various vehicle configurations. This study included practical constraints and margins,

[5]Although the X-59 has not yet met the "flown" criterion at the time of writing, industry reports credibly suggest this is imminent with no remaining obvious barriers.

such as reserve mission energy, battery cycle life, and engine-out safety (by enforcing either autorotation capability, motor redundancy, or a mass allocation for a ballistic parachute, depending on configuration).

Several other programs have built and flown uncrewed research aircraft with MDO use, such as the Facebook *Aquila* solar-powered UAV [62], the MIT *Jungle Hawk Owl* long-endurance UAV [63], and the X-48B blended-wing-body demonstrator [64–66]. Other programs document MDO usage for narrower subsystem- or component-level design, as in the detailed design of the Boeing 787 Dreamliner [34].

Outside of these cases, documented *industrial* use of MDO for complete-aircraft conceptual design remains uncommon, relative to the vast number of industry programs conducted in the past few decades. This observation is especially surprising in light of the widely-recognized potential utility of MDO in industry [40, 45].

One compelling partial explanation for this lack of use is a lag effect stemming from long timelines of new aircraft development programs – it takes time for new tools (such as MDO methods) to proliferate throughout industry, due to sunk-costs on design pipelines for existing programs. Indeed, although the present literature review finds that documented industry use of MDO is scarce, the situation appears somewhat less dire than older surveys indicate [29, 31, 32, 34, 35]; this suggests that industry use may be gradually increasing. However, considering that aircraft MDO research and industry interest in optimization has existed for over forty years [29, 30], this lag effect alone does not constitute a complete explanation for the lack of adoption.

Ashley considers another possible explanation for the lack of visible MDO use in industry: that of "military classification or company proprietary considerations." [29] However, after exhaustive correspondence with dozens of aircraft design industry contacts, Ashley concludes that this limitation was only occasional and "not to an extent that would affect the principal conclusions." Therefore, it seems that the limited observations of formal mathematical optimization processes in industrial aircraft design are genuine, representing room for further improvement in closing this academia-industry gap [35, 47, 54, 67, 68].

2.3 Pivotal Advances in Design Optimization Research

On a more optimistic note, MDO's substantial academia-industry gap is equally attributable to academia's remarkable advances in design optimization research over recent decades. This progress is readily apparent from analyzing trends in academic literature. As shown in Figure 2-1, the fraction of aircraft design publications directly referencing MDO has grown substantially over the past 40 years.

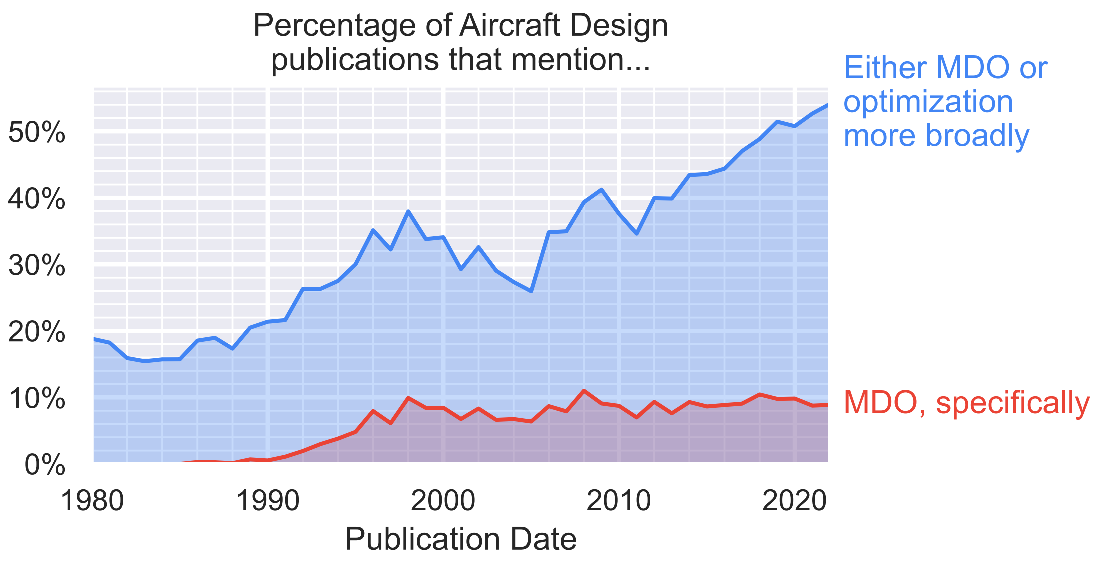


Figure 2-1: Prevalance of optimization-related keywords in academic literature with the keyword "aircraft design". Data from Google Scholar; includes industry-standard texts such as AIAA journals and conference proceedings. MDO-specific keywords (red line) include any of: "multidisciplinary design optimization", "MDO", and "MDAO". The blue line adds the keyword "optimization" to this list.

This statistic alone understates the true magnitude of MDO's impact. Arguably the most pivotal contribution of MDO research to aircraft design has been to catalyze a shift towards an *optimization* *mindset*: the recognition that optimization is not only a useful tool, but also a principled mathematical framework that can collectively represent both a design space and the engineer's design intent. Figure 2-1 shows that this mindset is a relatively recent development – the fraction of aircraft design publications mentioning optimization in any form has tripled since the advent of MDO, reaching over 50% today.

These optimization-based design processes have shown several benefits when used to augment traditional design methods such as point analyses, parametric surveys, and carpet plots [45, 51]. Most obviously, formal optimization methods can lead to improved design results and allow the consideration of many more design variables. Optimization can also help human designers discover clever cross-discipline synergies that might otherwise be overlooked [33], and its rigor can reduce the likelihood of biased decisions [45]. In cases where designer intuition is exhausted, such as with unconventional configurations, optimization provides a means to identifying useful directions for further exploration [33]. Torenbeek argues that an optimization-based design process can respond more quickly to unexpected changes in program requirements, whereas manual processes often require substantial redesign effort [45]. Finally, the optimization mindset itself has benefits, even beyond the results of an optimization study. for example, discussions about problem formulation (how to translate given requirements into a quantified optimization problem) can help a team of engineers align on design goals and expose subtle discrepancies in perceived requirements[6].

Another important area where MDO research has made significant progress is in defining the relationship between the human designer and the optimizer. The prevailing view in the early days of aircraft design optimization was that optimization would eventually advance to the point that computational tools could yield complete, production-ready designs with minimal human oversight – as evidenced by

[6]For example, is the objective to minimize fuel burn, or to maximize range? Is operating cost a quantity that should be strictly capped (a constraint), or instead only penalized (an objective)?

encouraging titles such as “Automating the Design Process” [30, 69, 70]. Over time, this has largely given way to a more balanced view: although the optimization solve itself may be well-served by computational means, this forms only one small part of the larger design process. Inputting the engineer’s design intent into the optimizer accurately (“problem specification”) and extracting intuition from the results (“interpretation”) are challenges in their own right, and best addressed by iterative human-computer teams. As described by Drela in a 1998 optimization study [33], “[Engineering optimization] is still an iterative cut-and-try undertaking. But compared to [traditional] techniques, the cutting-and-trying is not on the geometry, but rather on the precise formulation of the optimization problem.”

When considering this full design process (i.e., from initial requirements to product delivery), the scope of challenges becomes even broader. Indeed, industry adoption of MDO is often limited by non-computational user frictions [7, 36, 71]. In the present work, we deliberately term these user frictions “costs”, borrowing optimization nomenclature to emphasize that minimizing these frictions is an implied part of the objective function of any optimization-driven design process. Figure 2-2 summarizes a variety of design literature to present a holistic view of this complete process as applied to aircraft design [36, 45, 50, 71–74]. Frictions that a designer may experience from a computational optimization framework at each step are shown in red.

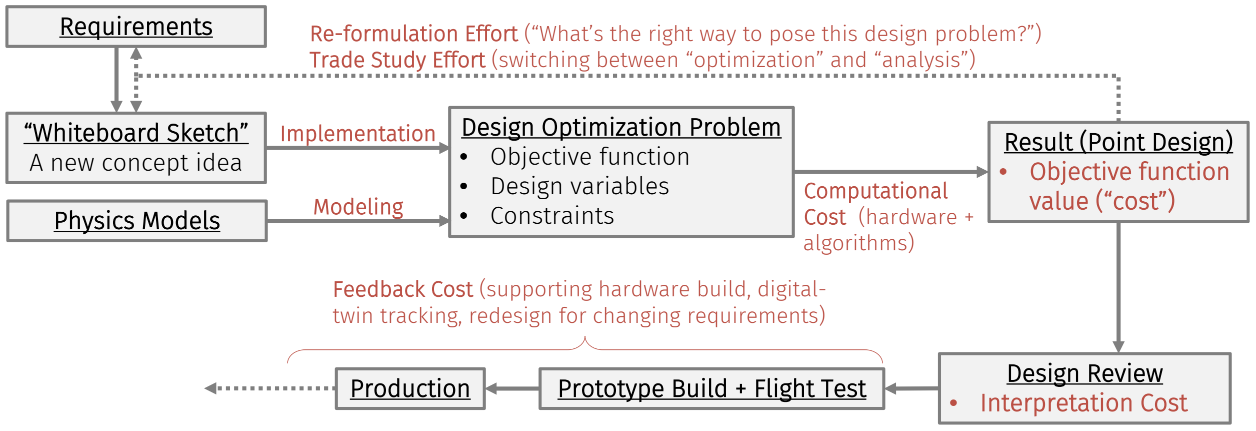

Requirements
Re-formulation Effort ("What's the right way to pose this design problem?")
Trade Study Effort (switching between "optimization" and "analysis")
"Whiteboard Sketch"
A new concept idea
Implementation
Design Optimization Problem
Objective function
Design variables
Constraints
Result (Point Design)
Objective function value ("cost")
Physics Models
Modeling
Computational Cost (hardware + algorithms)
Feedback Cost (supporting hardware build, digital-twin tracking, redesign for changing requirements)
Production
Prototype Build + Flight Test
Design Review
Interpretation Cost

Figure 2-2: A high-level overview of the optimization-driven design process, as applied to aircraft design. "Costs" (user frictions to be minimized) associated with each step are shown in red.

In the aircraft design process model of Figure 2-2, the initial point of departure is a set of high-level requirements [45, 73]. The first design step is to develop a set of candidate concepts ("whiteboard sketches"), a qualitative and configuration-focused process that largely leverages designer creativity and experience [72, 74]. Concepts and the associated requirements are then translated into a formal mathematical design optimization problem consisting of an objective function, design variables, and constraints. Physics models are invoked as needed to support this process; this modeling process incurs a cost that depends on the modeling flexibility of the chosen MDO paradigm [1]. The optimization problem is then solved, which incurs a computational cost and yields a point design.

This process of mapping potential concepts to problems and problems to optimized point designs is often iterative. Even for expert users, it is rare that a design optimization process will fully reflect design intent on the first attempt [33]. In addition, post-optimality studies may be used to assess performance robustness to off-design conditions. The total time required to close this re-formulation loop is a critical driver of how the human interacts with the MDO tool, as it directly rate-limits how much design exploration can be performed.

For a practical aircraft development program, however, this is only the beginning. The point design must survive external validation and design review by a team of subject-matter-experts; the cost of this process is a direct function of the tools an MDO framework supplies to aid result interpretability. Even beyond this, in prototyping, flight testing, production, and deployment, the design process still imposes framework requirements. For example, an analysis and optimization framework that supports a wide range of modeling fidelities has the potential to can smoothly transition from a conceptual design tool to a digital thread [75, 76]. Likewise, a design tool can be used during the production process to support manufacturing decisions, such as whether an unexpected change in a component supplier still produces a feasible design with desired margins.

The remainder of this chapter will discuss some of the most important advances in optimization, design processes, and scientific computing in recent decades, with a focus on how these advances can reduce the costs incurred at various steps of this holistic process.

2.3.1 Two Directions: "Wide" vs. "Deep" MDO

Before proceeding further, it is important to clarify the scope of aircraft design optimization problems that are of interest in this thesis, as the definition of "MDO" is somewhat overloaded. More precisely, the modern field of MDO can be decomposed into two related but distinct subfields which have developed different strategies and mindsets to address different design needs [68]. Figure 2-3 illustrates this distinction, showing how the term "MDO" can refer to separate tasks at different stages of the aircraft design process.

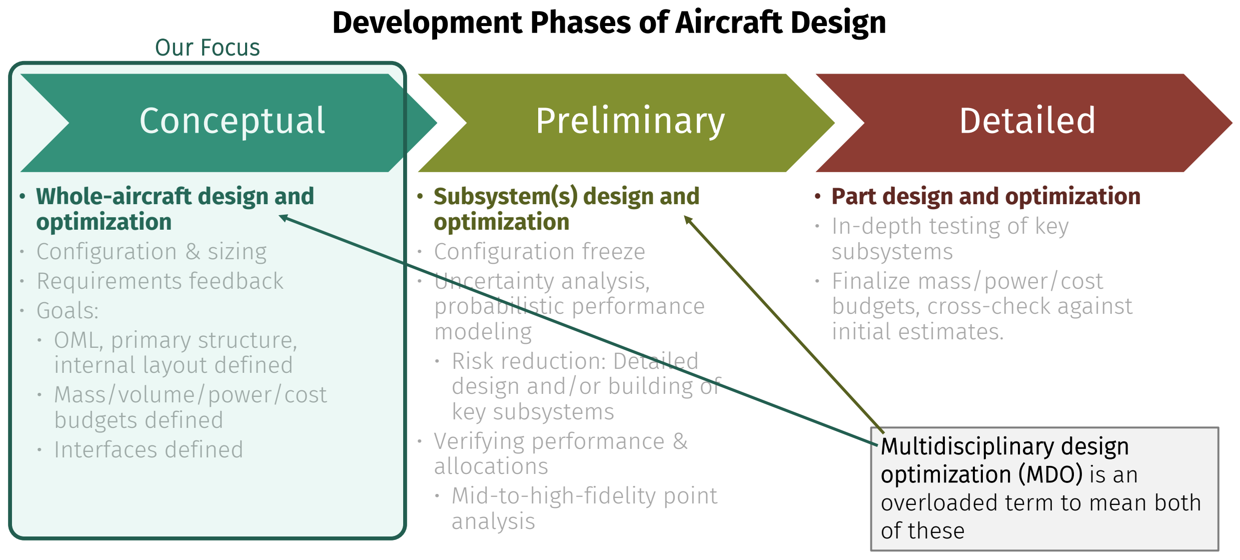

Development Phases of Aircraft Design
Our Focus
Conceptual
Preliminary
Detailed
Whole-aircraft design and optimization
Configuration & sizing
Requirements feedback
Goals:
OML, primary structure, internal layout defined
Mass/volume/power/cost budgets defined
Interfaces defined
Subsystem(s) design and optimization
Configuration freeze
Uncertainty analysis, probabilistic performance modeling
Risk reduction: Detailed design and/or building of key subsystems
Verifying performance & allocations
Mid-to-high-fidelity point analysis
Part design and optimization
In-depth testing of key subsystems
Finalize mass/power/cost budgets, cross-check against initial estimates.
Multidisciplinary design optimization (MDO) is an overloaded term to mean both of these

Figure 2-3: Multiple definitions of the term "multidisciplinary design optimization". Here, conceptual, or "wide" MDO is contrasted against preliminary, or "deep" MDO.

The first subfield, and the subject of the present work, is that of **conceptual**, or **"wide" MDO**. This is essentially a formalization of the conceptual sizing problem and typically involves casting a wide net to capture as many first-order dependencies as possible. The intended use case is often clean-sheet design of a complete aircraft, and the end goals are often initial identification of strong design drivers and assessment of concept feasibility. Sensitivity analysis is also a key desired output of a wide MDO process. Frameworks that tend to specialize on this class of wide MDO problems are TASOPT [16] and GPkit [7].

These goals requires an enormous breadth of models to be considered, as practical aircraft designs are shaped by an enormous number of constraints that do not appear in first-principles performance analysis (e.g., the Breguet range equation). For example, a drag reduction of the vertical stabilizer is of little interest if engine-out performance constraints preclude certification; likewise, a more efficient engine is of little interest if acoustic constraints preclude customer acceptance. This need for analysis breadth can lead to design problems with many hundreds of models spanning anywhere from five to twenty relevant disciplines. To illustrate this breadth, consider the following non-exhaustive list of example disciplines that might be included:

- Aerodynamics
- Structures and weights estimation
- Propulsion
- Power systems and thermal management
- Flight dynamics (trim, stability, and control)
- Internal geometry and packaging
- Mission design (concept of operations) and trajectory optimization
- Takeoff and landing field length analysis
- Certifiability, safety, and engine-out performance
- Life-cycle emissions
- Cost modeling and manufacturability
- Acoustic signature
- Ride quality
- Survivability and stealth

Each of these disciplines might include dozens of submodels [16, 45, 73, 77]. To keep problems tractable despite their large scope, these models are typically low-fidelity; often, they are either derived from first principles physics with appropriate simplifications or regressions to statistical data. Despite the reduced accuracy of these models, they fulfill two critical functions. First, they apply *optimization pressure* to steer the optimizer towards realistic designs, often in subtle, interdisciplinary ways. For example, increasing fuselage length requires increased landing gear length and mass to maintain a proper takeoff rotation angle, reducing the attractiveness of such design changes. Secondly, these low-fidelity models provide information on which constraints might be active near the optimum, allowing computational resources to be devoted to fidelity improvement only where it is needed. Often, these wide MDO problems are posed as large single-level optimization problems, which allows the large number of cross-disciplinary constraints to be computationally represented in a shared namespace [46].

The second subfield of MDO is that of **preliminary**, or **"deep" MDO**. Here, the goal is often to provide detailed design refinement of a very close initial guess. Because the low-hanging fruit has often been picked by this stage, the design problem is often more local in nature, and the number of models is typically smaller than in wide MDO. Likewise, continued improvement from a good initial guess requires high-fidelity models, since the objective function tends to be relatively flat near the optimum and hence highly sensitive to model inaccuracy[7]. For example, RANS-based CFD[8] models are often used for aerodynamics, at significant cost: optimization runtimes of 1,000 to 100,000 CPU-hours are not uncommon [78]. To maintain tractability, relatively few disciplines are considered – often only aerodynamics and structures. The intended use case is often detailed design of a single component or subsystem, such as a wing. Due to computational challenges that tend to be more prevalent in high-fidelity models (e.g., PDE-constrained optimization, adjoint-based gradients, numerical stiffness of models, and convergence of models that use iterative solvers), the subfield of "deep" MDO has placed a large focus on MDO architectures that allow multi-level decomposition of the original optimization problem [28]. Frameworks that tend to specialize on this deep MDO subfield are MACH-Aero [79], and, to some extent, OpenMDAO [9].

While both subfields share broad similarities and provide useful design insight, they are fundamentally attacking different problems: wide MDO is aimed at clean-sheet conceptual design, while deep MDO tends to aim at detailed refinement of a close initial design. Historically, this schism became increasingly apparent roughly in the 1990s and 2000s, as various research groups began "spending" their increasing computational budgets in different ways: either model breadth or model depth. Examples of wide MDO research efforts in the time since this divergence include PASS [44] by Kroo, TASOPT [16] by Drela, and various GPkit-based [46] aircraft design codes. Modern ideas around deep MDO research can trace their origins to high-fidelity adjoint-based aerodynamic shape optimization studies by Alonso,

[7] A similar effect also occurs with constraint equations.
[8] Reynolds-averaged Navier-Stokes; Computational fluid dynamics

Martins, and other contemporaries [58, 80, 81].

2.3.2 Advances in Optimization Algorithms

The current academic state-of-the-art for MDO paradigms has largely centered on two approaches: gradient-based methods with analytical gradients, and disciplined optimization methods. The former tends to be more popular in deep MDO applications, while the latter tends to be more popular in wide MDO applications; however, exceptions exist. These two existing approaches are defined and discussed in turn below. Pros and cons of these existing approaches, as compared to the proposed new paradigm, are summarized in Table 3.1 of the subsequent chapter and discussed fully in Appendix A.

Gradient-Based Methods with Analytical Gradients

Gradient-based optimization offer fundamental scaling advantages over gradient-free methods, and are hence preferred for large-scale design optimization when model characteristics allow [71, 82]. The effectiveness of gradient-based methods is highly contingent on the speed and precision of gradient calculations. To illustrate this, Kirschen and Hoburg show that gradient-based optimization gives inadequate performance on even modest engineering design problems if simple finite-differencing is used [83]. Likewise, Adler et al. show that the inaccuracy of finite-differenced gradients can prevent accurate optimization [84].

Because of these considerations, a leading academic approach to gradient-based MDO involves manually supplying exact analytical gradients to the optimizer for each submodel. Frameworks like OpenMDAO and MACH-Aero [9] provide example implementations of this bring-your-own-gradients approach, forming the basis for several interesting case studies [42, 43, 85, 86]. Although this paradigm is most often applied to deep MDO problems (e.g., high-fidelity aerodynamic shape optimization), it has been applied to wide MDO problems in examples such as OpenConcept [85] and OpenAeroStruct [86].

Disciplined Optimization Methods

Another MDO paradigm that has become popular in recent years is tht of disciplined optimization methods, borrowing nomenclature from Grant [87], Agrawal [88], and Boyd [89, 90]. These methods are considered *disciplined* as they impose a strict set of rules on the optimization problem formulation, which allows the optimizer to make assumptions about the problem structure (such as convexity, or log-convexity). These assumptions allow the optimizer to make more efficient use of computational resources and provide guarantees of global optimality. Geometric programming in particular has proven to be a useful disciplined MDO paradigm for aircraft design applications, as originally observed by Hoburg [46]. GPkit [7, 46, 83], a framework that demonstrates this geometric programming concept targeting engineering design applications, has led to successful case studies such as the Jungle Hawk Owl aircraft [63] and Hyperloop system studies [7].

2.3.3 Other Advances in Design Optimization Practice

A recent notable shift in aircraft design optimization perspectives has been a renewed focus on geometry representation and parameterization. For example, Haimes and Drela [91] contended in 2012 that "constructive solid geometry (CSG) is the natural foundation for [aircraft design optimization]". This publication extended prior work by Lazzara, Haimes, and Willcox [92] that introduced the concept of "multifidelity geometry". Together, these works advocated for accurate 3D outer mold line representations from the earliest stages of conceptual design, with degenerate representations of this central geometry used to drive individual discipline analyses. Furthermore, they recognized the fundamental limitations of general-purpose (e.g., Parasolid-based) CAD tools in aircraft design optimization: because aerospace geometries tend to use complex, lofted surfaces, the resulting CAD models can be brittle with respect to parameter changes. This observation, along with later developments such as differentiable discretization techniques [93], led to renewed research on how aircraft geometry should be parameterized for optimization. For example, in recent years aircraft-specific geometry tools like OpenVSP [94] and Engineering Sketch Pad [93] have been developed for design optimization workflows, a trend that arguably stems from this line of research.

In a broader sense, another notable area where computational methods for aircraft design have long made inroads to industry is in inverse design tools. (An example of successful industry adoption in aerodynamics the inverse approach used by the XFoil airfoil design code [95] and others [65] to recover a shape from a specified pressure distribution.) Like optimization methods, these inverse methods can aid designers because they offer fundamentally different capabilities than traditional design methods (e.g., carpet plots). While traditional methods focus on the "forward problem" (design $\rightarrow$ performance), inverse methods and optimization methods both focus on the "inverse problem" (performance $\rightarrow$ design). Drela [33] shows that this more-manual inverse approach can lead to more robust designs than optimization, if the user is not familiar with the risks of an improperly-specified optimization problem.

2.3.4 Advances in Numerical Methods

In the past two decades several notable scientific computing techniques have matured to the point of practical use in engineering design optimization. Two of the most significant advances are automatic differentiation and automatic sparsity detection, which are discussed in detail below. A review focusing on these two techniques is available in work from Andersson et al. [96] and Rackauckas [97]. Review of several other notable scientific computing advances, such as GPU-accelerated computing, probabilistic programming, deep learning, and uncertainty quantification, are omitted here for brevity. A wider review of state-of-the-art scientific computing techniques is available from Lavin et al. [98]

Automatic Differentiation

Here, we review literature on *automatic differentiation*, an advanced technique borne out of the machine learning community that fundamentally accelerates gradient-based optimization algorithms. A comprehensive modern survey is available by Baydin et al. [99]. A 2008 textbook by Griewank and Walther [100] provides a more in-depth review of the mathematical foundations of automatic differentiation.

Gradient-based optimization methods are computationally bottlenecked by the cost of computing gradients [71, 82]. Existing state-of-the-art workflows have users manually derive and implement highly-efficient gradients for each model, but this is time-consuming and thus impractical for early-stage design. Automatic differentiation (AD) is an efficient and accurate way to automatically compute the gradients of a function *as represented in code* at runtime. In many cases, the computational complexity of this technique is fundamentally better than traditional approaches like finite-differencing; Griewank provides concrete upper bounds on the worst-case time complexity [101].

Automatic differentiation exploits the fact that any function can be broken into a series of atomic mathematical operators, which can then be represented as a computational graph. These atomic operators, sometimes referred to as "primitives", may consist of mathematical operations as small as a scalar addition or as large as a solution of a nonlinear PDE (e.g., continuous adjoint methods in PDE-constrained optimization). As a practical matter, this computational graph is often created by tracing program execution with an overloaded numerics library [102]. This graph construction process is illustrated in Figure 2-4. The chain rule can then be applied to this graph to compute the Jacobian of the function. Various modes of automatic differentiation effectively allow this Jacobian construction to occur either column-by-column (forward-mode) or row-by-row (reverse-mode) [71, 96, 103]; depending on the shape of this Jacobian, one strategy may be enormously more efficient than the other.

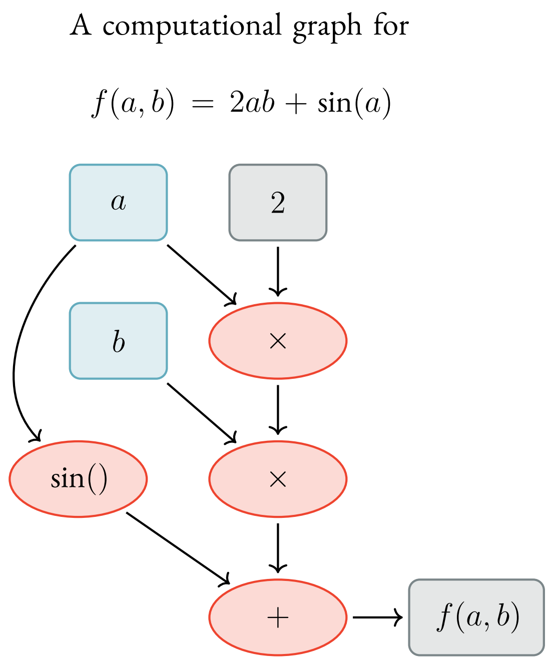


Figure 2-4: A computational graph, as would be used for automatic differentiation. Reproduced from Sharpe [1].

Advances in automatic differentiation (and in particular, combining this with GPU computing) are largely responsible for the revolution of practical advances in machine learning in the past decade [99]. Rackauckas in 2021 [97] states: "[Automatic differentiation] has become the pervasive backbone behind all of the machine learning libraries. If you ask what PyTorch or Flux.jl is doing that's special, the answer is really that it's doing automatic differentiation over some functions." The position of the present work is that similar scientific computing techniques can also unlock substantial practical advances in engineering design optimization.

Automatic Sparsity Detection and Jacobian Coloring

The process of gradient computation for optimization can be further accelerated if the sparsity of the constraint Jacobian is known a priori. This is because sparsity can guarantee structural independence of multiple columns[9] of the Jacobian, so the respective elements of the Jacobian can be evaluated simultaneously. Sparse Jacobians arise frequently in the modeling of physical systems, especially multidisciplinary ones [104]; because of this, this represents a significant opportunity in engineering design. The process of reducing the number of Jacobian evaluations by simultaneously evaluating independent columns is known as *Jacobian compression* and is illustrated in Figure 2-5. To enable compression, independent columns of the Jacobian must be grouped. This process is known as *Jacobian coloring*, referencing mathematical similarities to the graph coloring problem if the Jacobian's sparsity pattern is represented as a bipartite graph [105]. This coloring problem does not readily admit exact solution for Jacobian sizes of interest; however, Gebremedhin et al. provide a comprehensive review of graph coloring heuristics that work well in practice [2]. Martins and Ning provide a more concise summary that contextualizes this process around the backdrop of design optimization [71].

[9] or rows, if using reverse-mode automatic differentiation

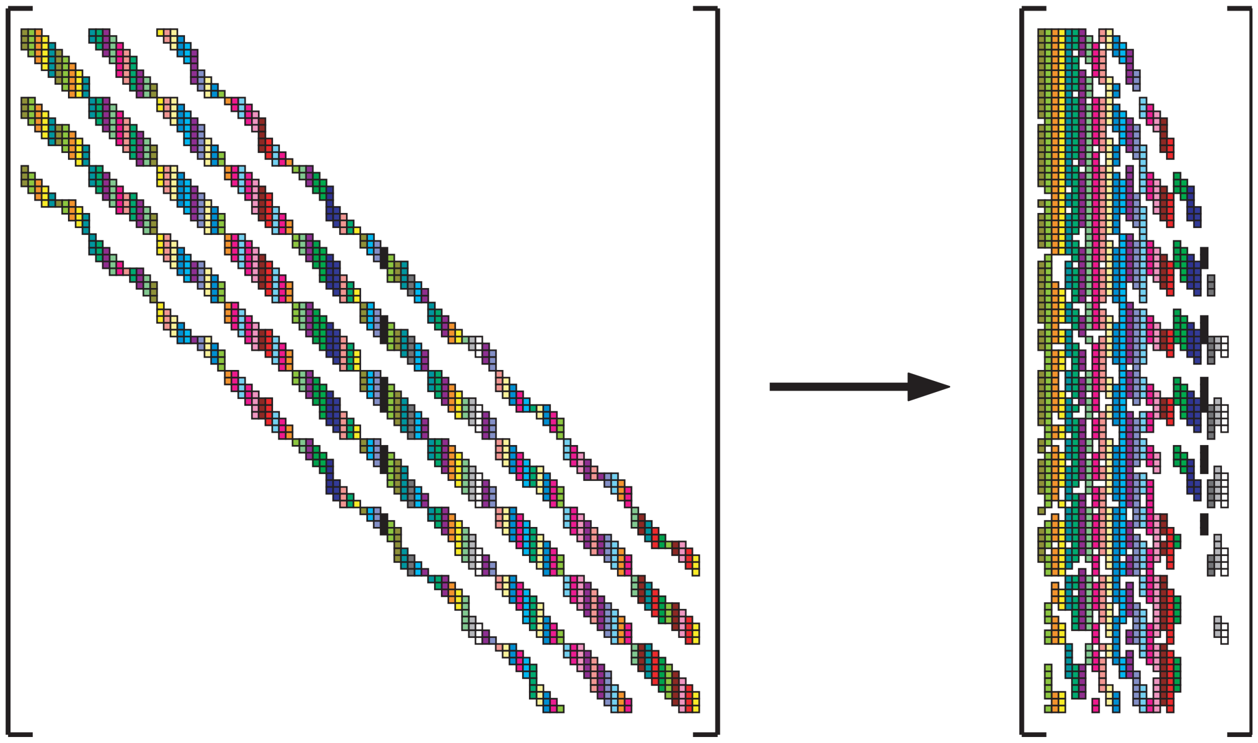


Figure 2-5: A sparse Jacobian can be evaluated more quickly by simultaneously evaluating derivatives that are structurally independent, a process known as Jacobian coloring and compression. On the left of the figure is the original Jacobian; on the right is its compressed form. Figure reproduced from Gebremedhin et al. [2]

Before Jacobian coloring and compression can be performed, the sparsity pattern of the Jacobian must be known. This is a nontrivial task and an area of active research, with the major complications summarized by Rackauckas [97]. In recent years, automated sparsity detection has enabled automatic construction of sparsity patterns using a tracing approach that is quite similar to that of automatic differentiation. This bypasses many of the potential edge-case failure modes that can occur when trying to infer sparsity purely by inspecting the dense Jacobian, such as conditional branching or cancellation. Indeed, the same computational graph can be used for this process, making this a computationally-efficient addition to a code base that already leverages automatic differentiation. Andersson et al. [96] provide an example implementation of this automated sparsity detection within the CasADi framework. Gowda et al. provide a more complete review on state-of-the-art methods in automated sparsity detection

[106], with state-of-the-art implementations of this by Ma et al. [37].

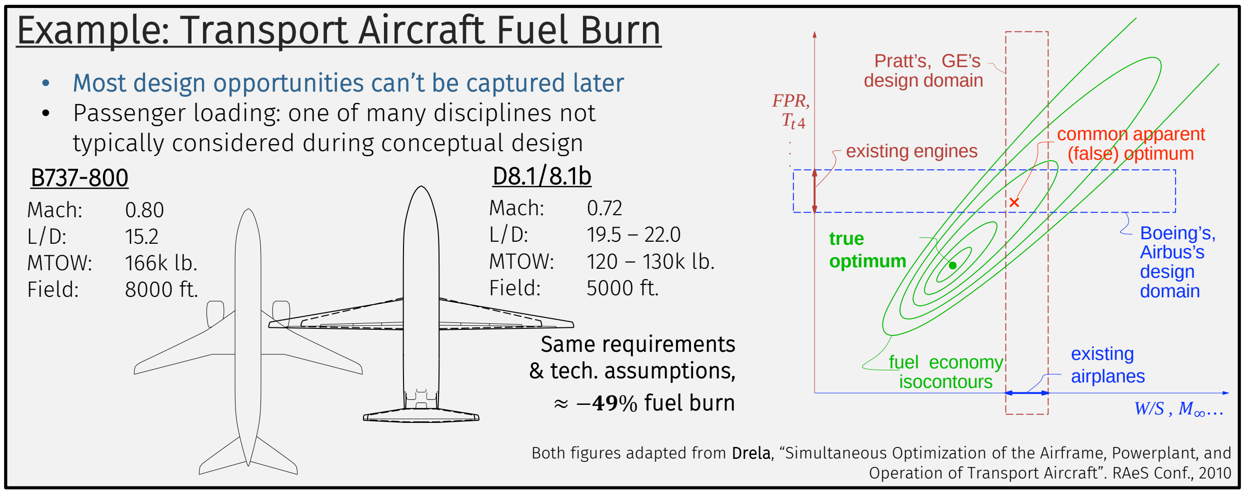
2.4 Motivations for Improving Accessibility of Computational Design Optimization

This thesis focuses on improving the accessibility of advanced optimization methods for industry practitioners, with a particular focus on techniques applicable to conceptual ("wide") MDO for aircraft. This focus is motivated by several factors.

Motivation 1: A Majority of Performance, Risk, and Cost is Committed Early

First, the vast majority of the performance of the final design is determined by early-stage conceptual design decisions. This can manifest in both a positive and a negative sense: after the conceptual design is frozen, most design opportunities cannot be captured later and most design regrets cannot be fixed. Figures 2-6 and 2-7 illustrates the two sides to this coin with practical examples.

The first example, shown in Figure 2-6, reproduces conceptual design work on the D8 "Double Bubble" transport aircraft configuration by Drela [3, 4]. Among other technology improvements, the configuration leverages a strategy of improving passenger loading/unloading speed, which allows for a reduced cruise Mach number while retaining comparable door-to-door travel times (a surrogate for passenger acceptance). This synergistic strategy creates a feedback loop which dramatically reduces fuel burn compared to designs that are optimized on a per-component decomposition basis. For example, one of Drela's main observations from a 2010 work [4] is that "multi-discipline optimization considerably increases the fuel savings compared to single-discipline optimization."


Example: Transport Aircraft Fuel Burn

- Most design opportunities can't be captured later
- Passenger loading: one of many disciplines not typically considered during conceptual design

B737-800	
Mach:	0.80
L/D:	15.2
MTOW:	166k lb.
Field:	8000 ft.

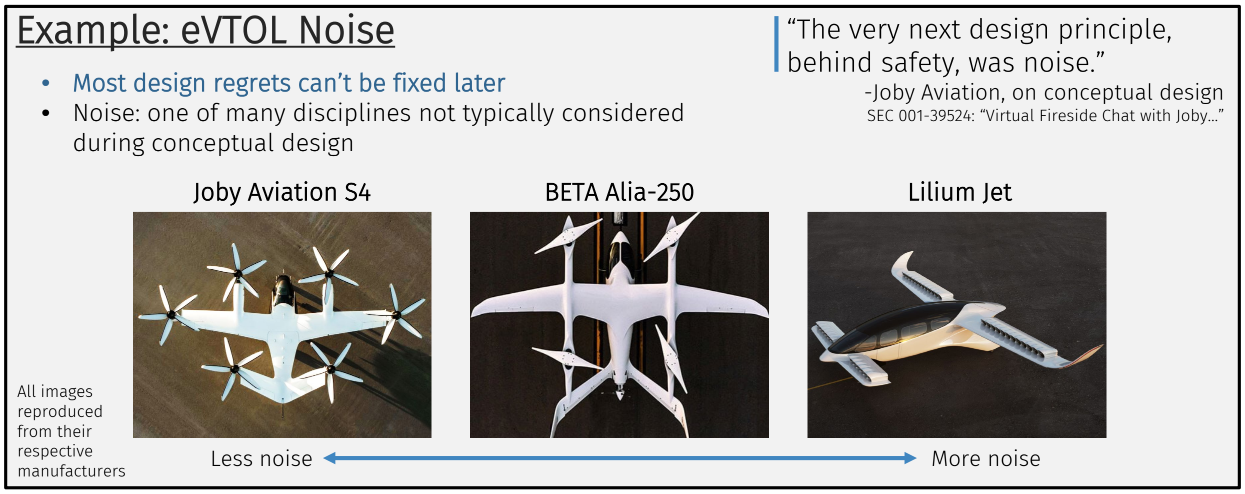
D8.1/8.1b	
Mach:	0.72
L/D:	19.5 – 22.0
MTOW:	120 – 130k lb.
Field:	5000 ft.

Same requirements & tech. assumptions, ≈ −**49**% fuel burn

Both figures adapted from **Drela**, "Simultaneous Optimization of the Airframe, Powerplant, and Operation of Transport Aircraft". RAeS Conf., 2010

Figure 2-6: In an example from Drela [3], large potential fuel savings are available for transport aircraft if cross-discipline couplings are considered early in the conceptual design process. Figure elements reproduced from Drela [4].

In a second example, shown in Figure 2-7, the other side of this effect is shown. Here, various electric vertical-takeoff-and-landing (eVTOL) aircraft design concepts are compared. Given the significant energy limitations of battery-powered aircraft, much of the focus of eVTOL conceptual design in the early 2010s was on traditional unit-economics performance metrics of range and payload fraction. However, as these vehicles begin to approach FAA certification and initial deployment, significant regulatory concerns have been raised about the community noise impact of such vehicles. Aeroacoustic noise, to first order, is a strong function of propulsor disk loading and blade tip speed [107], factors which require substantial redesign to modify later. Hence, manufacturers have had varying levels of difficulty in adapting to this renewed focus on acoustics, as this metric is largely locked in from conceptual design decisions made many years ago.


Example: eVTOL Noise

- Most design regrets can't be fixed later
- Noise: one of many disciplines not typically considered during conceptual design

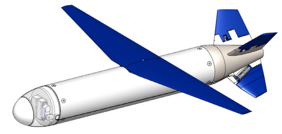
"The very next design principle, behind safety, was noise."
-Joby Aviation, on conceptual design
SEC 001-39524: "Virtual Fireside Chat with Joby..."

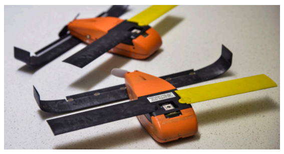
Figure 2-7: Design regrets may arise if conceptual MDO studies do not include all relevant disciplines. This example of contemporaneous eVTOL designs assesses aeroacoustic noise, a factor that is largely determined by conceptual design decisions made many years prior. Although noise is not traditionally included in conceptual aircraft design relations, it is crucial for certification and acceptance. Thus, including many disciplines in conceptual design – even non-traditional ones, like acoustics – is key for avoiding major downstream design regrets.

Motivation 2: New Technologies Reignite a Need for Early-Stage Design Exploration

A second motivation for the focus on conceptual MDO is that recent technological shifts have significantly expanded the aircraft design space compared to previous eras, calling for new tools to aid in down-selecting design concepts.

As a first example shown in Figure 2-8, miniaturization and uncrewed aircraft have opened up new trade spaces to explore, calling for renewed focus on first-principles early-stage design optimization. These micro-drone aircraft are able to take more risks with exotic designs, due to shorter design cycles, lower cost, and reduced certification and safety risks. New disciplines, such as packaging and folding considerations, drive new trades – in many cases, volume allocation trades can be as sensitive as mass fraction trades in conventional aircraft. Another new trade is that of size, weight, and power (SWaP) allocations to autonomy and onboard computing. For small air vehicles, flight computer power requirements can equal or exceed propulsive power, creating new incentives that tilt the design space towards control simplicity [108]. New missions, particularly those that are "dull, dirty, and dangerous" are enabled by uncrewed aircraft with attritable design and cost optimization. Finally, physics scaling laws, like the square-cube law and transitional Reynolds numbers, cause unique concerns for small-scale aircraft that warrant first-principles conceptual design optimization.


(a) MIT Firefly, a Mach 0.8, rocket-propelled micro-UAV [109, 110]


(b) MIT Perdix, an air-launched ALE-55-class ISR UAV [53]

(c) Transonic DP, a 545-mph dynamic soaring glider with 100 G turn capability (photo: Spencer Lisenby)

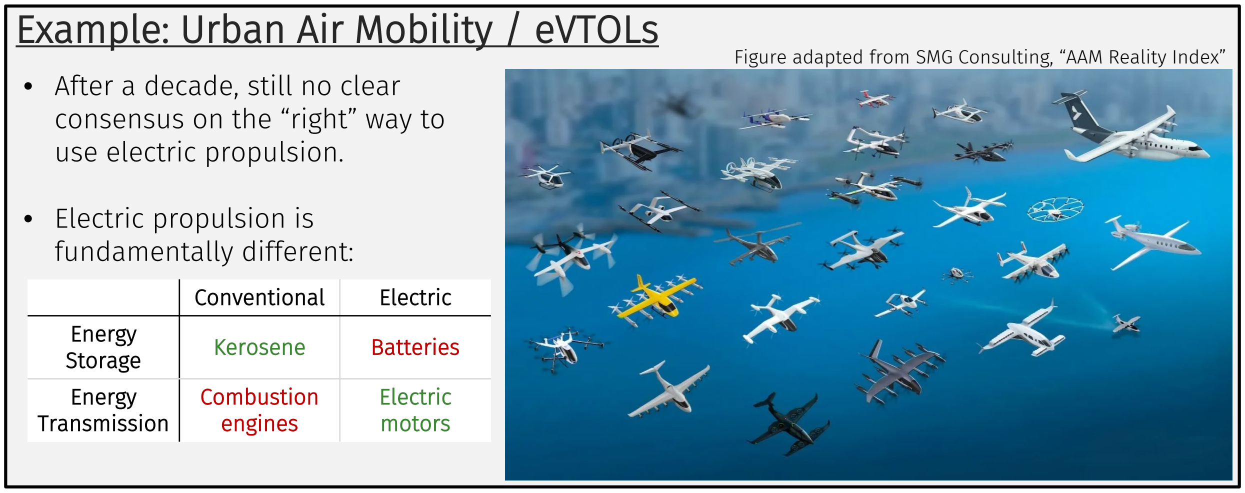
(d) Black Hornet Nano, an 18-gram ISR helicopter (photo: Richard Watt/MOD)

Figure 2-8: Examples of new design spaces enabled by miniaturization and uncrewed aircraft technology in recent years.

Another prominent new design space that motivates revisiting first-principles conceptual aircraft design is electric aviation, as shown in Figure 2-9. Electric propulsion offers a fundamentally different configuration trade space compared to conventional propulsion. In terms of energy storage, modern lithium batteries have realizable[10] specific energies that are roughly 1/25th that of kerosene, placing extreme sizing demands here. On the other hand, electric motors have excellent specific power across a wide range of size scales, dramatically fewer moving parts, and wider power bands compared to combustion engines. These factors make electric motors far more modular, essentially allowing designers to place propulsors at will – electric aircraft with eight or more propulsors are not uncommon. The enormous design space that electric propulsion opens up is evident in Figure 2-9, where the diversity of

[10]in other words, after accounting for the generally-higher powertrain efficiencies of electric propulsion due to the lack of a thermodynamic gas cycle

aircraft configurations is wholly unprecedented compared to other aerospace domains. Part of the value proposition of conceptual MDO is that it can make rigorous design within this large space more tractable.



Example: Urban Air Mobility / eVTOLs

Figure adapted from SMG Consulting, "AAM Reality Index"

- After a decade, still no clear consensus on the "right" way to use electric propulsion.
- Electric propulsion is fundamentally different:

	Conventional	Electric
Energy Storage	Kerosene	Batteries
Energy Transmission	Combustion engines	Electric motors

Figure 2-9: Electric propulsion opens up a much larger aircraft configuration space, since electric motors offer higher specific power and modularity than combustion engines. A focus on early-stage conceptual design is critical for down-selecting from this diversity of configuration options. Figure adapted from SMG Consulting [5].

Motivation 3: Design Space Exploration

The final and most significant motivation for this thesis' focus on early-stage MDO is that it provides value far beyond merely the point design that results from an optimization study. The true value of conceptual MDO is in determining which questions a design team should be asking. Examples of such questions that might be fueled by a design tool leveraging conceptual MDO are given in Figure 2-10.

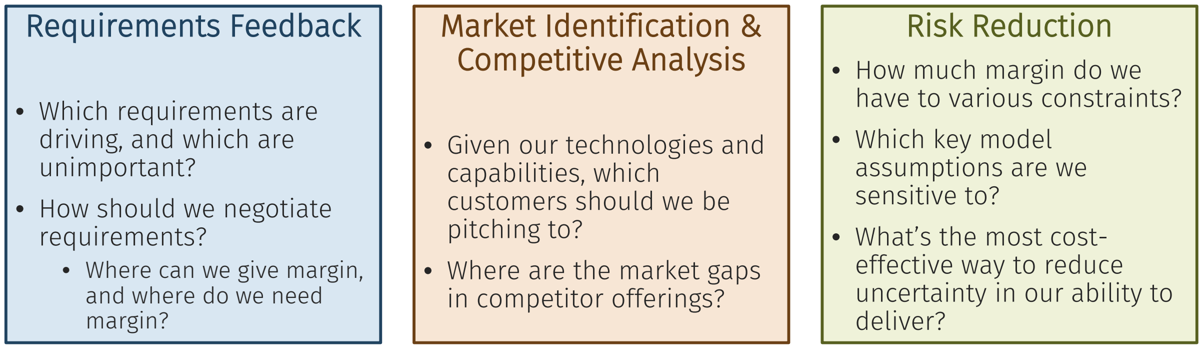

Requirements Feedback
• Which requirements are driving, and which are unimportant?
• How should we negotiate requirements?
• Where can we give margin, and where do we need margin?
Market Identification & Competitive Analysis
• Given our technologies and capabilities, which customers should we be pitching to?
• Where are the market gaps in competitor offerings?
Risk Reduction
• How much margin do we have to various constraints?
• Which key model assumptions are we sensitive to?
• What's the most cost-effective way to reduce uncertainty in our ability to deliver?

Figure 2-10: In addition to answering specific design questions, conceptual MDO tools can help designers develop a broader intuition about the problem space and determine which questions to ask next. This figure gives examples of such exploratory questions.

Though some of these questions can be answered with traditional methods, like post-optimality studies and parameter sweeps, modern conceptual MDO tools can supercharge this to tens, hundreds, or thousands of design variables, allowing for a much more comprehensive exploration of the design space. The inclusion of similarly large numbers of relevant constraints effectively prunes the design space, reining unrealistic designs that a parameter sweep might otherwise yield. This capability is especially important in the early stages of a program, where the design space is often poorly understood and the design team is still developing an intuition for the problem. In this sense, conceptual MDO is a tool for *design space exploration*, not just design optimization.

Chapter 3

A Code Transformations Paradigm for Design Optimization Frameworks

This chapter introduces a new paradigm for engineering design optimization frameworks called *code transformations*. Section 3.1 defines the paradigm and introduces the related concept of code traceability. Section 3.2 presents AeroSandbox, a reference implementation of this paradigm in an engineering design optimization framework. Section 3.3 benchmarks this framework against existing ones, measured by the technical and non-technical metrics of Figure 2-2. Finally, Section 3.4 introduces observations of ideal syntax and interface design for MDO frameworks.

3.1 Introduction and Definitions

In recent years, several advanced scientific computing techniques have proliferated that offer fundamentally new capabilities by using non-standard interpretations of numerical code [97]. Examples of these techniques include:

- Automatic differentiation [101], a technique that allows efficient and accurate evaluation of a function's gradient at runtime
- Automatic sparsity detection [105], a technique that identifies which of a function's outputs may be affected by each input
- Automatic problem transformations (in the context of numerical optimization), including techniques such as:
  - Problem scaling [111], to improve the conditioning of Hessians and linear sub-problems
  - Redundant constraint elimination
- Common subexpression elimination [96], where repeated calculations are automatically identified and rewritten for faster speed via pre-computation
- Backend-agnostic programming, which can enable hardware accelerators (e.g., GPUs), different math library backends, just-in-time (JIT) compilation, and automatic vectorization & parallelization [103]

In this work, we collectively call this set of computational techniques *code transformations*. Formally defined, a code transformation is any operator[1] that a) intercepts some representation of the user's original code at runtime, b) automatically applies some improvement based on analysis of the code itself, and c) returns this improved function to be executed in-place of the original.

This definition is similar to that of a "compiler optimization" in computer science, though a distinction can be drawn in the implied level of code abstraction where the improvement is applied. Compiler optimizations typically refer to lower-level improvements at the level of source code static analysis or a language-level syntax tree (e.g., dead-code elimination, loop fusion, and static type inference and specialization). By contrast, code transformations broaden this to also include improvements at the higher level of computational graphs dynamically constructed within domain-specific modeling languages, or at the level of a data structure describing a complete numerical method (e.g., an optimization problem formulation).

Another useful point of comparison for this code transformations definition is "scientific machine learning" (SciML), which is a term that has become increasingly popular to describe several of these higher-level transformations [37, 98, 112]. In particular, this term has been applied to end-to-end automatic differentiation of physical simulators, especially in cases where it is then used for machine learning applications such as parameter estimation, surrogate modeling, or scientific hypothesis testing via probabilistic programming. SciML methods can be seen as a subset of code transformation techniques that emphasize differentiability and compatibility with machine learning frameworks, and implementations often make various engineering tradeoffs that favor this goal [113]. Therefore, an informal definition of code transformations is that it is a union of both compiler optimizations and scientific machine learning.

[1]i.e., a higher-order function

3.1.1 Code Traceability

The main benefit of generalizing these techniques under a "code transformations" abstraction is to highlight that all of these advanced techniques essentially impose the same shared requirement on numerical code: a property we refer to as code *traceability*. Here, a piece of numerical code is defined as "traceable" if we can construct and directly inspect some functional representation (often, a computational graph) of this code at runtime.

There are several possible options to construct and access these functional representations that satisfy this definition. The most common method (and the one used throughout this work) is aptly called *tracing* in the machine learning literature[2] [99, 103, 116]. This process involves taking functional numerical code, and, rather than evaluating it with standard numerical inputs, instead passing in a "tracer": a symbolic-like dummy data type that dynamically records operations performed on it[3]. The output of the code then includes a recorded computational graph representing the code execution path (the "trace" or "tape" [117]), which can then be manipulated and transformed to enable the advanced techniques listed above.

For this tracing process to work, all mathematical functions used within this numerical code must be able to accept and return these tracer objects[4]. In practice, this often means that traceable code must be written with a specialized numerical framework that accepts that tracer type—we must be able to intercept math library calls in order to properly handle tracers. This also implies that traceability is a property that is defined with respect to a specific numerical framework, rather than a universal property

[2]Tracing is not the only means by which code transformations can be performed—one example alternative is direct source code transformation, which can be seen in frameworks like Tapenade [114]. However, tracing-like strategies are by far the most common in modern, syntactically-rich languages; comprehensive discussion of the tradeoffs and motivations for this are given by Maclaurin [115].

[3]In some frameworks, the tracer is used only for graph construction (e.g., JAX's `Tracer` [103]). In others, the same object serves an additional purpose during numerical evaluation, where it works as a concrete numerical data type (e.g., PyTorch's `Tensor` [117]).

[4]Another observation is that this tracing process is generally easier to implement in languages with dynamic typing or multiple dispatch, as these languages make it easier to switch between traditional and tracer data types in numerical code.

of the code itself.

This concept of traceability allows us to intuitively predict which operations are likely to "break the trace" and render the target code incompatible with code transformations. For example, explicit type-casting[5] often causes problems, since a tracer variable may be coerced into a concrete numerical type, resulting in a lost computational graph. Likewise, any interfaces between programming languages (e.g., a Python function calling a C function) will usually break the trace, as type information is lost across the language boundary. This insight can be used to guide the design of numerical code to be more traceable, and hence more amenable to code transformations.

Traceability is also the reason why it can be challenging to add many of these advanced techniques (e.g., end-to-end problem-level automatic differentiation) into an MDO framework unless it was designed to support these from the start. For example, often standard MDO disciplinary interfaces only allow for concrete numerical data types; so, while individual analyses may be traceable, tracing the full optimization problem end-to-end is usually not possible unless the framework accommodates this. This inherently limits the kinds of problem accelerations that can be applied. As a more concrete example of this, one may find themselves able to use AD to find the partials of a single analysis, but directly computing the constraint Jacobian of the full MDO problem may not be possible[6]. This is a key motivation for the development of a new paradigm for MDO frameworks that considers traceability from the start, as opposed to simply adding on these advanced scientific computing techniques to existing paradigms.

[5]An example of type-casting would be explicitly converting an `int` to a `float`, rather than allowing an array library's type promotion rules to handle this passively.

[6]Instead, the framework needs to stitch these partials together. This becomes particularly messy if tracers cannot be end-to-end propagated through analyses that project from low- to high- to low-dimensionality spaces, like from a design space (low-dim) to a mesh space (high-dim) to an output space (low-dim). Here, the user needs to manually implement some kind of forward-over-reverse AD gradient stitching, rather than letting the framework handle it end-to-end. This may be acceptable for one or two disciplines, but for problems with dozens of interacting submodels it becomes overly burdensome for the user. It also splits the computational graph at these large intermediate data structures, which can cost speed.

3.1.2 Implications for Engineering Design

The main goal of this chapter is to introduce this idea of code transformations (via traceable code) as a possible computational paradigm on top of which an MDO framework can be built. This work deliberately aims to broaden the focus of framework design towards traceability, rather than towards any one specific transformation technique (e.g., automatic differentiation). By finding this common thread, we enable a much more general set of computational techniques to be applied, and this approach remains more scalable as new transformation techniques are developed.

If these code transformation techniques can be applied to engineering design optimization, they could offer significant runtime speed improvements over the black-box optimization methods that form the vast majority of industry use today [71, 98]. As shown later in this chapter, these achievable speeds are comparable to those of state-of-the-art optimization methods in academia, such as disciplined optimization methods[7] [7, 87, 88, 90] and gradient-based methods using user-provided analytic gradients[8] [9, 118, 119].

A crucial advantage of code transformation over many state-of-the-art methods is that they can be applied *automatically* – most of the benefits of these advanced techniques can be gained without requiring any additional effort (or mathematical expertise) from the user. This ease-of-use is critical for practicality – Grant notes that many existing paradigms are hamstrung by a "expertise barrier" [87], which inhibits industry adoption. To assess this further, Table 3.1 compares code transformations to existing MDO paradigms across three key practical metrics. These metrics are derived from the high-level view of the user frictions in engineering design optimization, as given in Figure 2-2. Considerations and further

[7] such as geometric programming and disciplined convex programming

[8] sometimes referred to as "adjoint methods" in reference to a common method for manually deriving these gradients for more-complex analyses

discussion of these tradeoffs are given in Appendix A.

Table 3.1: A subjective comparison of tradeoffs between existing MDO framework paradigms and the proposed *code transformation* paradigm. The industrial state-of-the-art is largely de facto *black-box optimization*. The academic state-of-the-art has two major branches: *gradient-based methods with analytical gradients*, and *disciplined optimization methods*. More detailed discussion of these assessments, including definitions and reasoning, is given in Appendix A.

MDO Framework Paradigm	**Example Frameworks and Tools**	**Ease of Implementation** (sketchpad-to-code)	**Runtime Speed and Scalability** (code-to-result)	**Modeling Flexibility**
Black-box Optimization	SUAVE [120], OpenMDAO* [9], TASOPT [16], PASS [44], FAST [121], FLOPS [122], FBHALE [62], **almost all industry codes**	**Great**	**Limited**	**Best**
Gradient-based with Analytic Gradients	MACH-Aero [79], OpenMDAO* [9], OpenConcept [85]	**Limited**	**Best**	**Great**
Disciplined Optimization	GPkit [7], other convex methods [87, 90, 123], AMPL [124]	**Good**	**Good**	**Limited**
Code Transformations	AeroSandbox† [1], JAX‡ [103], ModelingToolkit.jl‡ [37]	**Good**	**Great**	**Good**

* Can use either paradigm, depending on user's implementation

† Introduced as part of the present work

‡ These are computational tools to facilitate code transformations, rather than design optimization frameworks

In short, the main claim this chapter aims to demonstrate is that a code transformations paradigm yields a favorable compromise between all three tradeoffs: it achieves computational speeds similar to the latest academic methods, retains a learning curve comparable to methods already accepted by industry today, and does not impose overly-burdensome mathematical restrictions. If this hypothesis is validated, it leads to a compelling value proposition: if we can develop new methods for industry engineers to easily write traceable design code, then we can give them access to a host of advanced scientific computing techniques and help shrink the academia-industry MDO gap described in Chapter 2.

Accordingly, the contribution of this chapter is twofold. First, this chapter will conceptually introduce code transformations as a new paradigm for engineering MDO frameworks. Secondly, the thesis will demonstrate strategies that allow traceability of engineering design code with acceptably low user effort, enabling the use of code transformations in practice.

3.2 AeroSandbox: A Proof-of-Concept Framework Implementation

To research the feasibility, opportunities, and limitations of a code transformations paradigm for engineering design optimization in greater detail, we have developed an experimental MDO framework called *AeroSandbox*. AeroSandbox is implemented as a Python-based framework aimed primarily at supporting conceptual-level engineering design. As a design *framework* (analogous to OpenMDAO [9] or GPkit [7]), AeroSandbox is a library that provides underlying utilities for formulating and solving general design problems – this is contrasted against a design *tool* that specializes in supporting a targeted design application (e.g., TASOPT [16]). The code is made publicly available under the open-source MIT license with the goal of gathering as much practical user feedback as possible; further access details are given in Section 3.5.

3.2.1 Graph Construction and Optimization

Fundamentally, AeroSandbox is a tool for building and transforming large computational graphs, and indeed these graphs can be directly inspected both computationally and visually, as Figure 3-1 demonstrates. AeroSandbox uses its own specialized numerics module to intercept and trace user code, and it can interface with tracer-like objects from either the CasADi [96] or JAX[9] [103] libraries to produce a computational graph compatible with these respective libraries. Coverage is provided for a wide range of mathematical functions, including most of the popular NumPy library [125] and key parts of the SciPy library [6].

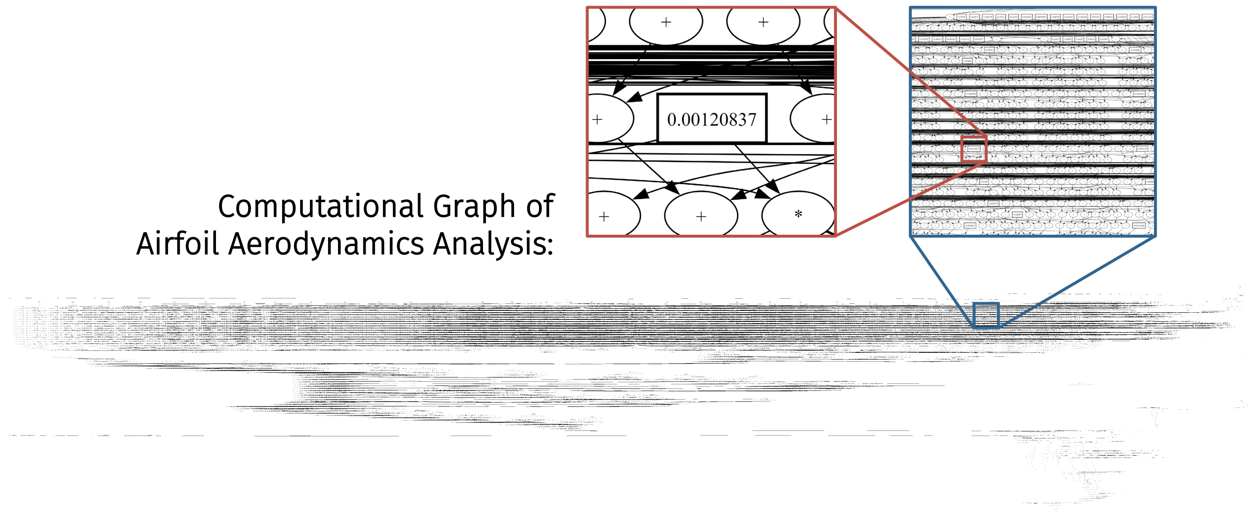


Figure 3-1: An example computational graph, as constructed by AeroSandbox at runtime. Zoomed-in view allows visualization of the scale of such graphs, which can grow to millions of nodes and edges for problems of practical interest. The analysis depicted here is an aerodynamics analysis of an airfoil at a given flow condition, using NeuralFoil (described further in Chapter 7). Elements of the analysis that are not functions of design variables are pre-computed and collapsed into a single node in the computational graph, so the exact runtime graph will depend on the user-specified optimization problem formulation.

During graph construction, code variables that the user specifies as optimization variables are assigned tracers and included as input nodes to the computational graph. Because this happens at runtime on an as-

[9]Experimentally available via the `jaxify` branch of the AeroSandbox GitHub repository, linked in Section 3.5.

needed basis, those parts of the computation that are not functions of optimization variables are evaluated as standard numerical functions. This essentially allows static parts of the optimization problem (e.g., constants, parameters, or frozen variables; and any analyses that are a pure function of these) to be evaluated once and then be reused across multiple function evaluations, which can significantly reduce computational overhead.

While constructing this graph, AeroSandbox deliberately makes certain framework-level decisions which are tradeoffs that tend to work well in engineering design optimization, as contrasted with other possible applications like machine learning [113]. An example of one such choice is the decision to use static computational graphs, where the graph is constructed once during problem specification, rather than repeatedly at each function evaluation during optimization[10]. This static approach results in a reduced computational overhead per-function, which is important as engineering analysis tends to involve deep, scalar-heavy[11] computational graphs. This also makes it more worthwhile to apply global transformations (e.g., common subexpression elimination) to the graph, since the overhead of this is only incurred once. On the other hand, this same decision inherently prohibits value-dependent language-level control flow from being recorded in the graph, which creates challenges in other cases (e.g., adaptive-step numerical integrators). Tradeoffs such as this are made at every step of the framework architecture, from graph construction to code transformation heuristics to optimizer strategies; often there are no unconditionally correct choices, and the end result is a framework that makes numerical choices favoring engineering design cases.

Once the graph is constructed, various user-specified inputs and outputs are connected to form a data structure representing a numerical optimization problem. With this information, code transformations

[10]Behavior analogous to this latter approach is seen in PyTorch, for example, where evaluation and tracing happen simultaneously.

[11]Relative to traditional machine learning, which tends to focus more on large kernel operations (e.g., matrix multiplication and other einsum-like functions) that perform much more computational "work" per call.

can be applied. Most transformations, like automatic differentiation, automated sparsity detection, and problem scaling, are performed automatically and transparently to the user. Here, sensible default heuristics tuned on engineering design optimization problems are used. For example, the framework will automatically choose between forward-mode and reverse-mode automatic differentiation for the constraint Jacobian based on its dimensionality after coloring and compression. Likewise, problem scaling is applied on variables and constraints with a heuristic based on the user-provided initial guess, any bounds constraints, and the constraint Jacobian at the initial guess.

After appropriate code transformations are applied, the transformed problem is then solved using a numerical optimization backend. By default, this backend sends the solve to IPOPT [126], a second-order gradient-based optimization algorithm that performs favorably in large-scale engineering design optimization [82]. Another benefit of IPOPT is that constraints are handled by a primal-dual interior point algorithm, which allows constraint sensitivity information to be obtained naturally as part of the solve. This core mathematical framework architecture, tradeoffs, and heuristics are discussed in further detail in prior work by Sharpe [1].

It should be emphasized that this core design optimization framework is intended to be broadly applicable to many kinds of large-scale engineering systems, and is not aerospace-specific. However, on top of this application-agnostic numerical framework, many optional aircraft-design-specific tools and physics modules are included, which make the framework especially well-suited to support these applications. These specialized tools are the focus of Chapter 5, though an overview of how these tools fit into the broader framework is given in Figure 3-2.

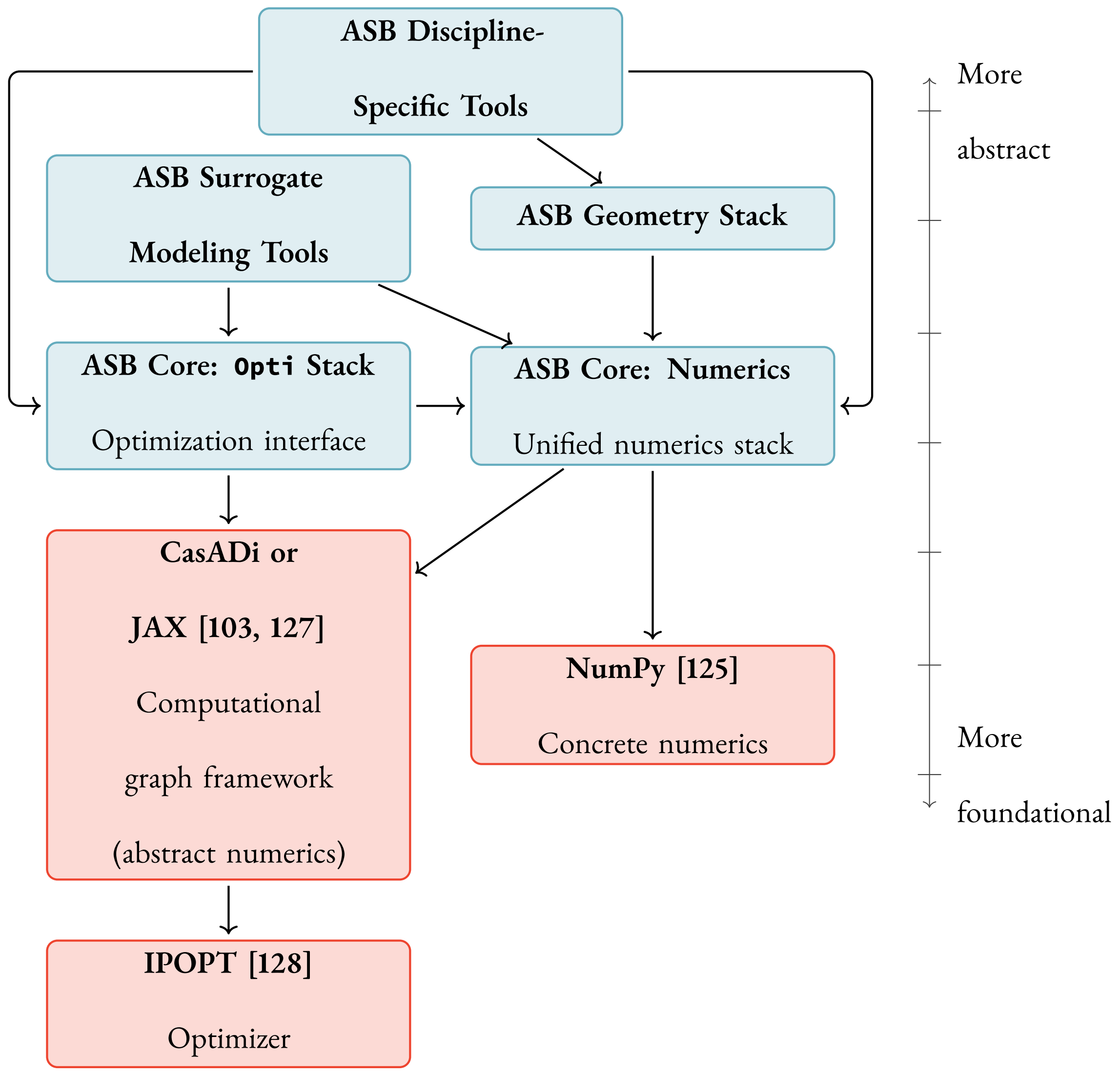


Figure 3-2: Dependency relationships between **AeroSandbox (ASB) components** and **external libraries**. Arrows point toward dependencies. Adapted from prior work by Sharpe [1].

3.3 Performance Comparisons

To test the hypothesis shown in Table 3.1, where a code transformations paradigm can offer a favorable compromise between ease-of-use and computational performance, we have conducted a series of benchmarking studies. Each benchmark is designed to compare performance on one of the three key practical metrics, and against at least one of the existing MDO paradigms listed in Table 3.1.

3.3.1 Code Transformations vs. Black-Box Optimization Methods

The first possible point of comparison is between code transformations and black-box optimization methods, which is the de facto standard in industry today. As shown in Table 3.1, the main advantage of a code transformations framework over such methods is a significant increase in runtime speed and scalability. To demonstrate this, we perform an optimization benchmark on the Rosenbrock problem [129]. This problem is a classic optimization benchmark, designed to be a stress-test as the optimum lies at the bottom of a shallow-curving valley. Here, we solve an N-dimensional extension of this problem, which conveniently gives a knob to smoothly dial up or down the difficulty of the problem [130]. This optimization problem is defined in Equation 3.1:

$$\underset{\vec{x}}{\text{minimize}} \quad \sum_{i=1}^{N-1} \left[100\left(x_{i+1} - x_i^2\right)^2 + \left(1 - x_i\right)^2\right] \tag{3.1}$$

Figure 3-3 illustrates this optimization landscape for the case where $N = 2$, showing the curved valley. For all N, the global optimum is at $\vec{x} = \vec{1}$, where the objective function evaluates to 0. This problem is chosen here because it shares many challenging aspects with engineering design optimization problems: it is nonlinear, nonconvex, and poorly-scaled. Likewise, the Hessian of the function changes substantially near the optimum (as evidenced by the curvature in the valley) – this is designed to put second-order gradient-based optimization methods (such as IPOPT) at a disadvantage. In this benchmark problem, we deliberately choose extremely poor initial guesses, with the goal of stress-testing solver robustness. Specifically, each element of the vector of initial guesses drawn from a random uniform distribution in the interval $[-10, 10]$.

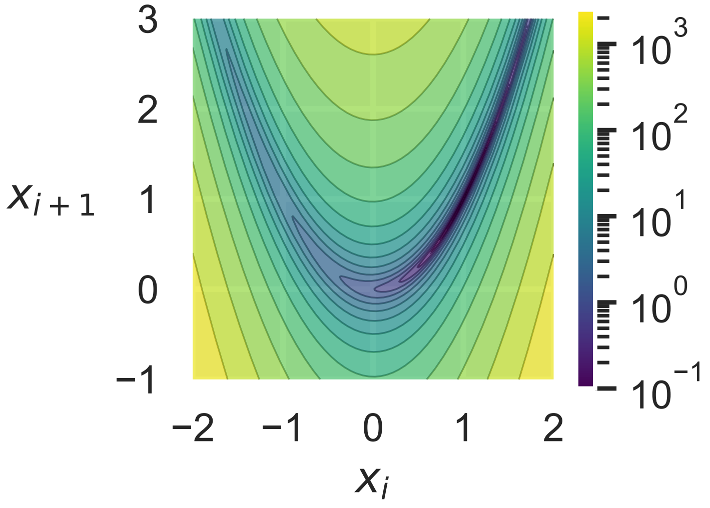


Figure 3-3: The Rosenbrock function, a classic optimization benchmark problem with mathematical formulation given in Equation 3.1. Here, the $N = 2$ case is shown for ease of visualization, though axes are labeled as x_i and x_{i+1} to illustrate that this curving valley occurs in every adjacent pair of dimensions for higher-dimensional variants.

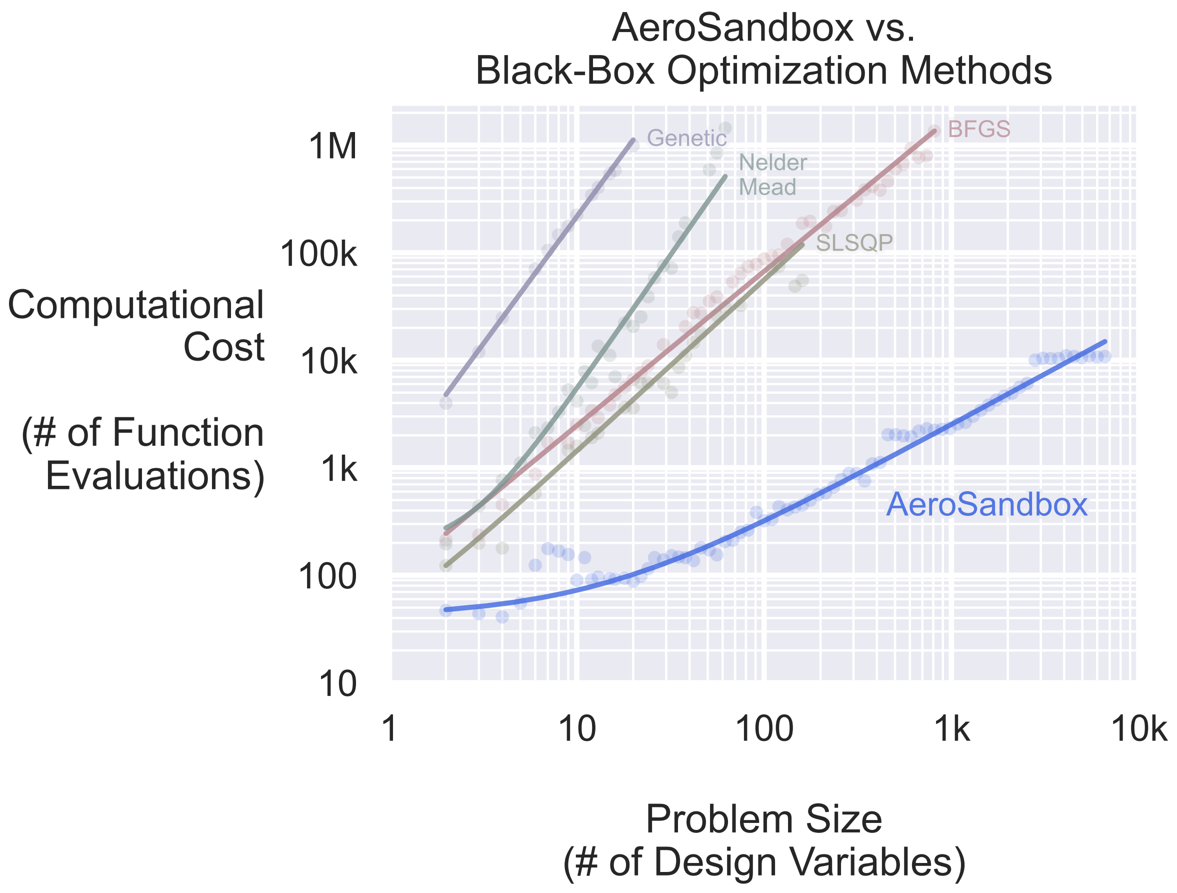


Figure 3-4: Comparison of optimization performance between AeroSandbox and existing black-box optimization methods on the N-dimensional Rosenbrock problem. Other methods are two gradient-free methods (Nelder-Mead simplex method and a differential evolution genetic algorithm) and two gradient-based methods (SLSQP and BFGS), all using common SciPy implementations [6].

In Figure 3-4, the computational cost of AeroSandbox (which leverages code transformations) is benchmarked against existing black-box optimization methods, as the problem dimensionality N is varied. Several notable features can be seen in this figure. First, we find that gradient-based methods (e.g., BFGS, SLSQP) scale much better to high-dimensional optimization problems than gradient-free methods (e.g., Nelder-Mead, genetic), consistent with theory and other literature [71, 131]. Because of this trend, these gradient-based methods are the most representative example of black-box methods used in industry today. As example points of comparison, the SUAVE aircraft design suite [120] conventionally uses black-box SLSQP optimization, and the TASOPT aircraft design code [16] uses a black-box Nelder-Mead simplex method.

Second, we find that AeroSandbox offers significantly faster asymptotic runtime compared to all examined black-box methods. As problem dimensionality increases, the number of function evaluations required for AeroSandbox optimization to converge scales roughly linearly with the number of variables, while the equivalent metric for black-box gradient-based methods scales superlinearly. This improvement in scaling is mostly attributable to the use of automatic differentiation, which allows for more efficient gradient computation.

Third, we find that AeroSandbox offers improved convergence robustness to poor initial guesses compared to some methods, like SLSQP. In Figure 3-3, a method's convergence on the problem at a given dimensionality N is shown by the presence or absence of a colored dot. While some methods converge fairly reliably (e.g., AeroSandbox and BFGS), others like SLSQP show more spotty convergence, with failure becoming increasingly common as the dimensionality increases. This is attributable to two factors: a) the backtracking line search implemented by IPOPT within AeroSandbox and the BFGS solver within SciPy, which stabilizes the optimization process, and b) the presence of automatic problem scaling, which improves the conditioning of the optimization problem.

A final possible contributor to differing performance may be gradient precision issues that can occur with black-box optimization, if such gradients are obtained by finite-differencing. In cases where this is severe, inaccuracies in gradient computation may result in slower optimization convergence in the black-box case. Generally, this is not an insurmountable issue, however, and Martins [71] discusses possible options to mitigate this. For example, complex-step differentiation may be an option that enables higher precision if the black-box function handles (or can be modified to handle) complex data types with proper analytic continuation [132, 133].

In summary, this benchmark demonstrates that a code transformations paradigm can offer faster practical and asymptotic optimization performance than existing black-box optimization methods, especially as problem scale increases.

3.3.2 Code Transformations vs. Disciplined Optimization Methods

Another MDO paradigm that can be compared to code transformation is a broad class of techniques known as *disciplined* optimization methods. The basic premise of such methods is that: if the problem formulation can be restricted to specific mathematical forms, one can use solvers built to exploit the underlying mathematical structure, enabling vast speed improvements over black-box methods. One example form is convex programming, in which the user expresses their model relations with convexity as a constraint, allowing a convex solver to gain speed advantages [90].

When implementing this paradigm into a design framework, one challenge is in how to a) validate that the user's problem formulation is compatible with the required mathematical form, and b) guide the user to write mathematically-compatible code. As an example of how to approach these issues, Grant [87] introduces the notion of "disciplined convex programming", where the framework rigorously tracks various mathematical properties (e.g., convexity) for each user-specified expression, ensuring that the optimization problem remains structured in a way that allows for a specialized solver.

From the perspective of aircraft design optimization, this paradigm became particularly intriguing with the advent of disciplined geometric programming [88, 89]. In this method, we restrict not to the mathematical space of convex functions, but rather that of log-convex functions (i.e., monomials, posynomials, and signomials, which are described further by Burnell [7]). This interest from an engineering perspective is primarily due to research by Hoburg [46], who first applied such methods to conceptual aircraft design and demonstrated that many common first-order sizing relationships in this field are well-approximated by geometric programs. Further research by Hoburg, Kirschen, Ozturk, Burnell, and others [63, 83, 134] have extended this work with the development of GPkit, a geometric programming framework for engineering design optimization.

Using this framework, Kirschen shows that geometric programming (GP) methods can achieve significant runtime speedups over black-box optimization methods on engineering design problems [83]. In addition, this speedup is achieved largely without sacrificing the syntactical clarity of the user's code, which makes expressing design problems relatively efficient.

However, the mathematical restriction to certain categories of log-convex functions can be significant, and working around this by reformulation or approximation can sometimes prove burdensome in engineering practice. As an illustrative example, a case study by Vernacchia [135] documents an effort by an experienced end-user to build geometric-programming-compatible analysis tools for compressible aerodynamics. Here, a common empirical model for the transonic lift-curve slope ($C_{L\alpha}$) is found to break GP-compatibility due to the presence of a $\tan^2()$ function. To fix this, it is instead replaced with a modified 12th-order Taylor series approximation. This strategy indeed works to restore GP-compatibility, but crafting such approximations can be both time-consuming and error-prone. These complex approximations can also lead to reduced interpretability during scenarios such as an engineering design review.

In some cases, approximation as a geometric program is not possible without unacceptable loss of fidelity, so models of interest must instead be written as a *signomial* program [48, 83], a broader class of mathematical functions. This approach restores compatibility with the framework, but it sacrifices many of the benefits of geometric programming: most critically, problems become combinatorially-complex with respect to the number of signomial constraints. In both the case of geometric- and signomial-programming, these mathematical restrictions also limit the extensibility of design codes to higher-fidelity analyses, which are often not easily representable through such relationships.

Therefore, from this perspective, a valuable objective for any new MDO paradigm should be to achieve runtime speeds comparable to these disciplined optimization methods, while relaxing some of their mathematical restrictions to the extent possible. The code transformations paradigm that is proposed here achieves the latter goal of removing some mathematical restrictions: rather than restrict user code to log-convex expressions, the user is restricted to the much-broader category of C^1-continuous expressions. (Like disciplined methods, a specialized numerics library is still required, however. In the case of code transformations, this is due to the traceability requirement.) This relaxation of mathematical restrictions allows users to formulate many more engineering design problems without the need for mathematical rewriting, and the ability to embed common numerical operators such as linear solves and integrators enables extensibility to higher-fidelity modeling.

However, the question remains whether this relaxation of mathematical restrictions comes at the cost of computational performance. To investigate this, we conduct a benchmark study comparing code transformations (via AeroSandbox) to disciplined methods (via GPkit) on an example engineering problem. The specific problem is directly reproduced from the GPkit user documentation [7, 8], in order to ensure that a high-quality and representative GPkit code implementation is available for comparison. This problem in question is a static Euler-Bernoulli cantilever beam analysis problem, where a distributed load is applied; a diagram of the setup is shown in Figure 3-5. Values of various constants are taken from Burnell [8] and reproduced in Listing 5.

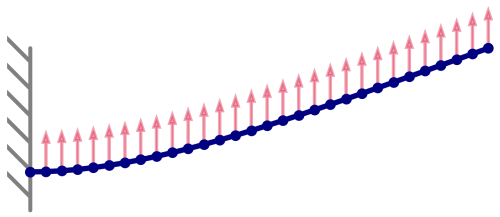


Figure 3-5: Setup for a cantilever beam analysis problem, which is identical to an example in the GPkit user documentation [7, 8] and used as a benchmark to compare solution performance across various frameworks. The beam is discretized into N elements, and a uniform distributed load is applied.

Mathematically, this problem is represented as a fourth-order differential equation, which is discretized into N elements. The goal of the analysis is to compute the state vector (deflection, slope, moment, and shear) at each point along the beam, given some specified distributed load. A useful observation is that, assuming the discretization is performed in a way that expresses integrands as a linear function of these state vectors (e.g., trapezoidal integration), this problem can be solved in a single sparse linear solve. However, this linearity is only advantageous if a framework can exploit this structure, and log-transformation will lose this property.

Numerically, the problem is written in residual (implicit) form, which allows it to be formulated as an optimization problem in various frameworks. As-given, this problem is a pure analysis problem, and the number of constraints (including boundary conditions) exactly equals the number of degrees of freedom. However, one can easily imagine this problem being implemented as a discipline-specific analysis within a larger MDO framework, where the beam analysis is coupled to other analyses (e.g., a wing sizing analysis) and design variables (e.g., wing thickness).

In Figure 3-6, the wall-clock runtime of AeroSandbox and GPkit is compared as the number of elements N is varied. In this chart, four labeled cases are given:

1. **GPkit (cvxopt)**, where the problem is solved using GPkit's default convex optimization solver, CVXOPT [136]. Notably, CVXOPT is free and open-source, so the inclusion of this case allows

an accurate “open-source to open-source” comparison to be drawn. The exact syntax used in this case (and in the MOSEK case below) is given by Burnell [8].

2. **GPkit (mosek)**, where the problem is solved using GPkit’s interface to a commercial solver, MOSEK [137]. This is included to show the performance of GPkit in cases where users have access to a high-performance proprietary solver, and indeed the runtime is roughly an order of magnitude faster than the CVXOPT case.

3. **AeroSandbox**, where the problem is formulated with syntax as close to its basic mathematical form as possible. The exact syntax used here is given in Listing 5; notably, the coding style is declarative and allows the framework to set up its own discretization routines. In this case, a substantial speedup is achieved, in large part due to the fact that the problem is not solved in log-space, and hence the problem’s natural linearity can be exploited by a gradient-based solver. Thus, in this case, the problem can always be solved in one optimization iteration.

4. **AeroSandbox (using GP formulation)**, where the problem is given in its log-transformed form to AeroSandbox, which is identical to what underlying solvers of GPkit would see. This case is perhaps the most interesting of the four, because it shows that even with the same mathematical structure, code transformations can enable accelerations beyond what can be achieved with specialized solvers alone. In this case, subsequent benchmarking reveals that most of this additional speedup is due to sparsity detection and Jacobian compression, since the constraint Jacobian of this problem is quite sparse (due to locality of the governing equations).

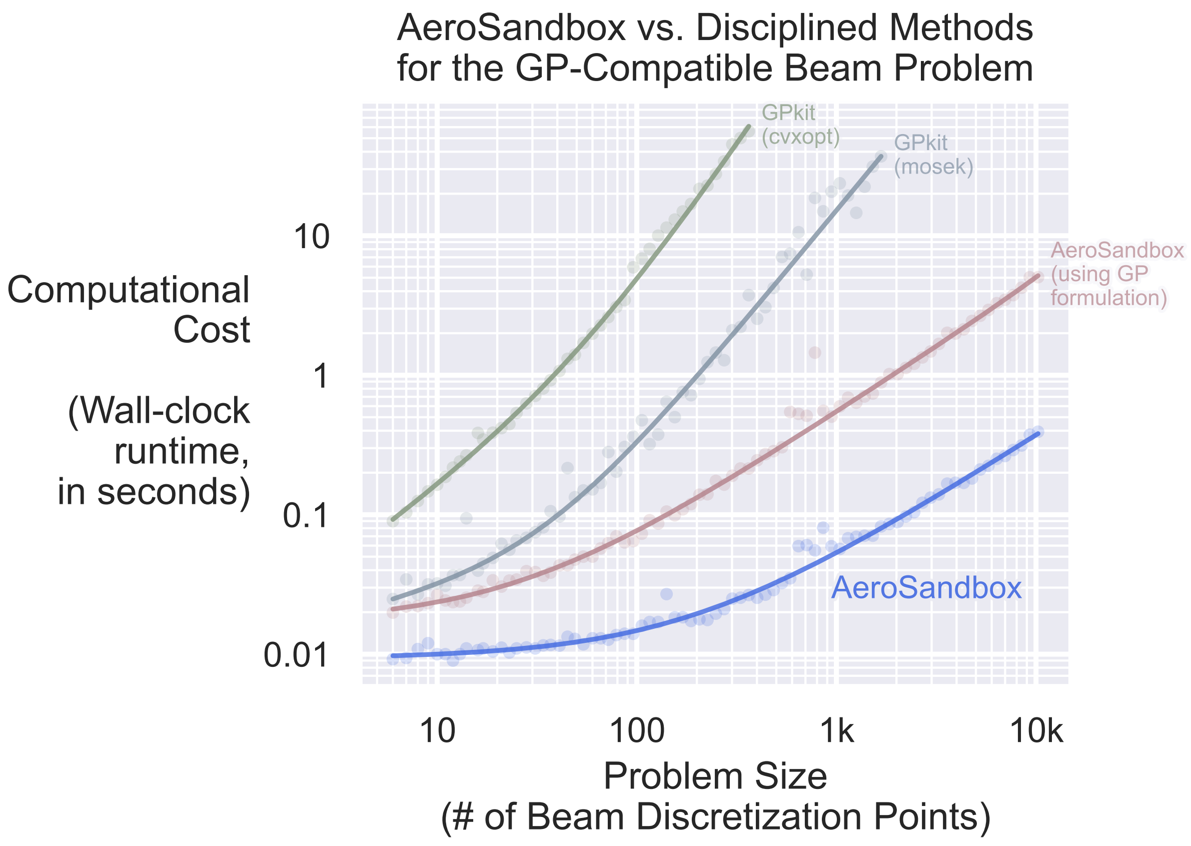


Figure 3-6: Comparison of runtime speed between AeroSandbox and GPkit on the cantilever beam analysis problem. The problem is formulated as a static Euler-Bernoulli beam analysis, with the number of discretized elements, N, is varied to test scalability. The number of optimization variables (i.e., degrees of freedom) is $4N$, as the governing equation is originally a 4th-order ODE and is decomposed to a larger system of first-order ODEs. In all cases, measured runtime includes both problem formulation and solution; time for library imports is excluded. AeroSandbox code implementation is available in Listing 5, and the GPkit code implementation is given by Burnell [8].

Another interesting observation from Figure 3-6 is the fact that the wall-clock runtime of the AeroSandbox case asymptotes to roughly 10 milliseconds in the limit of small N. This is rather large given the simplicity of the problem; if this analysis problem were explicitly formulated as a sparse linear program and given to such a specialized linear solver, it would likely be solved in a fraction of a millisecond. This discrepancy is due to the overhead of the optimization framework, which includes both tracing and applying code transformations. However, this overhead only scales weakly with problem size, and hence the runtime scales linearly with the number of elements N in the limit of large N. This contrasts to the GPkit runtime, which scales superlinearly with N. This trend illustrates a common tradeoff in framework architectures: better asymptotic performance often comes at the cost of increased overhead for small problems, due to the need to find and exploit structure.

The payoff of this computational overhead, however, is that an end-user can take advantage of the sparse linear structure of the problem without needing to explicitly recognize and formulate it as such. This trades engineering effort for computational effort, which may be a favorable tradeoff in the context of quick-turnaround conceptual design problems. This automation also lowers the expertise barrier for users, as they do not need to be aware of the underlying mathematical structure of the problem to achieve good performance.

Other comparisons between a code transformations paradigm and disciplined optimization methods, including two conceptual aircraft sizing case studies, are given in prior work by Sharpe [1]. In short, code transformations can equal or exceed the runtime speed of disciplined optimization methods such as geometric programming, even on problems where this disciplined optimization approach is applicable. This is a promising result, as it suggests that code transformations may be able to offer the best of both worlds on engineering problems: the computational performance of disciplined optimization methods with fewer mathematical restrictions.

3.3.3 Code Transformations vs. Analytic-Gradient Methods

The final comparison to be made is between code transformations and an analytic-gradients paradigm, which focuses on using user-provided partial derivatives to facilitate gradient-based optimization. This is the preferred approach when especially fine-grained control over the MDO architecture and gradient computation process is required, as is often the case in high-fidelity engineering design optimization problems. The ability for users to directly patch in their own gradients is practically required to enable many advanced techniques in PDE-constrained optimization, such as adjoint methods and differentiable volume mesh morphing.

When implemented, user-specified analytic gradients can be faster than its code-transformations analogue of automatic differentiation. This is both for code reasons (e.g., the framework no longer incurs a tracing overhead) but also for mathematical ones (e.g., the ability to use continuous adjoints, for which there is no direct analogue in automatic differentiation [138, 139]).

However, the real benefit of analytic gradients is in the potential for reduced memory cost, especially in the case of the largest-scale design optimization problems. This is because the user can use strategies like checkpointing or time-reversal of ODEs to reduce the memory cost of gradient computation, which is challenging to automate in a general-purpose automatic differentiation framework [139–141]. For this memory reason, a framework based on analytic gradients is in fact the only tractable choice for many high-fidelity engineering design optimization problems.

Another framework benefit of ceding control over gradient calculation to user code is that it makes it cleaner to define disciplinary interfaces, as the "components" for which users specify partial derivatives form a natural disciplinary boundary. This can make it easier for the user to experiment with a wide variety of MDO architectures, which may improve performance in some cases [28]. To contrast with this, the code transformations paradigm is more naturally suited to monolithic architectures (and in particular, simultaneous analysis and design) due to the global nature of the computational graph it constructs [37, 142].

The primary downside of this analytic-gradients paradigm is that it requires substantial mathematical effort from the end-user. Formulating a quick design problem or extending existing analyses often requires hand-derivation and implementation of gradients, and this can stymie experimentation with radical design changes. In addition to the effort required, significant expertise in both mathematics and computer science is also required. Given that end users are often subject-matter-experts in their domain area, rather than in optimization *per se*, this can restrict the potential user base. Even for experienced users, errors in user-specified gradients are a common and difficult-to-debug source of optimization failure.

Because of this, a code transformations paradigm may be a promising alternative to analytic-gradient methods in lower-fidelity design optimization cases, such as conceptual design. In such cases, where the fine-grained control of analytic gradients is less crucial, a simpler user experience that automates some of this optimization machinery may yield reduced overall user frictions. Because of this, Table 3.1 suggests that the main advantage of a code transformations paradigm over analytic-gradient methods is improved ease of implementation.

To demonstrate what this looks like in practice, we implement a benchmark problem for comparison between AeroSandbox and OpenMDAO, which is a popular MDO framework that supports analytic gradients [9]. The specific problem is reproduced from the user documentation of OpenMDAO, once again to ensure that a high-quality implementation is available for comparison [10]. This problem is based on 1-dimensional Euler-Bernoulli beam theory, similar to the example of Section 3.3.2. However, in this case it involves not just analysis but also optimization: the thickness of the beam along its length, $h(x)$, is a design variable. The beam is discretized into N elements along its length, and the thickness distribution need not be uniform. A constraint on the total volume of the beam is imposed, and a cantilever boundary condition is applied. The optimization objective is to minimize the tip deflection of the beam while a point load is applied at the tip. This problem is illustrated in Figure 3-7, with the actual optimal result of the beam design problem drawn to-scale[12]. Additional numerical details and constants for this problem

[12]Although the assumed value for the bending stiffness EI has been scaled for Figure 3-7 in order to make the deflection more visible.

are available in Listing 1.

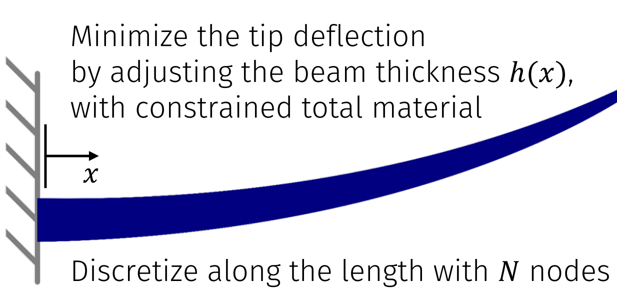

Minimize the tip deflection
by adjusting the beam thickness $h(x)$,
with constrained total material
x
Discretize along the length with N nodes

Figure 3-7: High-level formulation of a beam shape optimization problem, which is identical to an example in the OpenMDAO user documentation [9, 10] and used as a benchmark to compare solution performance across various frameworks. The beam is discretized into N elements along its length, with a point load at the tip.

In Figure 3-8, we compare the wall-clock runtime of AeroSandbox and OpenMDAO as the number of beam discretization elements N is varied. In the OpenMDAO case, the code implementation used for this comparison is given by the OpenMDAO development team [10], and this implementation leverages exact analytic gradients. The corresponding code for the AeroSandbox case is given in Listing 1. In this benchmark study, we see that a code transformations framework can achieve faster runtime than analytic gradient methods across a range of discretization resolutions from $N = 5$ to $N = 500$.

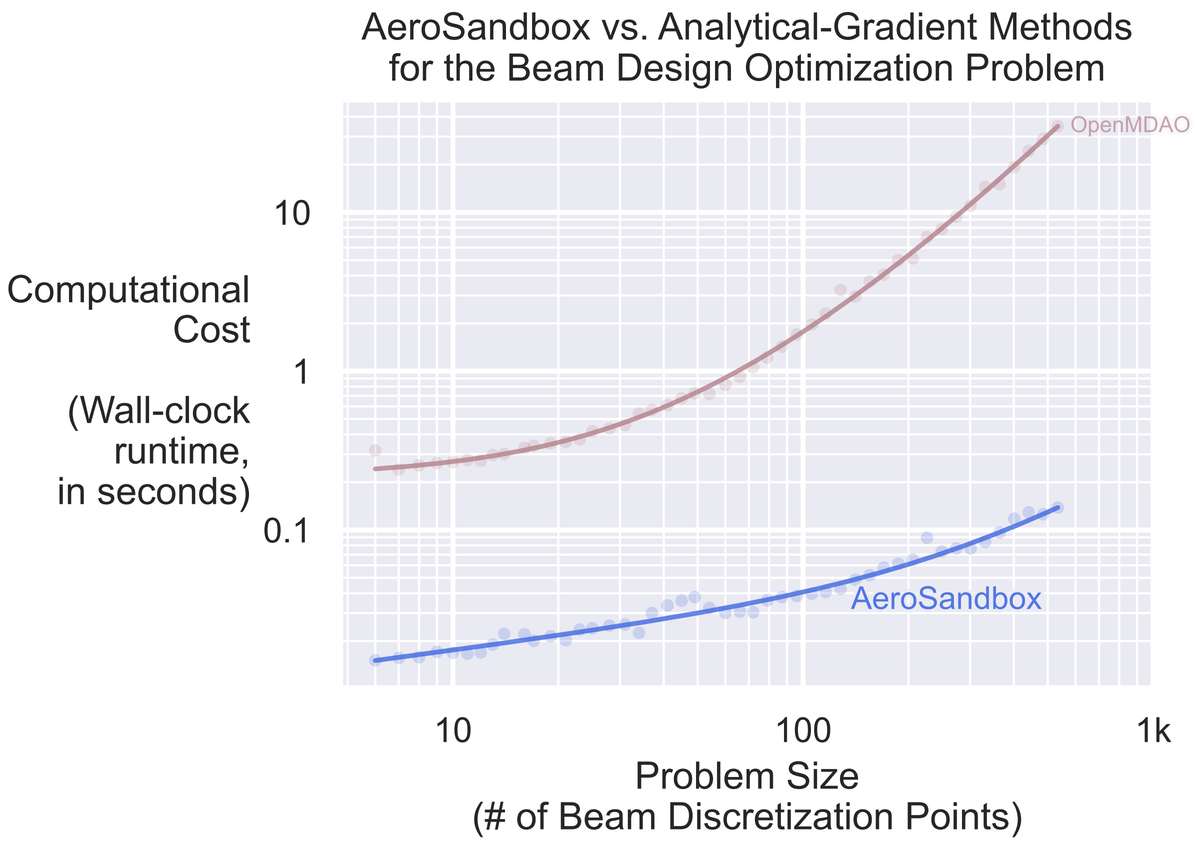


Figure 3-8: Comparison of runtime speed between AeroSandbox and OpenMDAO on the beam shape optimization benchmark problem. The number of discretized beam elements, N, varied to test scalability. In all cases, measured runtime includes both problem formulation and solution; time for library imports is excluded. AeroSandbox code implementation is available in Listing 1, and the OpenMDAO code implementation is given by the OpenMDAO development team [10].

However, the true advantage of a code transformations framework in this case study is not in runtime speed, but rather in a reduction in engineering time. This "code complexity" is challenging to precisely quantify, but one possible (imperfect) metric for this is the number of user-written lines of code required to implement a given problem. Referencing the high-level overview of the optimization user experience in Figure 2-2, this metric loosely corresponds to the amount of friction a user must overcome to implement a problem of interest. Table 3.2 shows that the AeroSandbox implementation of this problem requires roughly 8x fewer lines of code than the OpenMDAO one, which suggests that it requires much less

Table 3.2: Comparison of user-written lines of code and wall-clock runtimes between AeroSandbox and OpenMDAO on the beam shape optimization benchmark problem. AeroSandbox code implementation is available in Listing 1, and the OpenMDAO code implementation is given by the OpenMDAO development team [10].

	AeroSandbox	**OpenMDAO**
Problem-Specific Lines of Code†	35	287
Wall-Clock Runtime ($N = 5$)	0.016 sec	0.253 sec
Wall-Clock Runtime ($N = 500$)	0.235 sec	31 sec

† Includes any imported utilities that are specifically pre-written for this beam problem.

engineering effort to develop.

One key observation is that this reduction in lines of code (or equivalently, the increase in code expressiveness) afforded by code transformations makes the resulting design code far easier to review and interpret. This alleviates one of the key practical frictions identified in Figure 2-2, which involves passing an engineering design review from stakeholders. While performing a detailed code review on a design optimization script will always require some level of computational expertise, it becomes far easier to catch errors when the codebase is concise.

This discrepancy in engineering time occurs primarily because a true analytic-gradients framework requires every computation to be accompanied by a user-specified partial derivative[13]. This is true even for very simple calculations, such as the bending moment of inertia computation ($I = bh^3/12$) in this beam optimization benchmark problem. Furthermore, because these gradients must be linked to their respective functions, usually each of these computations requires the user to define a new data structure[14] to package this information; this also increases the verbosity of the modeling language.

Another reason for the increased verbosity of the analytic-gradients paradigm is that the user must manually-construct component Jacobians if they wish to take advantage of sparsity. (In the case of

[13]As an alternative, OpenMDAO allows users to fall back on black-box finite-differenced gradients, which brings both the advantages and drawbacks discussed in Section 3.3.1.

[14]In OpenMDAO nomenclature, this is referred to as a *component* and is done by extending a base class.

a beam structural shape optimization problem, exploiting sparsity is essentially mandatory to achieve acceptable speed.) In the case of the OpenMDAO implementation of this benchmark problem, roughly half of the problem-specific lines of code are dedicated to assembling the beam's sparse stiffness matrix[15]. Furthermore, because assembly of the global stiffness matrix is performed on the user side of the interface, it is most naturally expressed via a loop in the same language facing the user: Python. This becomes a natural computational bottleneck as problem scale increases, which may induce the user to offload this to a different programming language, adding more complexity to the design tool. In contrast, a code transformations framework with both automatic differentiation and automatic sparsity detection capabilities can handle this task automatically. This sparsity detection can also occur in the language where the computational graph is represented (e.g., in the case of AeroSandbox's CasADi backend, C++), which can result in a substantial speedup.

For high-fidelity design optimization problems and for problems with only a handful of individual disciplines, shifting the responsibility of these numerical implementation details to the user may not be prohibitive – indeed, it may be desirable in cases, as the gradient calculation method can be tailored for each analysis. In particular, given careful user implementation, analytic-gradient methods can allow smaller memory footprints than a code-transformations paradigm that relies on automatic differentiation; ultimately this is required when scaling to high-fidelity problems. However, for conceptual design problems, which typically involve a vast number of low-fidelity models, and where the goal is to quickly explore a large configuration space, the engineering effort required here can be a significant barrier to entry.

It is important to emphasize that, although this study demonstrates some challenges of using a frame-

[15] In this code, definition of the per-element stiffness matrix takes 34 lines, and assembling these into a global stiffness matrix takes 113 lines. In total, the code is 287 lines.

work based on analytic-gradients for conceptual design purposes, doing so is absolutely possible. Several successful efforts such as the OpenMDAO-based OpenConcept [143] and its related OpenAeroStruct [144] provide existence proofs of this; the code architectures of these tools provide useful examples of how conceptual design workflows within this paradigm can be structured. In these cases, the user experience is streamlined by a framework-provided library of pre-written components that encapsulate common disciplinary analyses, each of which also has manually-implemented analytic gradients. This can be a suitable approach, especially in targeted applications where the user's design intent does not deviate too far from the assumptions of these pre-written analyses. However, the end-user experience becomes more challenging when the user needs to extend these analyses or pose design problems outside the intended application scope of the framework, as they are no longer insulated from the frictions outlined here when defining new analyses.

Because of these reasons, we generally assess that a code transformations paradigm offers improved ease of implementation compared to analytic-gradient methods, as depicted in Table 3.1. This is especially true for conceptual design problems, where the user's primary goal is to quickly explore a large configuration space. On the other hand, for high-fidelity design optimization problems where giving the end-user fine-grained control over numerical processes is required, an analytic-gradients paradigm likely remains more suitable.

3.3.4 Summary and Discussion of Benchmark Studies

In all three of the comparison case studies examined in this section, it becomes clear that most framework architecture decisions ultimately come down to tradeoffs: usability, runtime speed, and mathematical restrictions must all be considered. The best choice of MDO paradigm for a given design problem will depend on the specific requirements of that problem, and the user's expertise and preferences.

However, the results of these benchmark studies suggest that a code transformations paradigm can offer a favorable general-purpose compromise between these factors, especially in the context of conceptual design optimization problems.

The benchmark studies of Section 3.3 discuss a variety of considerations to be weighed when developing an engineering design optimization framework, and how different architectural choices link to these tradeoffs. From these studies, we can begin to extract a more concise list of the high-level opportunities and challenges of using a code transformations paradigm for engineering design optimization, relative to other options:

- **Opportunities** (in the context of conceptual-level engineering design optimization):

    1. Code transformations yield performance comparable to bespoke, efficient numerical code, while reducing the amount of problem-specific code that users must write. Although there are many cases where performance could be further accelerated with more specialized numerics, we contend that code transformations generally represent a useful "80/20" Pareto tradeoff for the end user: they achieve most of the performance of bespoke code with a fraction of the engineering effort.

    2. By adding more separation between the problem formulation (i.e., physics modeling) and the numerical optimization process, code transformations let engineers focus on asking the right design question. As noted by Drela [33], developing a problem formulation that faithfully represents design intent is often the most challenging part of practical engineering design optimization. By abstracting away some elements of the numerical optimization process by default, code transformations can help users focus on this critical aspect of the design process.

- **Challenges**:

1. As alluded to in Section 3.1.1, the requirement that user-written numerical code be traceable imposes some restrictions on the end-user; the severity of this falls somewhere between that of the black-box- and disciplined-optimization paradigms. Most fundamentally, the user must formulate their design problem using syntax from a framework-provided numerics library, to allow tracing. This naturally creates a tension against usability, and choices about the syntax of this specialized numerical library can strongly affect the learning curve of an MDO framework.

2. Beyond just the *syntax* of a user's numerical code, a code transformations paradigm also imposes certain preferences on the *style* of a user's numerical code. In particular, for reasons discussed in Section 3.4.2, this paradigm favors a *functional* coding style that avoids side effects and mutable state, since this is more easily traced and transformed. In contrast to this, we later argue that the workflow of engineers working on physics-based design optimization problems naturally tends to lead to *procedural* coding styles, which creates another source of tension in framework development.

It is worth noting that both of the listed challenges of code transformations relate to framework-level tensions between the user experience and computational performance. This motivates a renewed focus on framework choices around syntax and interfaces, since these play a key role in determining the practicality of an MDO framework.

```python
import aerosandbox as asb
import aerosandbox.numpy as np

N = 500  # Number of discretization nodes
E = 1e3  # Elastic modulus [N/m^2]
L = 1  # Beam length [m]
b = 0.1  # Beam width [m]
volume = 0.01  # Total material allowed [m^3]
tip_load = 1  # Tip load [N]

x = np.linspace(0, L, N)  # Node locations along beam length [m]

opti = asb.Opti()  # Initialize an optimization environment
h = opti.variable(init_guess=np.ones(N), lower_bound=1e-6)  # Beam thickness [m]
I = (1 / 12) * b * h ** 3  # Bending moment of inertia [m^4]

V = np.ones(N) * (-tip_load)  # Shear force [N]
M = opti.variable(init_guess=np.zeros(N))  # Moment [N*m]
th = opti.variable(init_guess=np.zeros(N))  # Slope [rad]
w = opti.variable(init_guess=np.zeros(N))  # Displacement [m]

opti.subject_to([  # Governing equations
    np.diff(M) == np.trapz(V) * np.diff(x),
    np.diff(th) == np.trapz(M / (E * I), modify_endpoints=True) * np.diff(x),
    np.diff(w) == np.trapz(th) * np.diff(x),
])
opti.subject_to([  # Boundary conditions
    M[-1] == 0,
    th[0] == 0,
    w[0] == 0,
])
opti.subject_to(np.mean(h * b) <= volume / L)  # Volume constraint
opti.minimize(w[-1])  # Objective: minimize tip deflection
sol = opti.solve()
print(sol(h))  # Gives the optimized beam thickness
```

Listing 1: AeroSandbox implementation of the beam shape optimization problem. Written in Python.

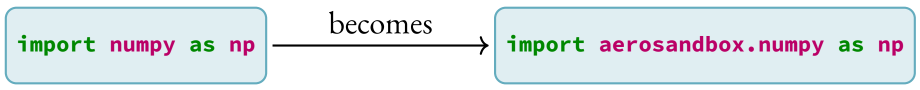
3.4 Syntax, Workflow, and Code Style Considerations

In order for the automatic code tracing technique to be adopted, it must be made as low-friction as possible – ideally invisible to the end user. This section discusses some strategies and requirements to achieve this, using the AeroSandbox framework as a case study.

3.4.1 Implications of Code Transformations on Coding Syntax

The main requirement for code transformations, traceability, requires that user code to be built on top of a custom numerics library such that functions can be intercepted. To minimize the syntax burden associated with this library, one possible strategy is to design the library to closely resemble the syntax of an existing common numerical computing library, such as NumPy [125]. In fact, one can even implement a direct 1:1 drop-in replacement for this numerical library, which enormously reduces the learning curve for the end user. The first major successful application of this strategy was with the advent of Autograd, an automatic differentiation library by Maclaurin et al. [102].

Inspired by this, AeroSandbox takes a similar approach of mirroring a NumPy-like interface for numerical operations, but with the added capability of tracing code execution. Access to this interface is enabled by essentially hijacking the user's NumPy import, which commonly occurs at the beginning of a user-specified analysis; this is shown in Figure 3-9. Because many engineers already write analysis code with NumPy syntax, writing new code and migrating existing analyses becomes much easier – in many cases, the only required change is a single import statement.



```
import numpy as np
```

```
import aerosandbox.numpy as np
```

Figure 3-9: Standard import of the AeroSandbox numerics stack, which acts as a drop-in replacement for the user's NumPy code. Figure reproduced from Sharpe [1].

In recent years, there has been growing momentum in the Python scientific computing community to develop a unified "array API" standard, to greatly expand interoperability and extensibility of numerical libraries [145]. This standard, which is being jointly developed by a consortium of developers representing various popular numerics libraries[16], aims to provide a common interface for numerical operations that can be implemented by any array library. If this standardization effort is successful, it may be possible within the next few years to hook into numerical function calls without replacing the imported library, making the end-user experience even more seamless.

When implementing a specialized array library such as the one illustrated in Figure 3-9, each function must be overwritten to allow for tracing. This can place a substantial burden on the framework developer, both in the initial development phase but also in terms of ongoing maintenance as the syntax of the mirrored library changes. Because of this, selecting the appropriate "attack surface" of numerical functions is crucial: too small, and the user's ability to express their design problem is limited; too large, and the framework developer is overwhelmed with maintenance tasks. To balance these competing tradeoffs, the AeroSandbox numerics library aims to recreate a carefully-chosen subset of commonly-used NumPy and SciPy functions, including:

- Array operations (initialization, indexing, concatenation, stacking, reshaping, etc.)
- Elementary functions (arithmetic, trigonometry, special functions, etc.)
- Conditionals and boolean logic[17]
- Linear algebra
    - Einstein-summation operations (matrix products, dot products, outer products, etc.)

[16]such as NumPy, PyTorch, Tensorflow, JAX, and others

[17]A particularly useful inclusion is the NumPy `where()` function, which allows element-wise conditional assignment. Note that the framework does not explicitly enforce C^1-continuity of the user's code across conditional boundaries, but attempting to optimize while using C^1-discontinous functions will often cause convergence issues. Further discussion is given in prior work by Sharpe [1].

  - Vector and matrix norms
  - Linear solves, Moore-Penrose pseudoinverses
  - Various factorizations (e.g., eigenvalue decomposition)

- Various common utility functions (e.g., `linspace()`)
- Interpolation, including N-dimensional and higher-order variants, and both gridded and scattered methods
- Higher-order discrete numerical differentiation (i.e., gradient reconstruction, for ODEs and PDEs)
- Higher-order numerical integration and quadrature, in both continuous and discrete variants (e.g., acting on either functions or sampled data)
- Coordinate transformations and rotations
- Surrogate modeling and regression tools (e.g., curve fitting tools, kriging, neural network activation functions, orthogonal polynomial families, etc.)

In the author's experience, this subset of functions is sufficient to express a wide variety of engineering design problems, and is a good starting point for a general-purpose engineering design optimization framework. However, the exact choice of functions to include will depend on the specific application domain of the framework.

The author's prior Master's thesis [1] further discusses these syntax-related choices and tradeoffs that should be considered when developing an MDO framework, using examples from the development of AeroSandbox.

3.4.2 Implications of Code Transformations on Coding Style

Another relevant consideration in the development of a numerical framework that allows for code transformations is the coding style that is encouraged by the paradigm. This section will discuss the tradeoffs of various styles in computer science terms – for a more concrete, aerospace engineering example of this in practice, see Section 3.4.3.

In general, it is much easier to write a traceable numerics stack if one can guarantee that the user's code follows a *functional* programming style, which avoids side effects and mutable state (collectively called "in-place operations", since they modify an array without allocating a new memory register) [102, 103, 115]. This is because it the most natural way to store intermediate values of a computational graph during evaluation is by directly attaching them to graph nodes. If the executed code uses in-place mutation, then different nodes in the graph may point to the same array – in other words, the same memory location. (An example of this is a pair of nodes that occur immediately before and after an indexed array assignment). This leads to a overwriting of intermediate values during execution. Unfortunately, many code transformations rely on accessing these intermediate values – for example, reverse-mode automatic differentiation requires these values to evaluate adjoint operators during the backward pass.

To address this, some early numerics libraries that support dynamic tracing, like Autograd [102, 115], simply forbid mutable state in user code – a functional programming style is required. PyTorch takes a similar approach, though a bit less severe: although the developers heavily discourage in-place operations, PyTorch still makes an attempt to handle them by attaching a "version counter" to each array that is incremented upon mutation [117]. This approach does not work in a fair number of cases[18], but it allows

[18]An example of this is when the array is externally-referenced both before and after mutation, and both version remain independently in-scope at gradient computation time. This makes memory duplication unavoidable and requires a computational graph rewrite.

PyTorch to support limited mutation in user code. Critically, in cases where this is not possible, the counter system allows PyTorch to alert the user, rather than silently producing incorrect results.

On the other hand, other tracing-compatible numerics libraries make an attempt to work around this limitation syntactically, which can allow the use of other coding styles (to varying degrees). For example, while JAX [103] does not directly support in-place operations, it does offer the user alternative syntax that can be used to replace in-place array element assignments with functional-style array transformations[19].

The CasADi numerics library implements an particularly sophisticated approach that completely allows for in-place mutation; to the best of the author's knowledge, this is the only Python-based automatic differentiation library[20] that achieves this. This is done by completely decoupling the tracing and evaluation of the computational graph, with two independent "register-based virtual machines" that separately store the traced graph and the executed graph. As described by Andersson et al. [127]:

> "The creation of [an executable expression] in CasADi essentially amounts to topological sorting the expression graph, turning the directed acyclic graph (DAG) into an *algorithm* that can be evaluated. Unlike traditional tools for AD..., there is no relation between the order in which expressions were created (i.e., a *tracing step*) and the order in which they appear in the sorted algorithm."

This solves the problem of intermediate value preservation at its root, since during evaluation, these numerical values are no longer directly attached to graph nodes, but instead stored in a separate memory register (possibly, with different graph topology than the initial computational graph). To the end user, the net result is that they can write code in a procedural style, with in-place operations, and still have it

[19]For example, in JAX, the in-place code `x[0] = 5` can be roughly replaced by the functional code `x_new = x.at[0].set(5)`. This places some burden of understanding mutability on the end-user, but in most cases it at least allows some workaround to be performed here.

[20]among those that support both forward- and reverse-mode automatic differentiation

be traced and differentiated. This is one of the main reasons why AeroSandbox makes the framework-level design decision to default to a CasADi-based numerical backend: it gives the user flexibility to write mutating code.

3.4.3 Coding Styles and the Engineering Design Workflow

We contend that lifting this restriction on only using functional coding styles offers significant usability advantages for engineering design applications. This stems from a belief that the natural workflow of engineers often leads to *procedural* coding styles, where the bulk of the logic is expressed as a sequence of (possibly stateful) instructions, rather than within the stricter confines of pure functions. To put it more colloquially, engineers tend to write their analyses into things that look more like scripts or spreadsheets, and less like libraries. To illustrate why this procedural coding style is popular in engineering practice, consider a representative hypothetical example of how an industry engineer might arrive at the decision to use a design optimization framework:

1. When an engineer initially embarks on a new development effort, the first computational task that an engineer will usually perform is a quick back-of-the-envelope forward analysis to develop some intuition: "Assuming some order-of-magnitude-reasonable design, what information do I need to make a reasonable estimate about its performance?" Here, the goal is not to optimize the design but rather to start understanding the basic problem physics and design drivers.

   For example, suppose an engineer wants to perform basic wing sizing. An initial forward analysis could be phrased as: assume a representative wing, and estimate its lift and drag using basic aerodynamic theory. The assumed wing might look something like Figure 3-10. In code, this might look like Listing 2.

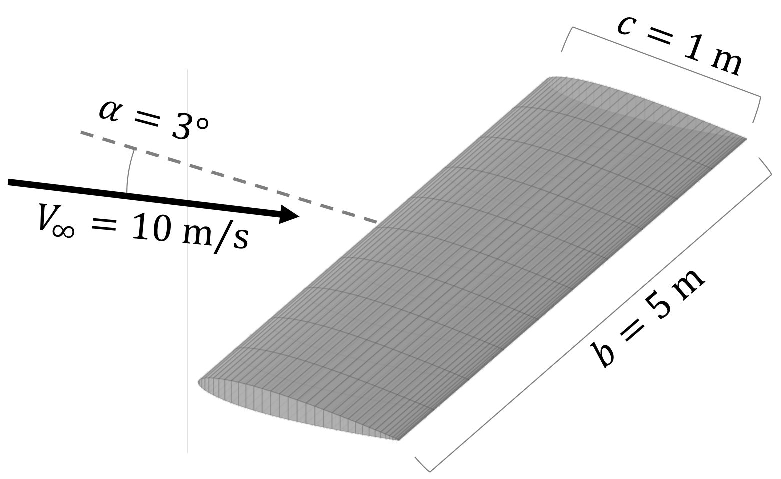


Figure 3-10: An example of an initially-assumed problem geometry for a wing sizing analysis.

```python
import numpy as np

kinematic_viscosity = 1.461e-5  # air, at 20 C [m^2/s]
airspeed = 10  # [m/s]
alpha = 3 # Angle of attack [deg]
chord = 1  # [m]
span = 5  # [m]

CL = (2 * np.pi) * np.radians(alpha)  # Lift coeff., based on thin airfoil theory

Re = airspeed * chord / kinematic_viscosity  # Reynolds number [-]
CD_p = 1.328 / np.sqrt(Re)  # Profile drag coeff., based on Blasius BL

AR = span / chord  # Aspect ratio [-]
CD_i = CL ** 2 / (np.pi * AR)  # Induced drag coeff., based on theory

CD = CD_p + CD_i  # Total drag coefficient

print(CL / CD)  # Result: 38.7
```

Listing 2: Example of a simple forward analysis script to estimate the aerodynamic performance of an untapered, unswept, untwisted, planar wing using napkin-math-level theory.

2. So far, one could argue that the code in Listing 2 could be considered both procedural and functional, because despite the single-level scope of the analysis, it is stateless and non-mutating. However, consider what happens when the engineer wants to build on top of this analysis and start

asking natural next-step questions. For example, "What is the minimum-drag wing that is possible, if the geometry and angle of attack are varied?", and "How does the resulting design need to change if an implicit lift-equals-weight closure is enforced, with some assumed wing weight model?"

In frameworks that require a functional style, we would need to rewrite the script with several major changes. First, the problem would need to be structured as a single top-level function wrapper that takes in all variables and returns all constraints and the objective[21]. Secondly, as a practical matter, all disciplinary analyses would need to be rewritten as standalone pure functions. (This makes little difference in this simple example, but in more complex MDO problems with hundreds of interacting models, this creates an enormous amount of boilerplate code and obfuscates design intent.) An undesirable side effect of all of this added boilerplate code is that it "locks in" the problem formulation – changing which quantities are variables, which are constraints, and which are fixed parameters requires a substantial rewrite of the entire analysis.

Within frameworks that allow optimization formulation in a procedural style, however, converting this initial analysis into a design problem becomes far easier. Expressing this optimization problem becomes a natural extension of the initial analysis, and this can be achieved *without adding new scopes*. This minimizes the amount of boilerplate code required to initially formulate highly-coupled design problems, because the user is not required to partition disciplines into separate namespaces[22].

To show this more concretely, Listing 3 gives an example of how a user might extend the initial analysis from Listing 2 to a simple design optimization problem that answers these questions. This

[21] A code example of this, for the curious reader, is in the publically-available FBHALE MDO code. Here, a single top-level master function takes in all variables and returns all constraints.

[22] Of course, as the design matures and the depth and complexity of a problem increases, separation of disciplines into separate modules with defined interfaces becomes increasingly attractive. However, in the case of initial conceptual feasibility analysis, a framework that mandates this separation often adds unnecessary overhead.

listing uses syntax from AeroSandbox, a design optimization framework that supports procedural coding styles. In Listing 3, the lines of code that differ from the original code of Listing 2 are highlighted, which shows the minimal change in overall code structure. Notably, new variables and constraints are simply "tagged" during the problem specification, rather than requiring a functional-style rewrite of the entire analysis. Even an implicit constraint (the lift-weight closure) can be expressed in a single line of code, without pulling out the evaluation of this constraint into a separate function.

```python
import aerosandbox as asb
import aerosandbox.numpy as np  # Patches in the AeroSandbox numerics stack

opti = asb.Opti()  # Start an optimization environment

kinematic_viscosity = 1.461e-5  # at 20 C [m^2/s]

airspeed = 10  # [m/s]
alpha = opti.variable(init_guess=3) # Angle of attack [deg]

chord = opti.variable(init_guess=1, lower_bound=0)  # [m]
span = opti.variable(init_guess=5, lower_bound=0)  # [m]

CL = (2 * np.pi) * np.radians(alpha)  # Lift coeff., based on thin airfoil theory

Re = airspeed * chord / kinematic_viscosity  # Reynolds number [-]
CD_p = 1.328 / np.sqrt(Re)  # Profile drag coeff., based on Blasius BL

AR = span / chord  # Aspect ratio [-]
CD_i = CL ** 2 / (np.pi * AR)  # Induced drag coeff., based on theory

CD = CD_p + CD_i  # Total drag coefficient

lift = 0.5 * 1.225 * airspeed ** 2 * CL * chord * span  # Lift force [N]
weight = (1 + chord * span ** 2) * 9.81  # A simple hypothetical weight model [N]
opti.subject_to(lift == weight)  # Implicit L=W constraint

opti.minimize(CD)  # Objective function
sol = opti.solve()  # Solves the problem

print(sol(CL / CD))  # Result: 115.2
```

Listing 3: Example of a simple design optimization problem to find the minimum-drag wing, starting from the initial analysis in Listing 2. Highlighted lines show elements added or modified from the initial analysis.

The ability to support these inline definitions of optimization problem elements makes it much easier to rapidly "turn the problem around" and change the problem formulation between the forward problem (analysis) and the inverse problem (optimization). For example, changing the problem parameterization (i.e., which variables is the optimizer free to modify, and which are frozen to assumed constants) typically requires significant rewriting in a functional problem formulation, but in a procedural one this becomes a seamless one-line code change. This becomes especially apparent in large-scale MDO problems, where restricting the user to a functional coding style forces them to be much more careful with sequencing the analysis order to maximize feed-forward information flow[23].

Based on this example, we contend that optimization applications to engineering design benefit strongly from a framework that allows for a procedural coding style – this tends to naturally mimic the workflow of practicing engineers as they incrementally develop their problem from analysis into optimization. An interesting observation is that this contrasts against the needs of other applications of traceable numerics frameworks, like machine learning. For example, Autograd developer Maclaurin [115] states that "Our experience using Autograd has been that a functional style is a natural fit for the sorts of modeling and inference problems we like to solve in machine learning, and it rarely feels burdensome." This makes sense, because machine learning workflows tend to involve relatively simple computational graphs that are data-heavy but operator-light[24], while engineering workflows tend to use computational graphs that are data-light but operator-heavy. (This can be seen in the example engineering computational graph of Figure 3-1.) This highlights that, although engineering design optimization has benefited heavily from machine learning advances in the past decade, there are some fundamental reasons why tools developed for machine learning are often not perfectly suited for direct use in engineering design optimization, and therefore frameworks designed with this purpose in mind are advantageous [113].

[23]because functional coding forces much more partitioning into separate scopes, and required information needs to be "piped around" to the appropriate scope level

[24]Consider that a multi-layer perceptron may consist of millions of parameters, but the forward pass can still be expressed in a dozen or so large kernel operators in code – machine learning code is typically highly vectorized. By contrast, an engineering MDO tool may have hundreds of empirical curve-fit-style models, leading to much more complex computational graphs and information flow.

3.4.4 Miscellaneous Considerations

A final relevant choice that any engineering design framework needs to consider is how to handle units in dimensional quantities, if at all. This is particularly interesting in the context of code transformations, because standard ways of handling units essentially "box" all dimensional quantities within a specialized data structure that contains both the numerical value and the associated unit[25]. Predictably, this can cause problems when tracing, since it forces expressions to use (and expect) specific data types – the units container. Because of this, passing a dynamic tracer object through the user analysis may not be possible, as functions are no longer type-agnostic. Hence, adding a required units system to an MDO framework often fundamentally-precludes strategies that rely on code transformations.

Even if this traceability issue is fixed, boxing all quantities in units objects also incurs significant performance penalties – regardless of whether code transformations are used. It is not uncommon to see order-of-magnitude slowdowns in engineering code that attempts to directly embed units. This is mostly due to two reasons. First, it forces functions to do type-checking and units propagation (essentially, a symbolic algebra step) on every operation, which is a significant overhead in operator-heavy code[26]. Secondly, this units-container data structure forces numerical data to be stored on the memory heap, rather than the stack (where intermediate arrays would normally be stored), which is slower due to the increased memory access latency. This problem becomes even worse if the user wishes to perform any hardware acceleration (e.g., GPU computing) or , since the container must be packed and unpacked at each operation.

Because of these reasons, even though explicitly storing units in engineering analyses and MDO

[25] For example, see the Pint package in Python [146], or Unitful.jl in Julia [147].

[26] An interesting analogy is that this is slow for the same reason that type-unstable numerical code (e.g., most Python code, where the heavy lifting is done in Python, like direct looping) is slower than type-stable code (e.g., code in any static language, like C++ or Fortran).

frameworks may initially seem attractive, it usually results in significant headaches down the line. Of course, it is still useful to have some strategy for preventing potentially-costly unit misunderstandings. Over the years, several strategies have emerged that, in the author's opinion, are superior to the direct unit-boxing approach:

One possible existing strategy is aggressive non-dimensionalization, where all user inputs and outputs are either expressed in nondimensional quantities, or (in cases where this is not possible) immediately nondimensionalized after input. This is a particularly common strategy in aerodynamics codes, as seen in codes like AVL [148], SU2 [149], Flow360, and others. However, while this strategy works well within a single tightly-scoped analysis, is not necessarily scalable for large multidisciplinary problems. In these large MDO codes, attempting to pass around a set of global reference quantities to all analyses quickly becomes unwieldy, so global nondimensionalization is not a practical solution.

An alternative strategy, and the one used by AeroSandbox, is simply to establish a convention that all variables are in base units from some globally-used *coherent*[27] units system, or derived units thereof. The most obvious such coherent system is SI, but others (e.g., foot-pound-second, using slugs for mass) are also feasible. AeroSandbox assumes an SI system, and hence all variables are in base or derived quantities (e.g., m, kg, sec, N, m/s, J, Pa, etc.). Regardless of the system chosen, coherence allows all derived units to be implemented without any scaling factors or unit wrappers, which improves both readability and computational performance.

In this strategy, unit interpretation can be facilitated by clear, interpretable variable names that make the expected dimensions of a given quantity unambiguous to the user; in the age of autocomplete in modern code editors, this requires minimal effort. In cases where long-standing industry convention is

[27]I.e., a system of units that involves no internal conversion factors; all derived units are products of powers of base units, with no scaling factors.

to use non-coherent units, this can easily be handled by an explicit indicative suffix. For example:

- `battery_capacity` $\rightarrow$ Joules
- `battery_capacity_kilowatt_hours` $\rightarrow$ kilowatt-hours
- `aircraft_endurance` $\rightarrow$ seconds
- `aircraft_endurance_hours` $\rightarrow$ hours

A consequence of this strategy is that quantities may differ in numerical value by many orders of magnitude; an elastic modulus may be on the order of 10^{11} Pa, while a wing skin thickness may be on the order of 10^{-3} m. Many years ago, when single-precision floating-point arithmetic was the norm, this could cause numerical issues due to the limited dynamic range of the data type. However, with the advent of double-precision floating-point arithmetic as the standard, this is nearly never an issue in practice. One could also be reasonably concerned that this scaling difference could cause optimization difficulties[28]; however, with the automatic scaling performed with AeroSandbox, all quantities are brought to a similar order of magnitude before the problem is passed to the optimizer, and this is not an issue.

Because of these considerations, the author recommends that code-transformation-based MDO frameworks avoid explicitly embedding units in the code via unit-boxing strategies, and instead rely on a coherent units system and clear variable naming conventions to prevent unit misunderstandings. This strategy is both more performant and more flexible, and industrial users of AeroSandbox have generally reported that this approach is intuitive and easy to adopt.

[28]because, although most second-order gradient-based optimziation algorithms are mathematically scale-invariant, the poor condition number of the Hessian (or Hessian approximation) could make for inaccurate steps

3.5 Computational Reproducibility

Source code, installation instructions, documentation, and walkthrough tutorials for the AeroSandbox framework are available at `https://github.com/peterdsharpe/AeroSandbox`. All materials are released under the MIT License, which allows broad permissions on how the software can be used.

The framework is also available on the Python Package Index (PyPI) as the package `aerosandbox`; to install, use the command `pip install aerosandbox[full]`, where the `[full]` suffix causes optional dependencies to be installed as well. For users less familiar with the Python packaging ecosystem, a step-by-step walkthrough of the installation process can be found in the appendix of prior work by Sharpe [1].

Chapter 4

Aircraft Design Case Studies and Framework Capabilities

As indicated in Table 3.1, the main benefit of the code transformations paradigm proposed in Chapter 3 is to achieve runtime performance comparable to specialized methods [1], but without sacrificing ease-of-use

[1] such as analytic-gradients methods or disciplined optimization methods

and mathematical flexibility. Chapter 3 also introduces AeroSandbox, a proof-of-concept framework built on this paradigm.

In this chapter, we take the next step of demonstrating that AeroSandbox enables rapid and useful design capabilities on problems of practical interest. To achieve this, a series of aircraft design case studies using AeroSandbox are presented, as these provide fruitful examples for discussion. Many of these case studies are inextricably linked to the research process through which the AeroSandbox framework itself was developed. AeroSandbox was developed using an iterative "spiral development process", as depicted in Figure 4-1. In short, at every step of the framework development process, a symbiotic bidirectional relationship between the framework and the case studies was maintained, with the goal of simultaneously asking and answering two questions:

1. **Framework-to-problem**: Using the capabilities of this design framework, how can we improve this specific engineering system and gain practical insight into the design space at hand?
2. **Problem-to-framework**: Based on the experience of performing this applied case study, what broadly-applicable user needs and workflows can we identify, and how can we improve our design framework accordingly?

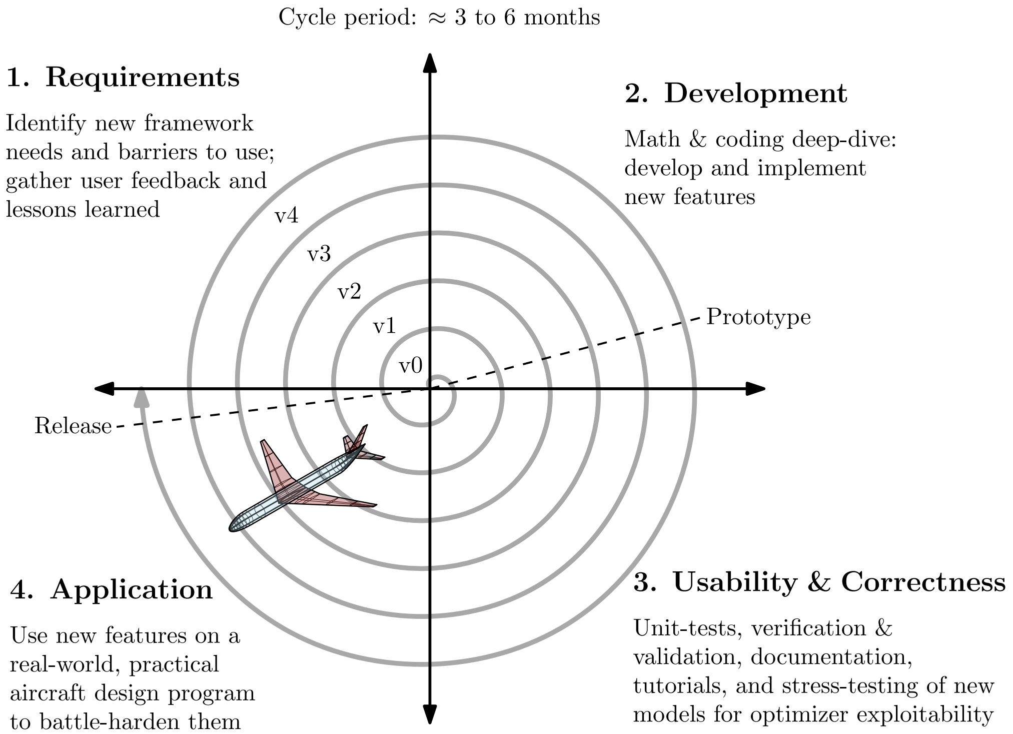

Cycle period: ≈ 3 to 6 months
1. Requirements
Identify new framework needs and barriers to use; gather user feedback and lessons learned
2. Development
Math & coding deep-dive: develop and implement new features
3. Usability & Correctness
Unit-tests, verification & validation, documentation, tutorials, and stress-testing of new models for optimizer exploitability
4. Application
Use new features on a real-world, practical aircraft design program to battle-harden them
v4
v3
v2
v1
v0
Prototype
Release

Figure 4-1: The iterative spiral development process used to develop AeroSandbox, where applied use and framework development are intertwined. The aircraft in the figure is a visualization output using the AeroSandbox geometry stack, and depicts the hydrogen-fueled aircraft from the case study of Section 4.4.

The process of iterating through this loop over several case studies resulted in framework changes that improved capabilities and ease-of-use. This chapter presents a few vignettes of these various case studies that were used to develop AeroSandbox. It is important to emphasize that, although the aircraft design case studies presented in this chapter are fascinating real-world problems, the main intended intellectual contribution of this portion of the thesis is not the design of these specific aircraft (or even the example framework itself[2]). Instead, the deeper goal is to introduce a principled paradigm for building engineering design optimization frameworks. Because of this, complete detailed enumeration of all modeling assumptions for specific problems is omitted in this chapter for brevity; however, both

[2]MDO frameworks naturally come and go over the years as new opportunities in scientific computing and programming languages emerge. It is not the author's intent or expectation that AeroSandbox be the "end-all" framework for aircraft design. Rather, it is a proof-of-concept implementation that aims to demonstrate the potential utility of one possible framework approach for readers interested in building a future framework.

references to literature discussing these details and links to the raw source code itself are provided for most case studies, for the interested reader.

4.1 Simple Aircraft Design Problem

This section includes content adapted from the author's prior publication [1].

A first simple aircraft design problem, called *SimpleAC*, is included here to show what design code might look like in code transformations framework at the early stages of conceptual aircraft sizing. At this "napkin-math" stage, the main analyses (e.g., aerodynamics, structures, propulsion) can be cleanly expressed in simple analytical expressions, and the optimizer fulfills the role of closing the sizing loop. Because of this, this problem is simple enough to include both the complete problem statement and the complete solution code directly here, which is useful for readers interested in observing the mapping between these. This design problem itself was proposed by Hoburg [150] and is reproduced (with slight modifications) by both Ozturk [151] and Kirschen [83]. It is restated here in full form:

Simple Aircraft (SimpleAC)

$$\begin{aligned}
\underset{AR, S, V, W, C_L, W_f, V_{\text{f, fuse}}}{\text{minimize}} \quad & W_f \\
\text{subject to} \quad & W \geq W_0 + W_w + W_f, \\
& W_0 + W_w + \frac{1}{2} W_f \leq L_{\text{cruise}}, \\
& W \leq L_{\text{takeoff}}, \\
& W_f \geq \text{TSFC} \cdot t_{\text{flight}} \cdot D, \\
& V_{\text{f, wing}} + V_{\text{f, fuse}} \geq V_f
\end{aligned} \tag{4.1}$$

where: D = Cruise drag

$Æ$ = Wing aspect ratio (here, the wing is assumed to be rectangular)

S = Wing area

V = Cruise airspeed

W = Total weight

C_L = Cruise lift coefficient

W_f = Fuel weight

$V_{\text{f, fuse}}$ = Volume of fuel in fuselage

We are also given the following physics models:

- The chord $c = \sqrt{S/Æ}$, from geometric relations.
- The drag $D = \frac{1}{2}\rho V^2 C_D S$
- The drag coefficient $C_D = \frac{\text{CDA}_0}{S} + kC_f\frac{S_{\text{wet}}}{S} + \frac{C_L^2}{\pi Æ e}$
    - $C_f = 0.074 \cdot \text{Re}^{-0.2}$, the Schlichting turbulent flat plate boundary layer model
    - $\text{Re} = \frac{\rho V c}{\mu}$
- The cruise lift $L_{\text{cruise}} = \frac{1}{2}\rho V^2 C_L S$
- The takeoff lift $L_{\text{takeoff}} = \frac{1}{2}\rho V_{\text{min}}^2 C_{L,\text{max}} S$
- The wing weight $W_{\text{wing}} = W_{\text{w, structural}} + W_{\text{w, surface}}$
    - $W_{\text{w, structural}} = W_{\text{w, c1}} \cdot \frac{N Æ^{1.5}\sqrt{W_0 W S}}{\tau}$
    - $W_{\text{w, surface}} = W_{\text{w, c2}} \cdot S$
- The aircraft's endurance $t_{\text{flight}} = \text{Range}/V$
- The fuselage drag area scales with fuel volume as $\text{CDA}_0 = V_{\text{f, fuse}}/(10\text{ m})$

- The total fuel volume $V_f = \frac{W_f}{g \rho_f}$
- The fuel volume in the wing $V_{\text{f, wing}} = 0.03 S^{1.5} A\!\!\!R^{-0.5} \tau$

where:	g	= 9.81 m/s^2, Earth gravity.
	ρ_f	= 817 kg/m^3, the density of fuel.
	Range	= 1000 km, the aircraft mission range.
	TSFC	= $0.6\ \text{h}^{-1} = 1.67 \times 10^{-4}\ \text{s}^{-1}$, the thrust-specific fuel consumption.
	k	= 1.17, the form factor.
	e	= 0.92, the Oswald efficiency factor[a].
	μ	= $1.775 \times 10^{-5}\ \text{kg}\,\text{m}^{-1}\,\text{s}^{-1}$, the sea-level dynamic viscosity of air.
	ρ	= 1.23 kg/m^3, the sea-level density of air.
	τ	= 0.12, the airfoil thickness-to-chord ratio.
	N	= 3.3, the ultimate load factor.
	V_{min}	= 25 m/s, the takeoff airspeed.
	$C_{L,\text{max}}$	= 1.6, the takeoff lift coefficient.
	S_{wet}/S	= 2.075, the wetted area ratio.
	W_0	= 6250 N, the aircraft weight excluding the wing and fuel.
	$W_{\text{w, c1}}$	= $2 \times 10^{-5}\ \text{m}^{-1}$, a wing weight coefficient.
	$W_{\text{w, c2}}$	= 60 Pa, another wing weight coefficient.

[a]This Oswald efficiency assumption is markedly higher than what would be realistically achievable, but it is reproduced from previous references to facilitate comparison [83, 150, 151].

This design problem can be translated from the formulation above into the solution code, which is given in Appendix C.2. Notably, the code of this problem is essentially a 1:1 translation of the problem statement into Python code, with almost zero additional boilerplate code required to set up this problem beyond the raw physics models and constants definitions. In fact, the code version of this problem is actually shorter on-the-page than the natural-language problem statement itself. This conciseness is a good approximate measure of the framework's usability, since it allows the user to focus on the problem physics and mathematical formulation, rather than the numerical mechanics of the optimization process.

The code given in Appendix C.2 converges to a solution in 14 iterations, with a median runtime[3] of 18 milliseconds. This is effectively-instantaneous for a human user, and this speed combined with the conciseness of the code gives the user great flexibility to explore the design space. The results of this AeroSandbox solve are shown in Table 4.1, and are consistent with the results of Ozturk [134].

Table 4.1: Solution of SimpleAC (Eq. 4.1), found with AeroSandbox.

Figure of Merit	**Optimal Value**
Fuel weight W_f	937.8 N
Aspect ratio $A\!\!\!R$	12.10
Wing area S	$14.15\ \mathrm{m}^2$
Cruise airspeed V	$57.11\ \mathrm{m/s}$
All-up weight W	$8,705$ N
Cruise lift coefficient C_L	0.2901
Fuel volume in fuselage $V_{\text{f, fuse}}$	$0.0619\ \mathrm{m}^3$

[3]Measured on a laptop a Ryzen 7 5800H CPU. Measures end-to-end wall-clock runtime, including problem formulation (i.e., tracing, in Python), optimization (via IPOPT, in C++), and function evaluation (i.e., on the CasADi VM, in C++).

4.2 MIT Firefly Micro-UAV

4.2.1 Vehicle Overview

One of the first aircraft design case studies used to develop AeroSandbox was the *MIT Firefly*, a small tube-stowed, rocket-propelled, transonic micro-UAV. The concept of operations of this air vehicle is summarized in Figure 4-2. Firefly is designed to be deployed in-flight from a host aircraft, launching from a small standardized canister. This requires that the aircraft fit within a $70 \times 70 \times 480$ mm box in its stowed configuration, but the requirements admit the option of unfolding the vehicle into a flight configuration immediately after launch. After launch, the air vehicle is required to maintain an airspeed of at least Mach 0.8 for at least 60 seconds, without loss of altitude; this is termed the "dash phase". After the dash phase, the vehicle continues to fly in a "glide phase" until it runs out of both propulsive energy and altitude, at which point it performs a controlled crash landing. During this glide phase, there are no airspeed or other trajectory restrictions. The objective of the vehicle design is to maximize the *total mission range*, including both the dash and glide phases.

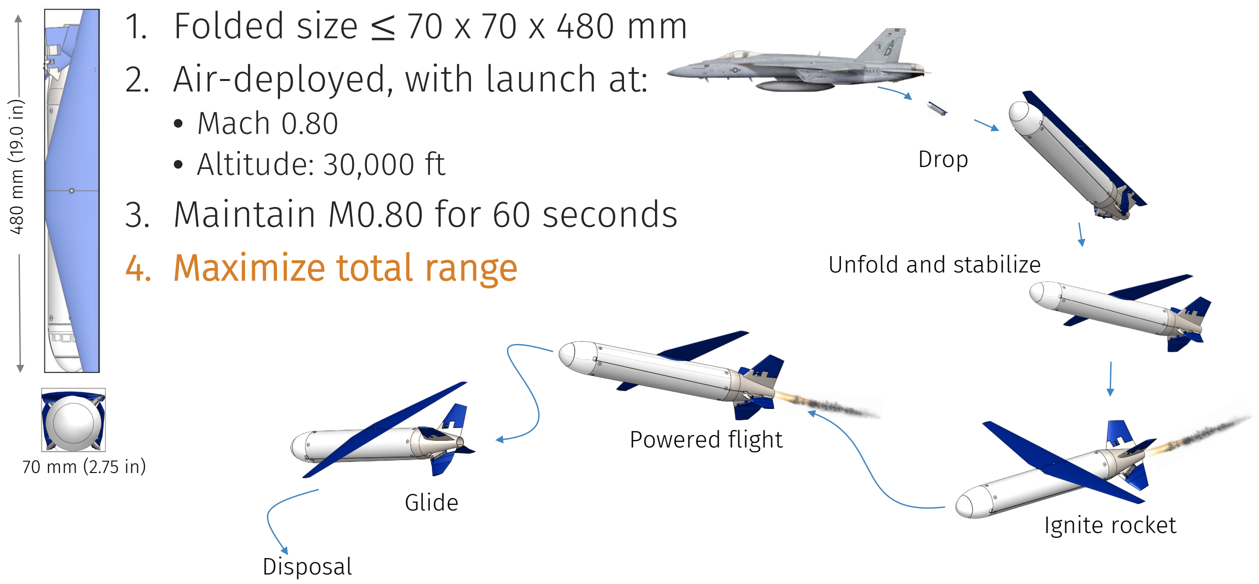


Figure 4-2: Concept of operations for the MIT Firefly UAV, adapted from Vernacchia [11]. CAD renders prepared by Julia Gaubatz [12]. F/A-18 image reproduced from McDonnell Douglas.

As explained by Vernacchia [11], Firefly aims to fill a previously-unexplored "small and fast gap" in the aircraft design space, with sustained cruise speeds of Mach 0.8 and a gross weight of 2.2 kg[4].

[4]For comparison: most air vehicles at this size scale are propeller-driven and fly at roughly Mach 0.2 or less. Most air vehicles capable of this sustained speed have gross weights at least an order of magnitude larger.

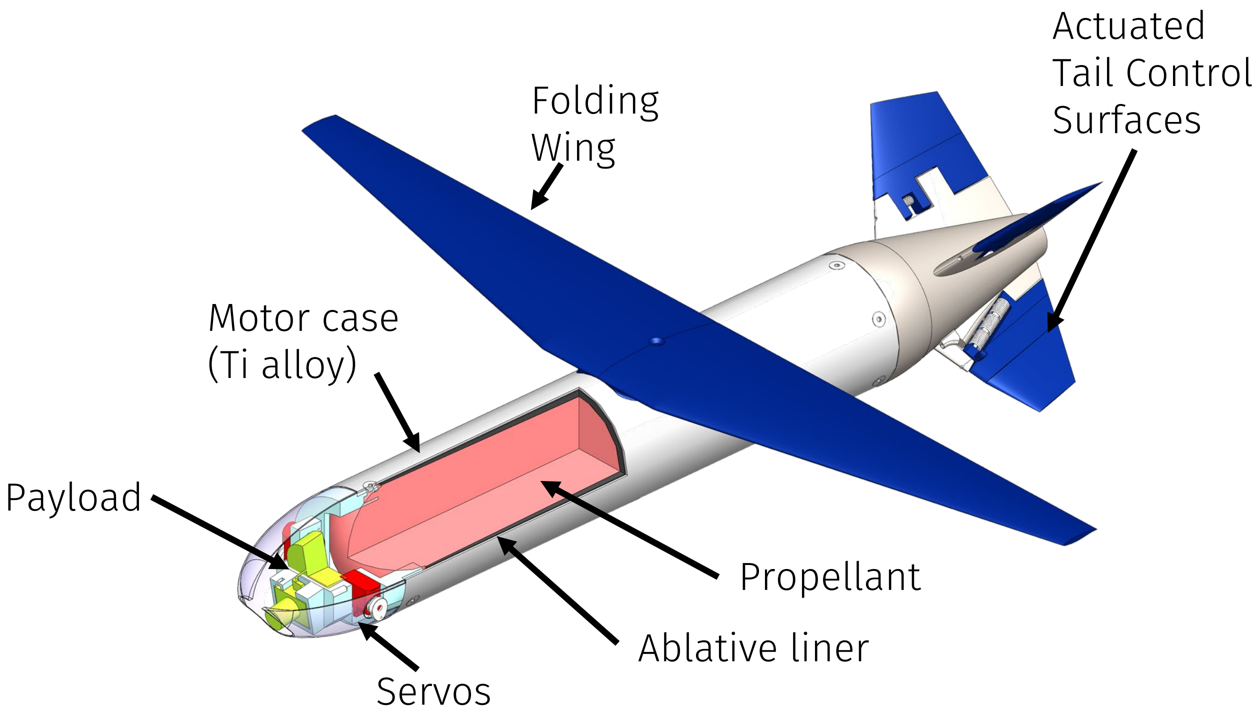


Figure 4-3: Cutout-view of MIT Firefly, showing various unique features. CAD render prepared by Julia Gaubatz [12].

Before even beginning the computational design process, we can observe some high-level design drivers from first principles. First, the vehicle is relatively insensitive to mass, which contrasts strongly with most aircraft design problems. This is because of a few reasons:

- The vehicle is air-launched at the dash speed and altitude, which means that no additional propulsive energy need be expended to reach this state. Although some of this initial kinetic energy is lost to drag during the initial stabilization transient upon deployment[5], this is a relatively minor effect.
- During the dash phase, the freestream dynamic pressure is extremely high relative to the vehicle's projected area. In fact, were it not for the glide phase, the folding wing shown in Figure 4-2 could be discarded, and body lift alone would be easily produce sufficient aerodynamic lift. Because of

[5]and added mass tends to increase the moments of inertia, which slows down the short-period flight dynamics mode

where this vehicle operates on the drag polar during the dash phase, the drag force is relatively insensitive to lift, and hence, vehicle weight.

- During the glide phase, the vehicle can be trimmed to any preferred airspeed, and naturally, the range-maximizing trajectory will have the aircraft fly at the best-glide speed. The glide ratio is essentially invariant to vehicle mass[6], so the range is also relatively insensitive to vehicle weight.

Because of this insensitivity to mass and the focus on total range, one can predict that the major design drivers for Firefly should essentially distill into the following:

- To maximize range during the dash phase: a) pack as much useful propulsive energy into the specified volume as possible, such that this dash phase can be sustained as long as possible and b) reduce drag, even at the expense of lift.
- To maximize range during the glide phase: achieve the highest L/D ratio possible.

This first-principles design driver calling for the highest-possible volumetric energy storage naturally leads to hydrocarbon fuels[7]. To achieve reasonable propulsive efficiency at Mach 0.8, the practical propulsion architectures are either a turbine engine or a rocket engine (which may be solid, liquid, or hybrid); tradeoffs are discussed by Vernacchia [11]. In short, a solid rocket engine requires far less "volume overhead" to be spent on the engine itself than any other option, especially at this size scale[8]. This allows the most possible volume to be spent on energy-rich propellant.

These design drivers also allow one to easily foresee that volume allocations will come at a premium when designing Firefly: expanding the volume available for propellant will inevitably result in fuselage

[6]barring some slight Reynolds number effects due to best-glide speed change, which are higher-order effects.

[7]as opposed to electric batteries

[8]Though first-principles physics favor a solid rocket, implementing this comes with some challenges. A particularly interesting one is how to slow down the burn such that the total impulse can be spread across a long enough duration. Vernacchia [11] and Mathesius [110, 152] explore a mix of clever chemical and physical strategies to achieve this.

shapes that generate more wave drag at the Mach 0.8 dash condition[9]. This sets the stage for an interesting multidisciplinary design optimization problem, where these coupled relationships can be explored further.

4.2.2 Aircraft Design Problem Formulation

The design of Firefly is formulated as a combined vehicle-and-trajectory optimization problem, as these two are inextricably linked. The trajectory is represented via a predetermined number of discrete points in time (roughly $N = 200$), which are connected via a direct collocation method[10]. This is depicted in Figure 4-4.

Because the flight physics fundamentally change between dash and glide phases (as the rocket motor no longer produces thrust after this point, and some aerodynamic changes regarding base drag occur), Firefly is inherently a multi-phase dynamics problem. Accordingly, the discretized dynamics are partitioned into two phases. Critically, to maintain C^1-continuity (and hence, optimization friendliness), we employ a strategy that we call *stretchy time* that ensures that any given discretization node never crosses this phase boundary, as this would cause a discontinuous change in the problem physics. In this strategy, the time values corresponding to phase transitions (e.g., when rocket motor burnout occurs; when the vehicle contacts the ground and terminates the mission) are free variables in the optimization problem. In contrast, the time values corresponding to intermediate points have their relative spacing within a phase fixed *a priori*. (In this case, this is done with "cosine-spaced", also called Chebyshev node, relative positions in time.) Points in time that fall exactly on a phase boundary are included in both phases, with a zero-time-difference collocation constraint that stitches the two phases together; this adds a few degrees

[9]because these higher-volume shapes are blunter, and hence cause a deeper low-pressure spike resulting in stronger shock formation

[10]A introduction to the math representing the dynamics here is given by Kelly [153].

of freedom but greatly simplifies indexing.

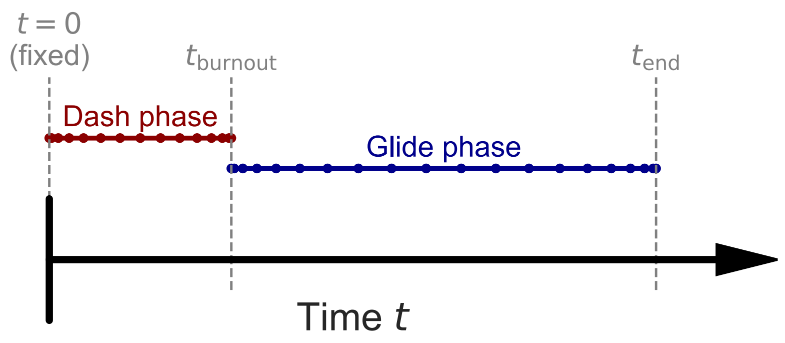


Figure 4-4: "Stretchy time" time discretization strategy used for the MIT Firefly MDO problem and other multi-phase combined-vehicle-and-dynamics optimization problems.

The physics formulation of the optimization problem can be summarized as follows:

MIT Firefly MDO Problem Formulation

- **Objective Function**: Maximize the total mission range, defined as the sum of the dash and glide phases.
- **Design Variables** (in total, 3,910 variables):
  - A series of trajectory design variables, including downrange distance, altitude, airspeed, flight path angle, and angle of attack. Note that this parameterization a) is two-dimensional, representing the trajectory in range-altitude space and b) makes a quasi-steady assumption[a], which essentially assumes that the short-period mode is much faster than the meaningful trajectory dynamics[b]. This quasi-steady assumption does not, however, assume that the vehicle is in force

equilibrium at each time point; the "raw input" of the control algorithm can be thought of as the angle of attack.

- Various other time-dependent variables, such as the control surface deflections, instantaneous fuel mass, chamber pressure, and nozzle exit pressure. These variables dynamically change vehicle mass properties (and hence stability and control) as well as propulsion performance (e.g., specific impulse), causing both to vary at each discrete time point that is analyzed.
- The vehicle design itself, which includes:
  - Geometric shape variables. The wing and tail geometries are parameterized by a collection of planform variables; airfoils were optimized separately, as the integrated high-dimensional airfoil shape optimization capabilities described in Chapter 7 were not yet developed at the time of this study. The general fuselage shape topology was fixed *a priori* based on manufacturing considerations detailed by Vernacchia [11], consisting of an ellipsoidal nose, a cylindrical midsection, and a conical boattail[c]. Various dimensions of this fuselage were allowed to vary, such as the fineness ratio of the nose (which has important wave drag implications) and the angle of the boattail. The relative position and incidence of all lifting surfaces was also left free.
  - Propulsion design variables. These include both chemistry considerations (e.g., the mass fraction of oxamide, a burn rate suppressant) as well as physical parameters (e.g., throat diameter, expansion ratio, ablative liner

thicknesses)

    * Variables giving a detailed component-wise weight and volume breakdowns of the aircraft, which allows satisfaction of implicit structural analysis models, as well as mass properties modeling. This approach allows natural inclusion of components like ballast, which may be needed for acceptable handling qualities.

Constraints (in total, 8,420 constraints), which are primarily drawn from six interacting disciplinary analyses:

- **Aerodynamics**: a component-wise buildup of lift, drag, and moment using AeroSandbox's AeroBuildup model, detailed in Section 5.3.
- **Stability & Control**: a linearized flight dynamics modal analysis, which uses stability derivatives computed by directly differentiating the aerodynamics model and constrains the spectral characteristics of the short-period, Dutch roll, and spiral mode dynamics.
- **Structures & Mass Properties**: weight closure and structural analysis using first-order models, calibrated to test articles for as-built Firefly prototypes.
- **Propulsion**: A 1D nozzle flow analysis (and in later versions, a detailed reacting chemical kinetics model implemented by Mathesius [152]), which computes the thrust and specific impulse of the rocket motor at each time point.
- **Trajectory / Dynamics**: A direct collocation method, which enforces the equations of motion at each time point, as well as the boundary conditions

at the start and end of each phase.

- **Volume accounting / Packaging**: A series of geometric constraints that ensure that the vehicle fits within the specified stowage volume[d], as well as that the propellant grain fits within the pressure chamber.

[a] using the definition from Drela [154]

[b] Equivalent ways to state this are that we assume the vehicle an instantaneously trim to a desired angle of attack, or that the nondimensionalized angular rates (e.g., $\bar{p}$, $\bar{q}$, $\bar{r}$) are very small.

[c] Original versions of this problem allowed far more geometric degrees of freedom, such as non-circular fuselages. Research in collaboration with Vernacchia [11] revealed that non-circular fuselages lead to unacceptable solid rocket grain cracking due to the pressure chamber's deformation under high pressure.

[d] A notable constraint is that the trailing edge of the main wing cannot be swept backwards for packaging reasons, as the wing is a single rotating piece. A two-part jackknife wing (e.g., *MIT Perdix* [53]) was considered, but we find that the propellant volume penalty of the added mechanisms thickness does not outweigh the aerodynamic gain.

Because every analysis module used in this Firefly design problem is compatible with a code transformations paradigm, formulating this as an "all-in-one" optimization problem within a simultaneous analysis and design (SAND) architecture is straightforward [28, 142]. This overall MDO problem architecture is loosely depicted in Figure 4-5, where most constraints are formulated implicitly. (This contrasts with nested approaches to closure, which is shown in Figure A-1)

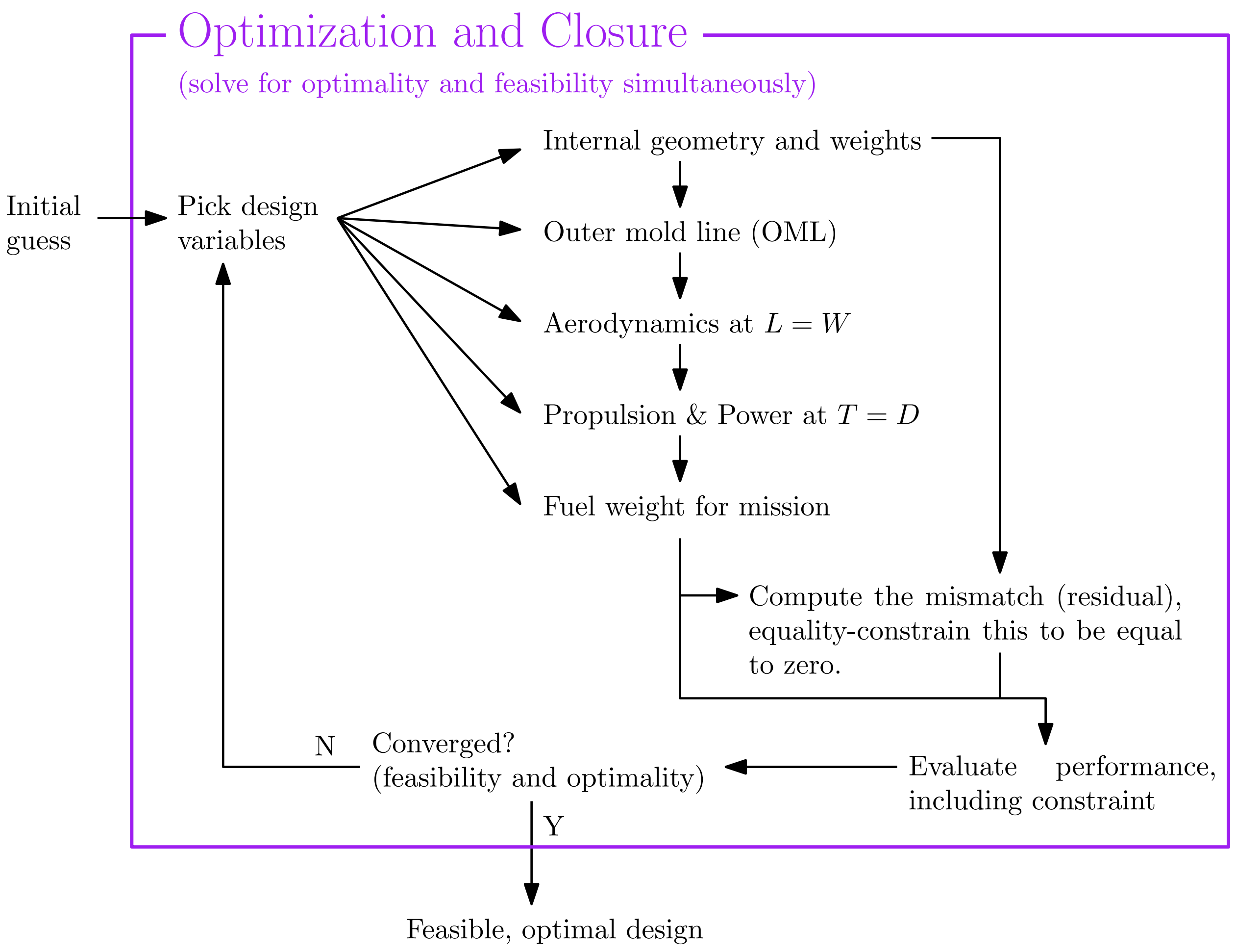

Optimization and Closure
(solve for optimality and feasibility simultaneously)
Initial guess
Pick design variables
Internal geometry and weights
Outer mold line (OML)
Aerodynamics at $L = W$
Propulsion & Power at $T = D$
Fuel weight for mission
Compute the mismatch (residual), equality-constrain this to be equal to zero.
Evaluate performance, including constraint
Converged? (feasibility and optimality)
N
Y
Feasible, optimal design

Figure 4-5: Schematic of the simultaneous analysis and design (SAND) architecture used for the MIT Firefly MDO problem.

The Firefly MDO problem is relatively high-dimensional, with 3,910 decision variables and 8,420 constraints, many of which are nonlinear and nonconvex. Evaluation of the problem constraints and objective take the bulk of the runtime; in particular, the $N = 200$ workbook-style aerodynamic analyses (one for each discrete point along the trajectory) performed at each iteration tend to be the most computationally expensive elements. Nevertheless, using AeroSandbox with CasADi and IPOPT backends, the problem converges to a solution in a wall-clock runtime of just 6.9 seconds on a laptop[11]. Equally notable is the fact that this problem can be implemented using only 613 lines of Firefly-specific

[11]with a Ryzen 5800H CPU

Python code, which is possible because many of the constituent analyses use general-purpose aerospace physics models that are optionally provided by AeroSandbox. This speed and conciseness allows the user to rapidly and interactively explore the design space, as well as to perform sensitivity analyses and trade studies.

4.2.3 Results

The point results of the Firefly design optimization problem yields the vehicle design shown in the CAD render of Figure 4-3. However, one of the first compelling framework-level observations that was made during this case study was how useful it is to be able to immediately visualize the aircraft geometry within seconds of solving the optimization problem. As quipped by aircraft designer Bob Liebeck[12], "you can tell a lot about an airplane by whether it passes the TLAR: 'That Looks About Right' ." Therefore, this case study was the motivation for building the AeroSandbox aircraft geometry stack, a code-transformations-compatible library that allows for rapid visualization of new designs. The raw outputs of this geometry stack are shown in Figure 4-6, and the resemblance between this initial OML render from AeroSandbox and the final CAD render of Figure 4-3 is apparent.

[12] personal correspondence

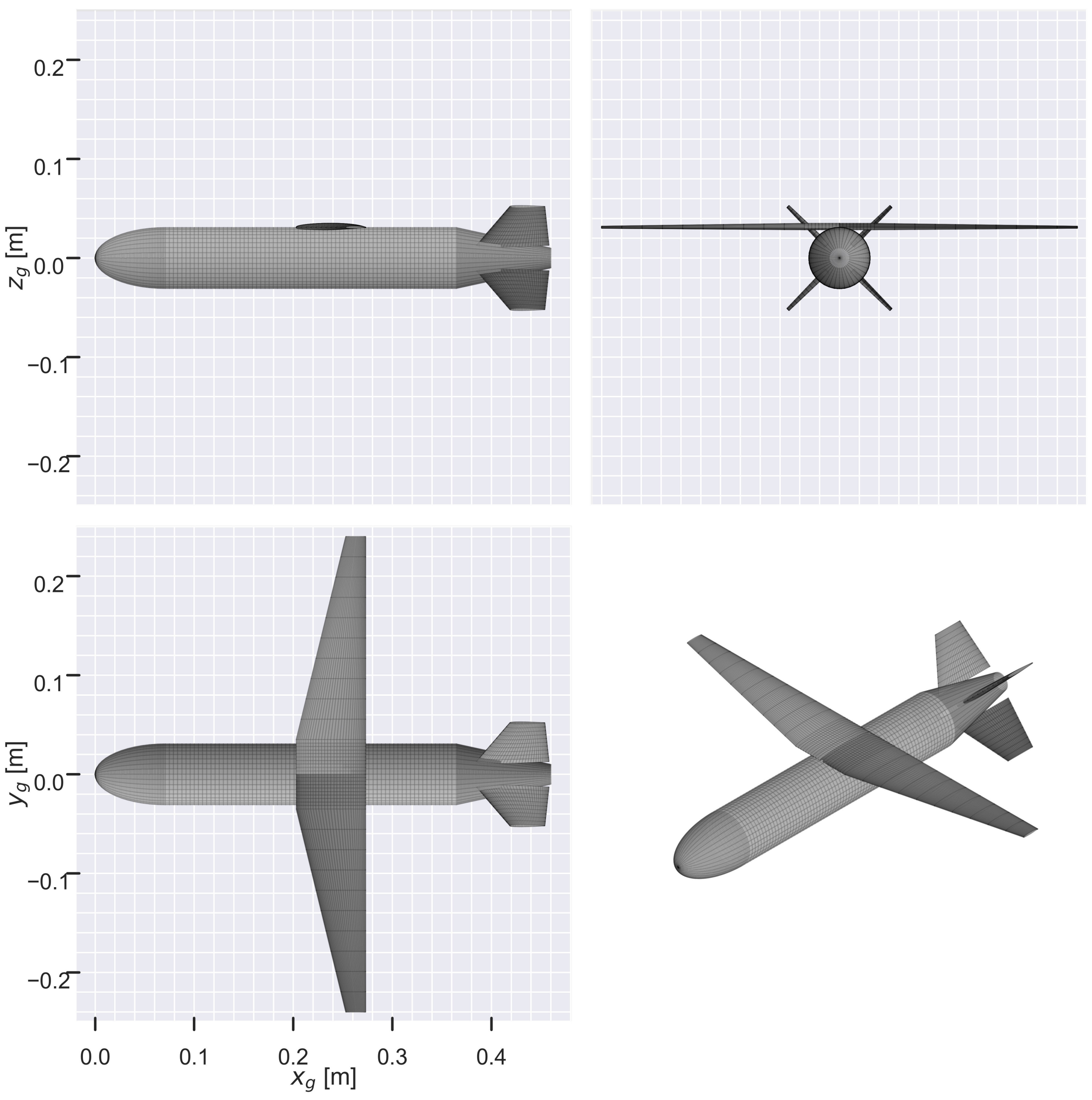


Figure 4-6: Raw user-facing output of the AeroSandbox geometry stack for the MIT Firefly UAV.

Internal volume and mass accounting of various components at the design point are shown in Table 4.2, with values that were later validated against as-built prototypes.

The resulting trajectory, which was simultaneously optimized with the vehicle, is shown in Figure 4-7. Here, a number of notable findings can be observed:

Table 4.2: Mass and volume accounting of various components of the MIT Firefly UAV, at the point design resulting from the formulation given in Section 4.2.2. Mixed units are the result of preferences by various project stakeholders.

Component	**Internal volume [in^3] (Percentage)**	**Mass [gram] (Percentage of gross)**	**x_g of CG, relative to nose datum [mm]**
Fuel (at gross)	38.8 (54.2%)	1019 (42.7%)	321
Battery	9.8 (13.7%)	322 (13.5%)	58
Case	7.9 (11.0%)	579 (24.2%)	230
Liner	5.7 (7.9%)	136 (5.7%)	321
Mechanisms	4.1 (5.8%)	20 (0.8%)	447
Payload	2.5 (3.5%)	100 (4.2%)	78
Avionics	2.1 (2.9%)	75 (3.1%)	100
Nozzle	0.7 (1.0%)	15 (0.6%)	447
Wing	-	53 (2.2%)	188
Tails	-	70 (2.9%)	436
Total (gross)	71.5 (100%)	2389 (100%)	248
Total (zero-fuel)	71.5 (100%)	1370 (57.3%)	195

1. The vehicle is capable of achieving far greater operational range than had been initially anticipated. Initial hand-calculated estimates based on an assumed vehicle design and trajectory had estimated that total ranges of 70 kilometers might be possible[13], while the combined optimization result yields ranges of over 230 kilometers.

2. This massive range increase is achieved partially by co-optimization of various aircraft subsystems, like making aeropropulsive trades on the fuselage shape, which influences dash range. This was expected, and is typical of the power of an MDO approach. However, the majority of the range increase was due to the unique trajectory that the optimizer found, which is shown in Figure 4-7. Here, the vehicle rapidly climbs to a very high altitude—well into the stratosphere—before gliding down. This allows the vehicle to take advantage of two clever effects:

[13]As a rough napkin-math calculation, a 60-second dash at Mach 0.8, followed by a glide at a L/D of 7 (owing to the low Reynolds number and assumed low-aspect-ratio wing) yields a range of 70 km.

(a) The arcing trajectory effectively lets the vehicle use gravitational potential energy as a battery. Because of this, the burn duration of the solid rocket motor can be much shorter for a given impulse, as excess energy goes somewhere useful (i.e., altitude) rather than somewhere wasteful (i.e., wave drag). By allowing a shorter burn, the propellant chemistry can be tweaked to use less burn rate suppressant (oxamide), which increases the specific impulse of the motor. This dynamics-propulsion coupling was not foreseen.

(b) By climbing high, the *indicated* airspeed of the vehicle is much lower while holding the Mach 0.8 constraint. This reduces the dash-phase drag penalty of having a large, high-aspect-ratio, high-L/D wing. By increasing the wingspan, the glide-phase L/D becomes much larger, which gives a large range increase. In addition to enhancing the glide ratio, climbing high also simply increases the glide range directly. This inter-phase aerodynamic trade was also unexpected.

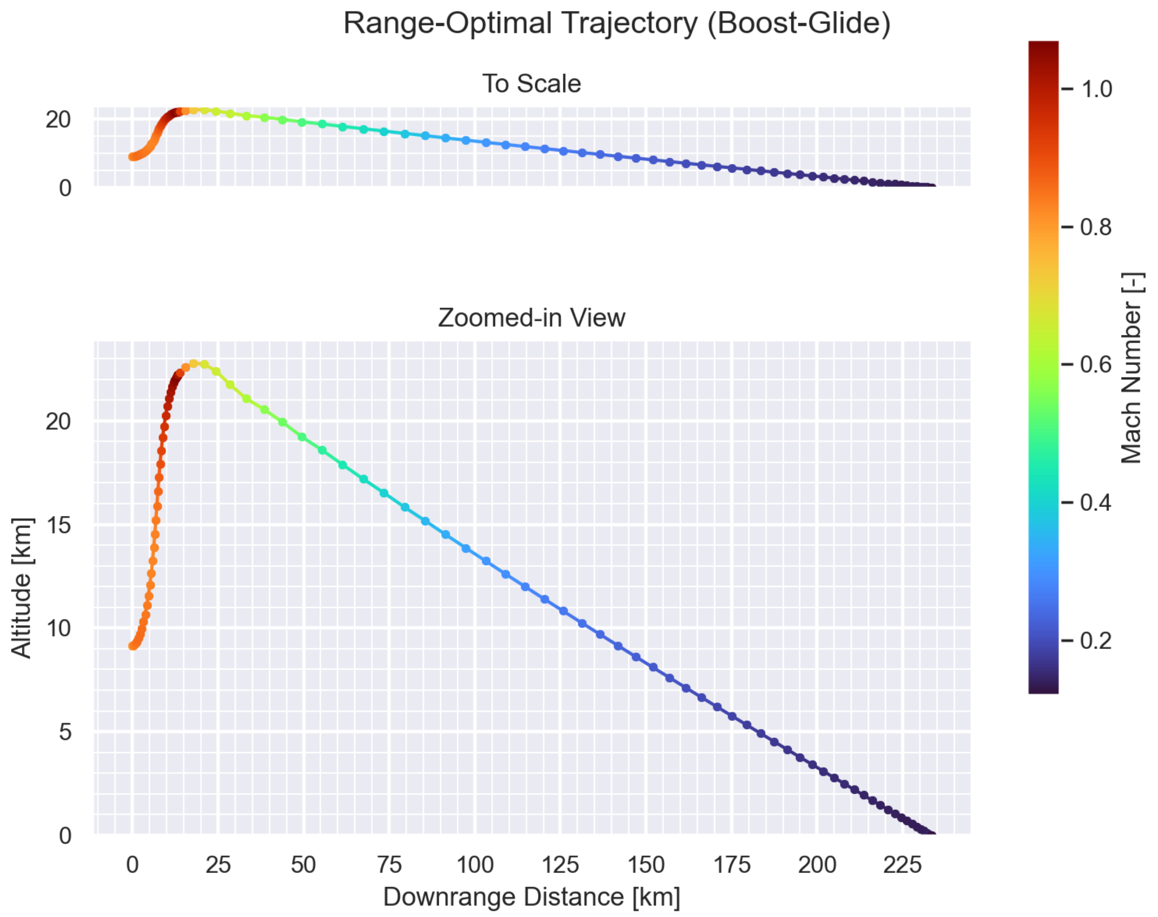


Figure 4-7: Trajectory that maximizes vehicle total range, resulting from the combined vehicle-and-trajectory MDO problem formulation given in Section 4.2.2.

This trajectory, while unexpected, indeed satisfies requirements, and its discovery was only possible through the use of a combined vehicle-and-trajectory optimization approach. This is a key benefit of the code transformations paradigm: the ability to rapidly explore the design space and discover unexpected interactions between subsystems.

4.2.4 Design Space Sweeps

As discussed further in Appendix B, the value of an engineering design optimization framework is not just in the point design, but also in the insight that it provides. An early example of this observation was made during the Firefly design study, where the optimizer was used to explore the tradeoffs between optimizing for total range (dash + glide) vs. optimizing for dash range alone. In AeroSandbox, this can be easily performed, all without leaving a procedural coding style with the following syntax:

1. Replace the objective function with a simple linear combination of the total and dash ranges, with a weighting factor $w \in [0, 1]$. Define w as an optimization parameter, using:

```python
import aerosandbox as asb
opti = asb.Opti()  # Initialize an optimization environment
... # Define the rest of the optimization problem

w = opti.parameter()
```

2. Instead of solving the problem with `opti.solve()`, solve it with:

```python
sols = opti.solve_sweep({w: np.linspace(0, 1)})
```

Using this syntax, the user can convert a point design optimization problem to a design sweep (i.e., looking at a set of optimal points, as some parameter is varied) in just two lines of code.

For this particular case study, the results of such a sweep are shown in Figure 4-8. Clearly, though designs can be optimized purely for total range or only for dash range, there are many opportunities where a small sacrifice in one can yield large gains in the other—this may be of interest to the practical designer. Likewise, it shows how the vehicle resulting from the optimizer (in the lower-right) differs dramatically from the original napkin-math "mental model" that was used for initial performance estimates. Because this initial design did not use an arcing trajectory, the wing area (and hence, glide-phase L/D) was quite limited.

Charts like these can be crucial in high-level conceptual decision-making, where an entire family of aircraft can be depicted in a single chart. The results of this design sweep take a few minutes to compute in serial, but parallelization across cores or computers can allow this to be presented to the user in seconds.

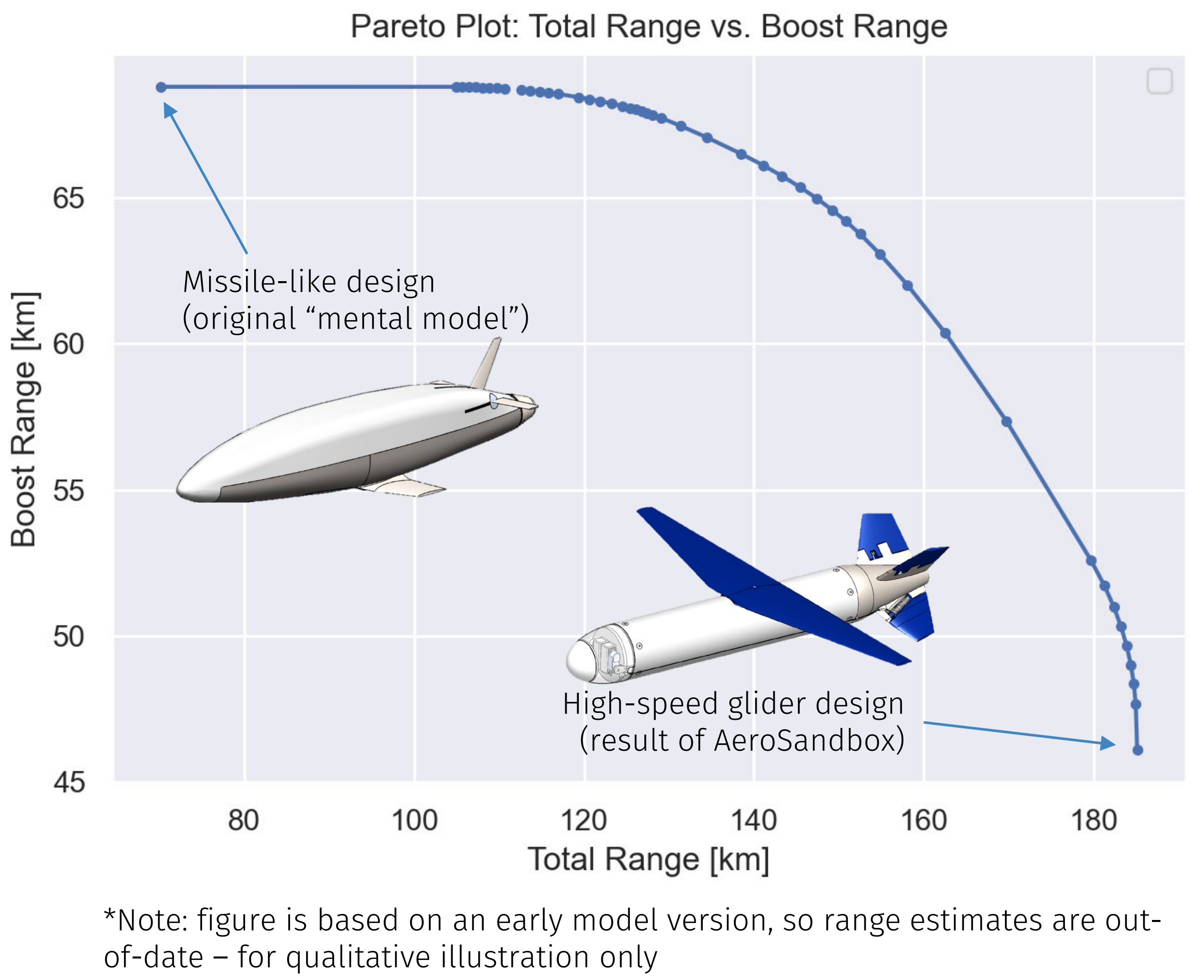


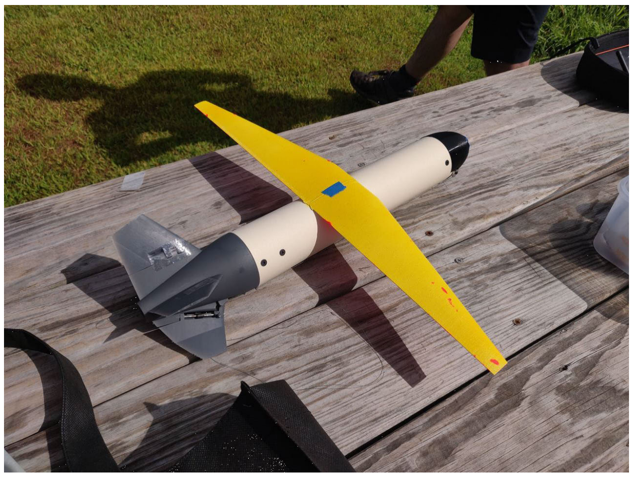
Figure 4-8: Pareto front of the MIT Firefly UAV design space, showing the tradeoff between total range and dash range. Each point represents both a unique vehicle design and trajectory.

4.2.5 Supporting Flight Test

Finally, AeroSandbox-based performance calculations were used to support the Firefly air vehicle through manufacturing, first flight, and a controls characterization flight test campaign, as described further by Gaubatz [12]. Media from this flight test campaign are given in Figure 4-9. The vehicle was found to be stable and controllable, and the flight test data was used to validate the aerodynamics model used in the optimization problem.

This experience also motivated the development of purpose-built aircraft flight dynamics tools within AeroSandbox, which can be used to directly reconstruct and visualize the flight trajectory of the vehicle. This can be performed for both computationally-optimized trajectories as well as from actual measured flight test data, allowing both design and post-flight analysis to be conducted in the same environment for easy model validation. An example of this is shown in Figure 4-10, which shows the trajectory of Firefly's first flight test, reconstructed and visualized in AeroSandbox. Notably, during this flight test the vehicle was air-dropped from a quadcopter[14], which can be seen in the steep vertical trajectory at the beginning of the flight as the vehicle stabilizes.

This experience emphasized how useful it is to have an MDO tool that can switch between analysis (e.g., only closure, no optimization) and design modes, as discussed in the birds-eye view of optimization shown in Figure 2-2. An MDO tool, if built correctly, has the potential to follow the vehicle through its lifecycle similar to a digital twin, providing value from the conceptual design phase through to flight test.


(a) As-built Firefly air vehicle, prepared for first flight.

(b) Photograph of the MIT Firefly air vehicle in-flight, cruising at roughly 40 m/s.

Figure 4-9: Photos of the MIT Firefly prototype. Vehicle constructed by Julia Gaubatz [12].

[14]with further details given by Gaubatz [12]

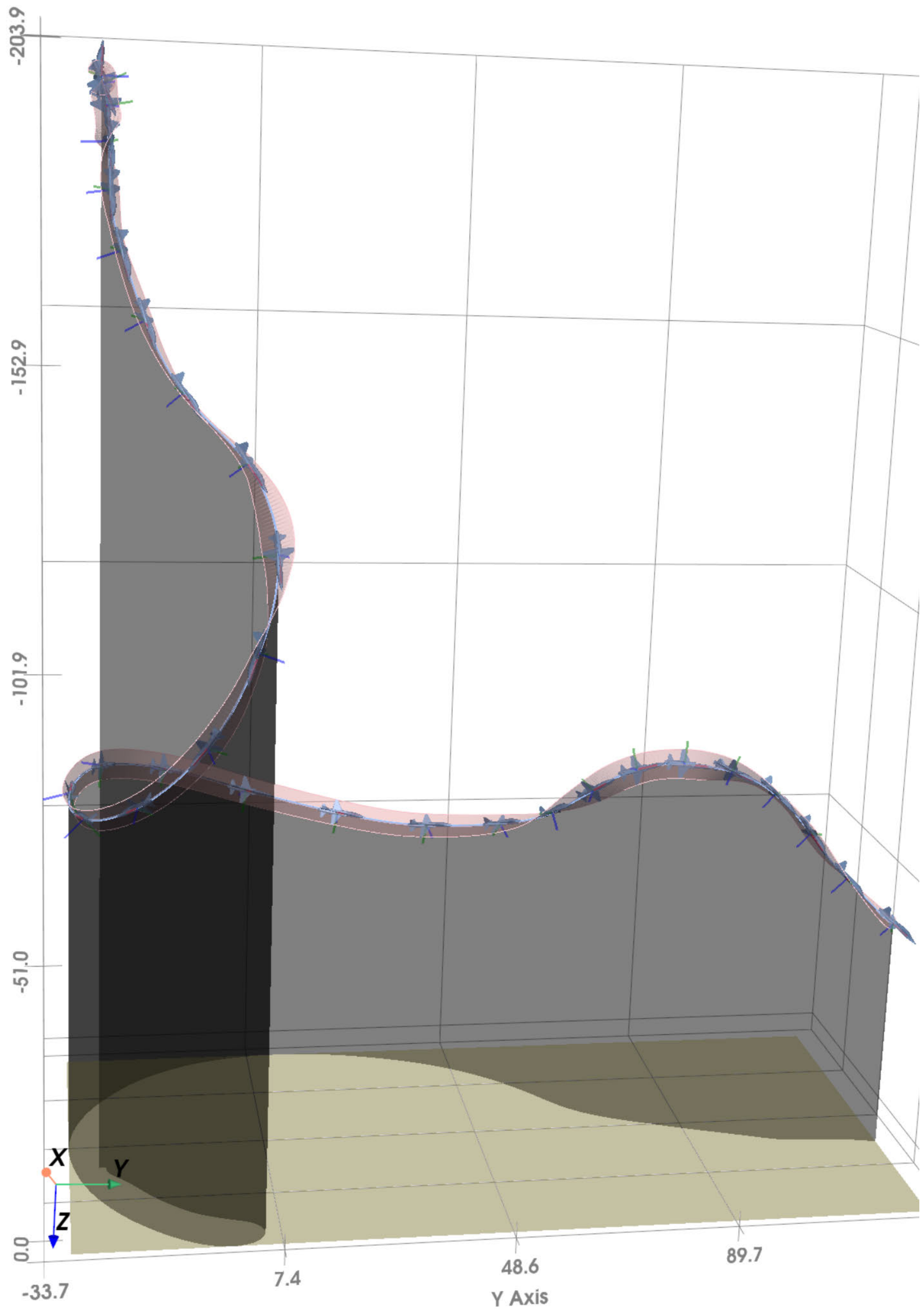


Figure 4-10: Flown trajectory of the first flight of MIT Firefly, reconstructed and visualized in the AeroSandbox dynamics stack. Vehicle is drawn using a generic aircraft model and at roughly 30x scale, which allows easier visualization of vehicle orientation. Flight test was air-dropped from a quadrotor, with the stabilization period visible at the beginning of the trajectory.

4.3 MIT Dawn (Electra.aero SACOS) Solar-Electric HALE Aircraft

4.3.1 Problem Overview

A second aircraft design case study that contributed heavily to the development of AeroSandbox was *MIT Dawn*, a solar-electric high-altitude long-endurance (HALE) aircraft. This aircraft development program was later transferred to industry partner Electra.aero and incorporated into the Stratospheric Airborne Climate Observatory System (SACOS) program, where the aircraft has achieved first flight and is undergoing further development.

The Dawn aircraft development program began as a set of solution-agnostic requirements aimed at supporting a climate science mission [13, 155, 156]. This science mission, conceived by an atmospheric chemistry research group at Harvard[15], aims to take atmospheric chemistry measurements of various free radicals in the stratosphere, which must be performed in-situ over long durations. This information would allow scientists to further reduce the uncertainty about the rate of climate change, which could allow for more precise and effective policy measures to be implemented [157]. A variety of aircraft design requirements were developed and negotiated with the science team, with the major driving ones as follows:

- **Payload**: 30 kg science package, with 500 W electrical power draw during daylight hours and 150 W at night
- **Altitude**: sustained flight at 60,000 ft. MSL (18.3 km) or above

[15]The Anderson Research Group, under the direction of Professor James Anderson.

- **Endurance**: 6 weeks continuous flight during July–August
- **Station-keeping**: with the ability to maneuver and maintain position over any location in the continental United States, in 95th-percentile winds

We initially approached this aircraft design problem by exploring a wide range of conceptual solutions. In addition to a solar aircraft, the team considered a superpressure balloon (E.g., Google Loon), a powered airship, a rotating fleet of conventional aircraft (e.g., Aurora Flight Sciences Orion), long-endurance hydrogen aircraft (E.g., AeroVironment Global Observer), ground-tethered balloons, and numerous others[16]. Each of these had a rapid feasibility analysis conducted in AeroSandbox based on first-principles physics, similar in scope to the SimpleAC problem given in Section 4.1. This capability to quickly pose sizing studies allowed these proposed concepts to be matured to the point where a meaningful and fair comparison[17] could be made; ultimately, a solar-powered HALE aircraft was selected, with further reasoning available in Sharpe et al. [13].

4.3.2 Vehicle Overview

The Dawn aircraft generally has a sailplane-like conventional configuration, as shown in Figure 4-11. On the outboard sections of the wing, two small all-moving aerodynamic surfaces are mounted on booms extending from the wing trailing edge. These two unique surfaces, called *tailerons*, are used for roll control and aeroelastic stabilization; further details on these surfaces are given in recent work by Sharpe, Ulker, and Drela [158]. Propulsion is provided by two propellers, which are wing-mounted and driven by electric motors. The payload is held in a separate pod, mounted with a truss beneath the main wing to allow for propeller clearance.

[16] Even some nuclear-powered aircraft concepts were briefly considered.

[17] i.e., a true apples-to-apples comparison, where an *optimized* variant of concept A is compared to an *optimized* variant of concept B

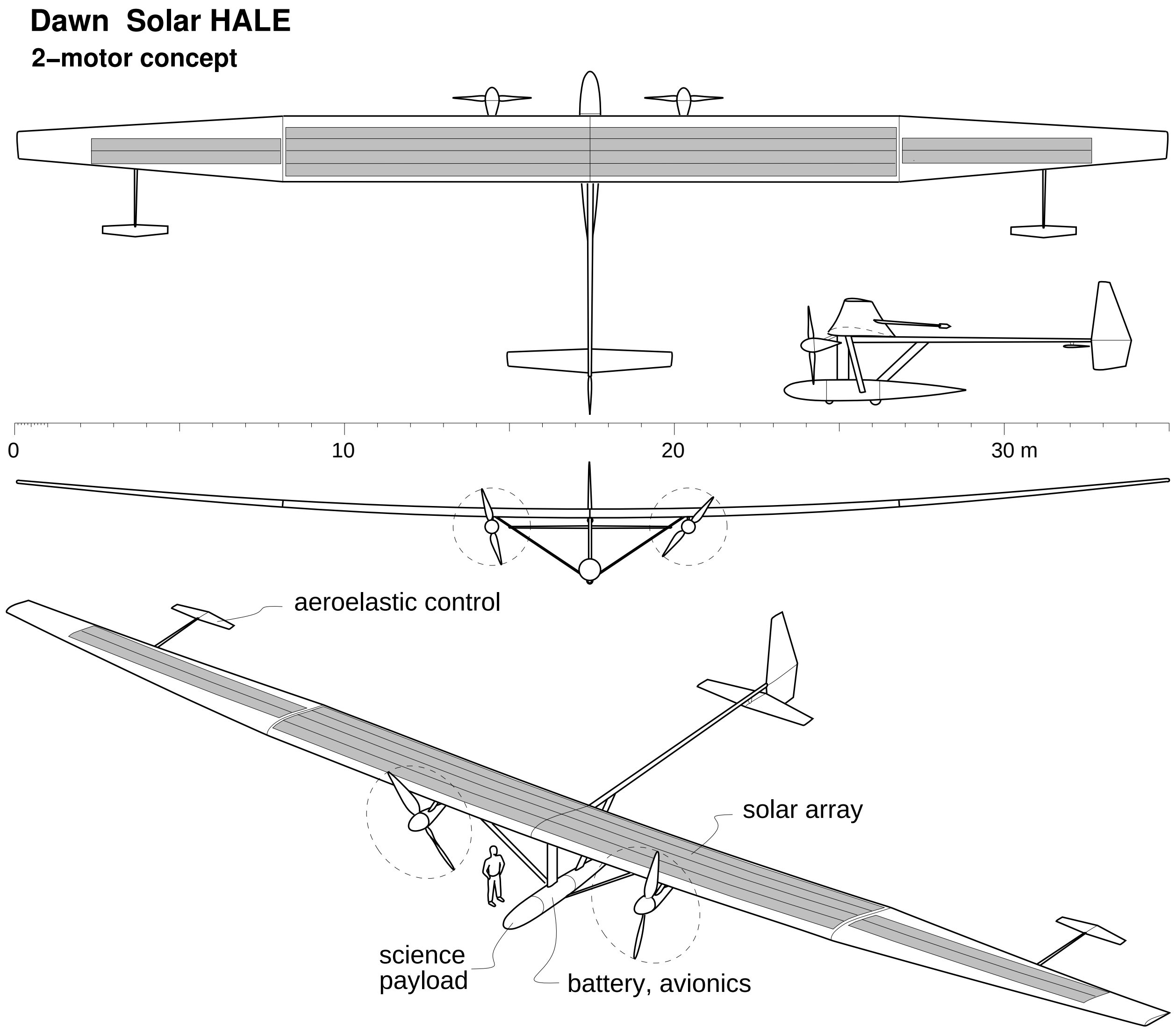


Figure 4-11: Basic aircraft configuration, approximate scale, and major components of the Dawn solar-electric HALE aircraft. Exact aircraft dimensions vary depending on technology assumptions. Figure illustrated by Mark Drela.

The aircraft is powered by a series of solar cells, which are mounted on the main wing in spanwise strings[18]. Approximately 80% of the wing's surface is covered with solar cells, with this solar area fraction limited by the curvature tolerance of cells when mounted near the leading edge. Various solar cells were considered, with the baseline design using Sunpower C60 cells with a cell-level efficiency of 24.3%[19].

[18]This arrangement keeps each cell within the string at similar solar incidence angles, which makes them impedence-matched and results in fewer power-point-tracking losses.

[19]Further modifications to the actual cell energy generation based on line-of-sight airmass absorption, atmospheric refraction, backscattering, surface albedo, covering transparency, and other effects are also considered.

Overnight flight is supported by a large battery, which is mounted in the pod. Due to the project's experimental nature, relatively advanced battery technologies are assumed; the baseline design assumes a cell-level battery specific energy of 450 Wh/kg[20]. Pack-level specific energy is slightly lower, as we assume the cells form 89% of the overall pack mass; this is based on as-built weights from the prior Aurora Odysseus solar aircraft.

The aircraft cruises at an altitude of between 60,000 and 65,000 ft. MSL, which is above the tropopause at most latitudes and seasons. This essentially nullifies the possibility of cloud cover, reduces exposure to aeroelastically-risky gusts, and reduces steady wind speeds[21] (reducing the required cruise speed for stationkeeping). An interesting mission strategy that is used here (which was discovered using the MDO process described in Section 4.3.3) is the idea of *altitude cycling*: during the day, the aircraft recharges its batteries until approximately 3 p.m. local solar time, after which it uses the excess solar power to climb higher. During the night, the vehicle glides lower (though never below 60,000 ft.), which reduces power draw and required battery mass. This essentially allows the vehicle to use gravitational potential energy as a battery, which is a similar clever combined-vehicle-and-trajectory optimization exploit that was found in the Firefly design.

Table 4.3 summarizes various key parameters of the aircraft in its cruise condition, and a high-level mass budget is given in Figure 4-12. From this figure, the difficulty of designing a solar-electric HALE aircraft with overnight endurance is apparent: the battery is nearly half the gross weight of the aircraft, and the payload mass fraction is a mere 8%.

[20]This is already achievable by a few commercially-available cells at the time of writing, such as the Amprius Technologies silicon-anode cells and Sion Power lithium-sulfur cells. These cells use exotic chemistries that come with some (acceptable) design considerations; further discussion is available in Sharpe et al. [13].

[21]At most points in the latitude-seasonality space, this altitude is above the jet stream.

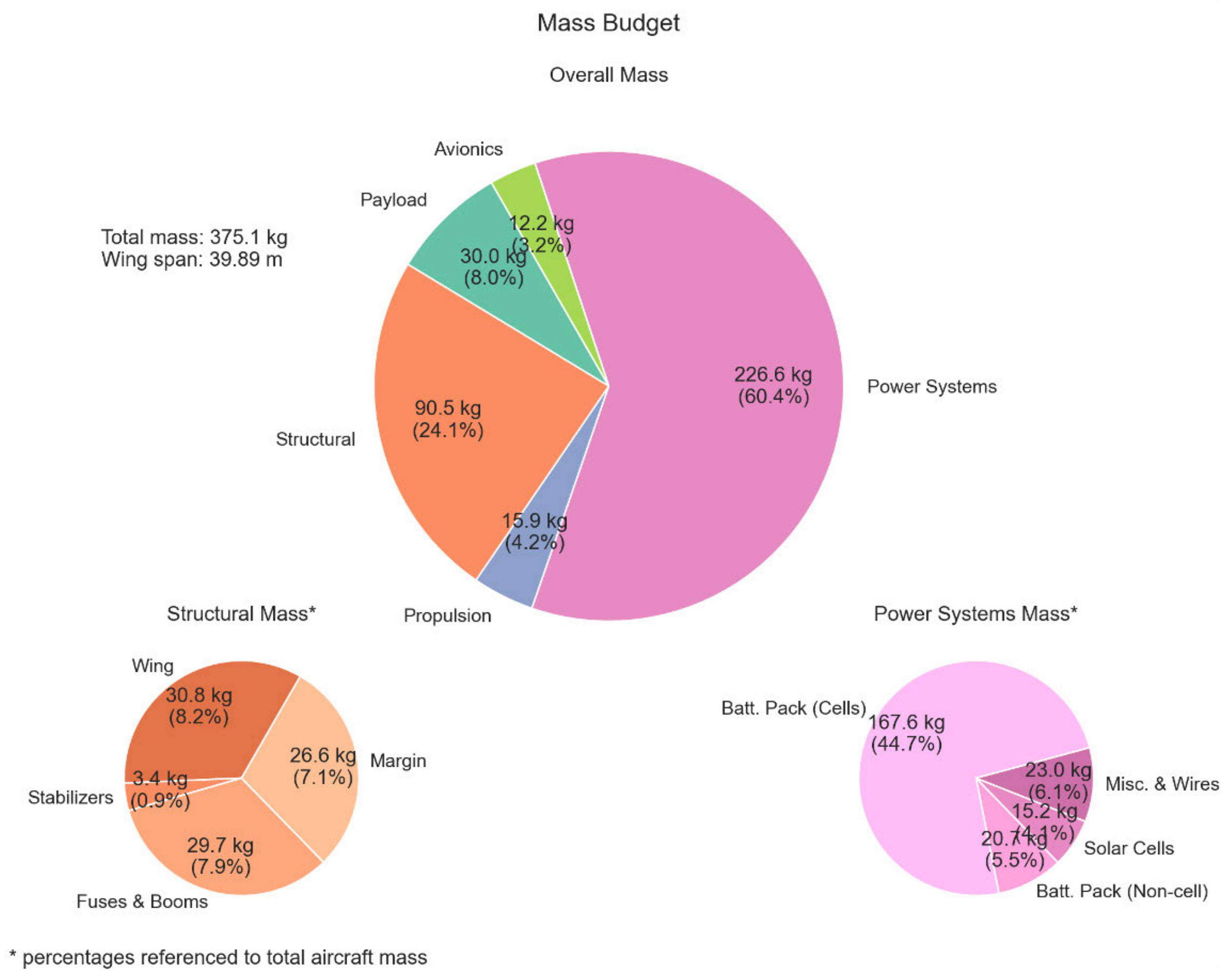


Figure 4-12: High-level mass budget for the Dawn solar-electric HALE aircraft. Large pie chart shows total breakdown, with the smaller pie charts showing subsystem-level breakdowns for the structural and power systems components. Reproduced from Sharpe et al. [13].

Table 4.3: Key specifications for the point design corresponding to the baseline mission. Reproduced from Sharpe et al. [13].

Figure of Merit	**Value at Design Point**
Gross weight	375 kg
Wingspan	39.9 m
Wing aspect ratio	23.4
Wing area	67.9 m^2
Wing loading	54.2 Pa
Cruise airspeed (true)	30.4 m/s
Altitude	18.7 km nighttime and peaking at 19.7 km on Aug. 31st
Total power output	Peak net battery draw: 5.07 kW
Battery capacity	75.4 kWh
Wing Reynolds number	383×10^3 ($c_{\text{ref}} = \bar{c}$)
Cruise lift coefficient	1.11
Cruise L/D Ratio	30.8

4.3.3 Aircraft Design Problem Formulation

This subsection includes content from the author's contributions to prior work by Sharpe, Dewald, and Hansman [13].

Just as with the Firefly design problem, the Dawn design problem was formulated as a combined vehicle-and-trajectory optimization problem. The trajectory represents a 24-hour cycle, beginning and ending at solar noon. This periodic interval is of interest here because of the inherent challenge of *diurnal energy closure*: a solar aircraft must be capable of ending any given 24-hour period within the mission envelope with more potential energy (in the forms of battery charge state and altitude) than it started with[22].

While sufficient energy generation is key to ensuring energy closure, energy storage also constrains

[22]Some additional energy margin must be included here, for robustness. In this study, we add energy generation margin by effectively decreasing the solar insolation in all conditions by a flat 5% knockdown.

the design space. The aircraft's battery must be sized so that its charge state remains within allowable bounds[23] - given the time-varying solar insolation and unsteady power draws (from the propulsion system, payload, and avionics), evaluating the feasibility of this constraint for a given solution becomes nontrivial. Furthermore, one would expect that a system with cyclic power injections might be optimally operated using a cyclic strategy; and indeed this is observed via altitude cycling under some assumptions.

To incorporate this trajectory information into the optimization problem, the flight equations of motion are once again embedded via a direct collocation method, with trapezoidal quadrature. A convergence study was performed for the solar aircraft design problem in order to determine an appropriate temporal resolution. Testing indicated that grid-independence was reliably achieved at a temporal resolution of 150 collocation points uniformly spaced over a 24-hour interval; therefore, this was the discretization resolution used in all further studies.

The physics formulation of the optimization problem can be summarized as follows:

MIT Dawn MDO Problem Formulation

- **Objective Function**: Minimize the wingspan of the aircraft. This choice of objective is somewhat unusual, and it is chosen as a proxy for measuring an important technical risk. Historically, the most common failure modes of solar aircraft have been aerostructural in nature[a], and wingspan is a good first-order measure of how severe these aeroelastic challenges will be. In addition, previous development programs for large solar aircraft (such as Aurora's Odysseus, at 74 meters wingspan) have attributed significant operational challenges (e.g., runway width, transport, support equipment, and cost) directly to wingspan.

[23] A maximum allowable battery depth-of-discharge of 85% was assumed.

- **Design Variables** (in total, roughly 2,500 variables)

  - Trajectory design variables, with a similar 2D parameterization as in Section 4.2.2. Unlike the Firefly problem, however, the steady assumption is taken further, and instantaneous force equilibrium is assumed at each time point; in this way, the flight dynamics essentially become an energy accounting problem, with all velocity derivative terms dropped. In other words, the "raw input" of the control algorithm can be thought of as velocity. This neglects direct simulation of *both* the short-period and phugoid flight dynamics modes, which are assumed to be much faster than the problem dynamics of interest (i.e., diurnal changes in energy injection from solar power).
  - Other time-dependent variables, such as the throttle setting, control surface deflections, and battery state of charge. Time-dependent cross-discipline coupling variables, like overall vehicle net power are also added.
  - The vehicle design itself, which includes:
    * Geometric shape variables, with a similar general parameterization methodology as the Firefly MDO setup of Section 4.2.2.
    * Propulsion sizing, including battery capacity, propeller diameters, motor voltage constants, and rated power of various electrical components
    * Detailed structural design variables, such as the wing rib count and payload truss sizing. Never-exceed speeds and ultimate load factors during gusts are also optimized, allowing the creation of a $V - N$ flight en-

velope with appropriate margins. As in Section 4.2.2, component-wise weights are also included as variables that are satisfied by implicit structural analysis models.

- **Constraints** (in total, roughly 4,000 constraints), drawn directly and indirectly from several interacting disciplinary analyses, with the major ones as follows:
  - **Aerodynamics**: a component-wise buildup of lift, drag, and moment using AeroSandbox's AeroBuildup model, detailed in Section 5.3. A nonlinear lifting line method and a vortex lattice method were also included as alternate options used to validate the buildup model. Optimization solutions typically differ by only a few percent between models, which implies good cross-validation. This is unsurprising, as a solar airplane represents a near-ideal case for typical theoretical assumptions for 3D aerodynamics (high aspect ratios, well-separated lifting surfaces).
  - **Stability & Control**: Static stability is verified through a workbook-style buildup of the location of the aircraft's neutral point, with static margin set as 20% of mean aerodynamic chord[b].
  - **Structures & Mass Properties**: Primary structure weights are physics-based where possible. For example, the wing spar weight and carbon fiber layup schedule are the result of an Euler-Bernoulli beam model with both bending and torsion[c]. Secondary structures are generally modeled using empirical statistical models. For example, many wing secondary structures are modeled

based on data from the *MIT Daedalus* project [77, 159, 160], with a 30% markup to mitigate operational challenges documented by the Daedalus team [161].

- **Propulsion**: A propeller model based on dynamic disc actuator theory with viscous correction factors is implemented [162]. Viscous correction factors are calibrated using the blade-element code QPROP with sectional airfoil data from XFoil [95, 163].
- **Trajectory / Dynamics**: A direct collocation method, which enforces the equations of motion at each time point. Path constraints, such as battery state of charge limits, altitude minimums, and airspeed minimums (i.e., station-keeping) are also added. In addition, periodicity constraints are added, which ensure energy closure over the 24-hour cycle.
- To support station-keeping constraints, a probabilistic model of both steady winds and gusts was developed using data from the ECMWF ERA5 reanalysis dataset [164]. The model is a function of latitude, longitude, and day of year, which allows global mapping of the feasible mission space.

[a]Example solar aircraft with at least one documented in-flight aeroelastic failure include NASA Helios, Google/Titan Solara 50, Facebook Aquila, and Airbus Zephyr.

[b]This static margin, which is a hair high for a high-L/D aircraft, is intended to mitigate risk. Fixing a too-aft CG during detailed design usually costs more weight than fixing a too-forward CG, due to differences in available moment arms.

[c]For speed, this is broken out as a separate optimization sub-problem, which is pre-computed for a wide range of spans and load distributions. A closed-form solution is then used as the "online" model during the broader MDO problem.

Further details on the Dawn MDO problem formulation and constituent models are available in Sharpe et al. [13]. As with the Firefly problem, the Dawn problem is implemented in AeroSandbox using a SAND architecture; Figure 4-13 shows the data flow within the MDO process in extended design structure matrix (XDSM) format [165].

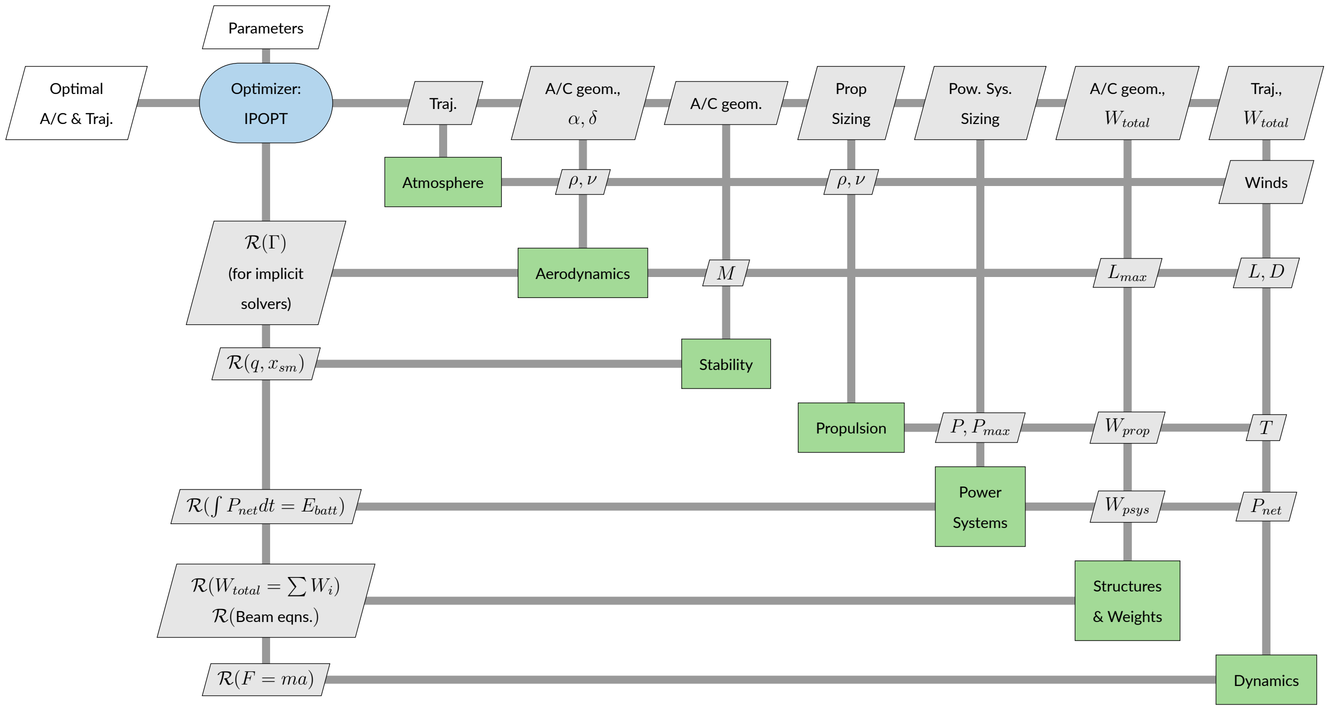


Figure 4-13: Extended Design Structure Matrix (XDSM) representation of the Dawn solar-electric HALE aircraft design optimization problem. Reproduced from Sharpe et al. [13].

4.3.4 Results and Framework Lessons

The result of the optimization process described in Section 4.3.3 is the point design that is summarized in Table 4.3 and Figure 4-12. The point design estimate produced using AeroSandbox was then further refined in collaboration with Electra.aero, leading to the successful construction and first flight of a full-scale demonstrator aircraft, as shown in Figure 4-14.

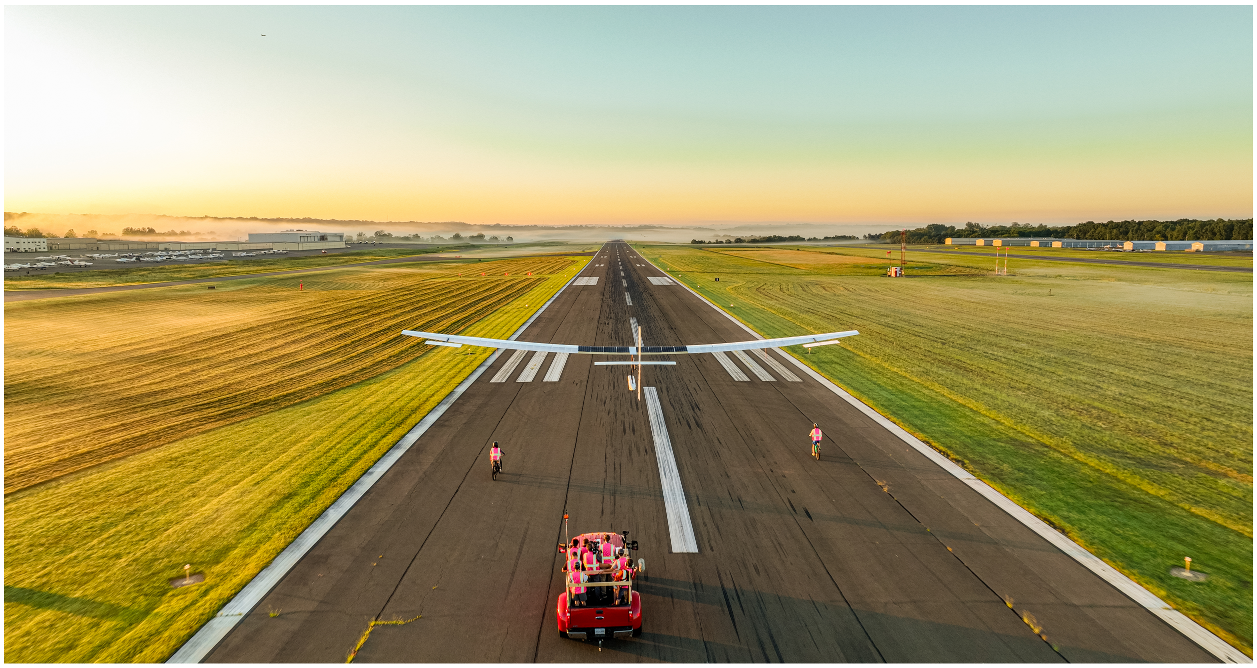


Figure 4-14: Photograph of the Dawn One solar-electric HALE aircraft lifting off during its first flight test, in September 2022. Reproduced from Electra.aero [14].

As with the Firefly design, however, many of the valuable lessons learned from the development of this tool were not only from the point design, but also from the trade studies one can quickly assess using this framework.

Rapid Performance Sweeps

As an example of this, a key early question during the design was "How does the required wingspan vary with latitude and seasonality?" To answer this, we designed 2,400 unique aircraft (a 60×40 grid in seasonality-latitude space), each the result of solving a unique instance of the MDO problem described in Section 4.3.3. With parallelization on a high-performance computing cluster (HPC), computing these results took several minutes, effectively allowing this question to be answered real-time in a meeting with major stakeholders.

The results of this study are shown in Figure 4-15, which plots the wingspan of the smallest-feasible aircraft as a function of season and latitude. Here, some interesting features are apparent. First, summertime missions (i.e., July in the northern hemisphere, January in the southern hemisphere) require the smallest aircraft, as expected. Interestingly however, mission becomes most feasible near extreme latitudes—this indicates that feasibility is less driven by solar intensity and more driven by the duration of the night[24]. Colloquially, solar panels are light and batteries are heavy.

Secondly, a steep change in the mission feasibility is observed near 40° N and 40° S latitude. This is due to changes in the tropopause height, which is a key factor that modifies the science requirements if the mission is generalized to other regions around the world. Due to global atmospheric convection cells, the tropopause altitude rises quickly near these latitudes.

Third, we can identify the impact of specific localized weather phenomena. For example, the southern polar vortex tends to raise stratospheric wind speeds during the month of November at latitudes near 60° S, which makes station-keeping extremely challenging[25].

[24]In the Arctic and Antarctic, the midnight sun effect allows quite small aircraft as almost no batteries are needed.
[25]Interestingly, an analogous phenomenon is not observed with the Arctic polar vortex in the northern hemisphere.

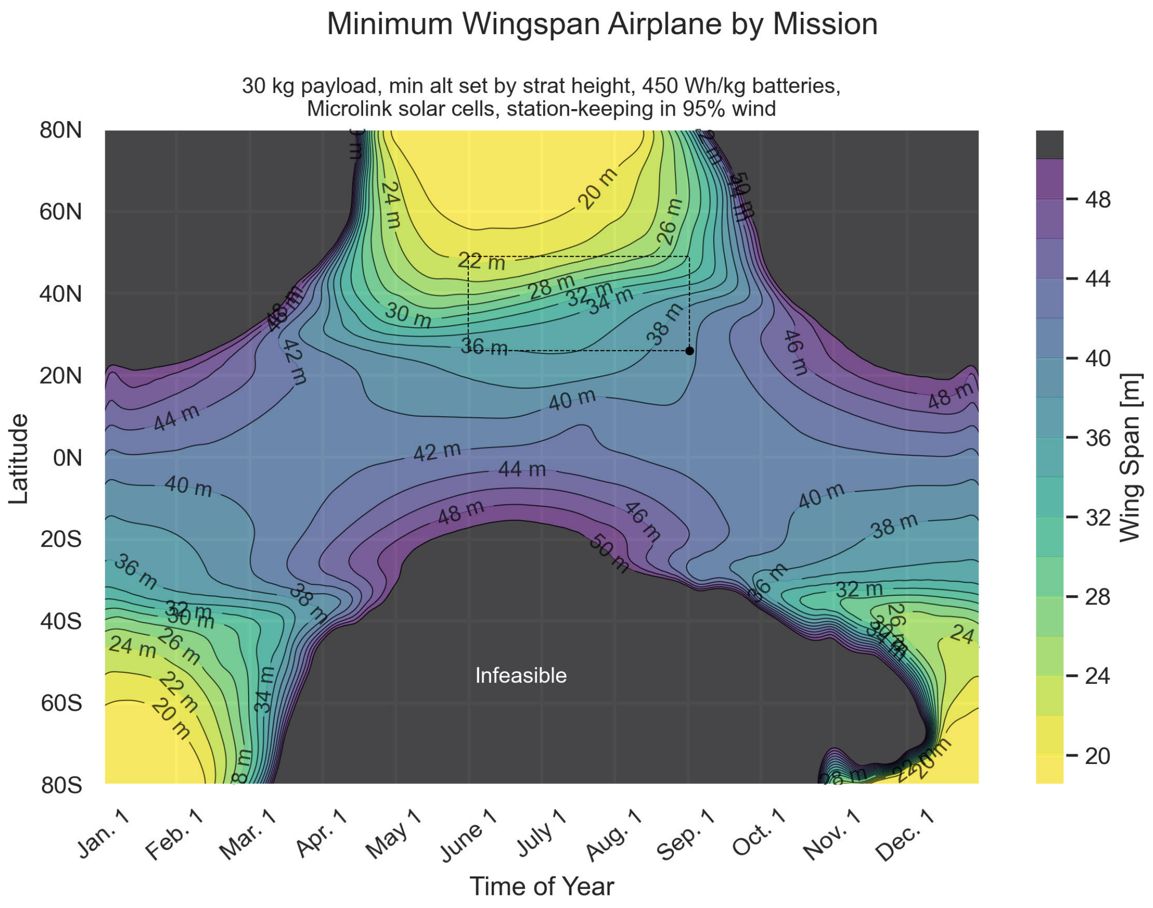


Figure 4-15: Feasibility of a solar aircraft throughout the seasonality-latitude mission space, as measured by the minimum required wingspan of an aircraft that achieves energy closure with the required payload. The baseline science mission, which involves summertime flight over the continental United States, is shown with a dashed black rectangle on the plot. Within this rectangle, the sizing case (i.e., most difficult mission) is shown with a black dot. Reproduced from Sharpe et al. [13].

Early mapping of this broader global mission space allowed the Dawn development team crucial insight into other potential missions of interest, such as Antarctic ice shelf monitoring, wildfire monitoring, methane monitoring, flood monitoring, and others. In a broader sense, this is a useful tool for business development, essentially allowing engineers to ask "Who else should we be pitching to?" at the earliest stages of the design process.

Rapid Problem Reformulation

One of the most useful features of a code transformations framework is that the separation between formulation and numerics allows for rapid problem reformulation. For example, suppose that instead of answering the baseline design problem (which assesses wingspan, given some fixed payload), we wish to instead assess what payload would be possible, given some fixed wingspan. This change can be implemented by changing just two lines of code, and after a few minutes of computation, results from this “flipped problem” are available for the user. This is shown in Figure 4-16, where a fixed span of 34 meters is assumed[26].

[26] Arbitrarily based on the wingspan of the MIT Daedalus aircraft, as a useful point of comparison for the design team.

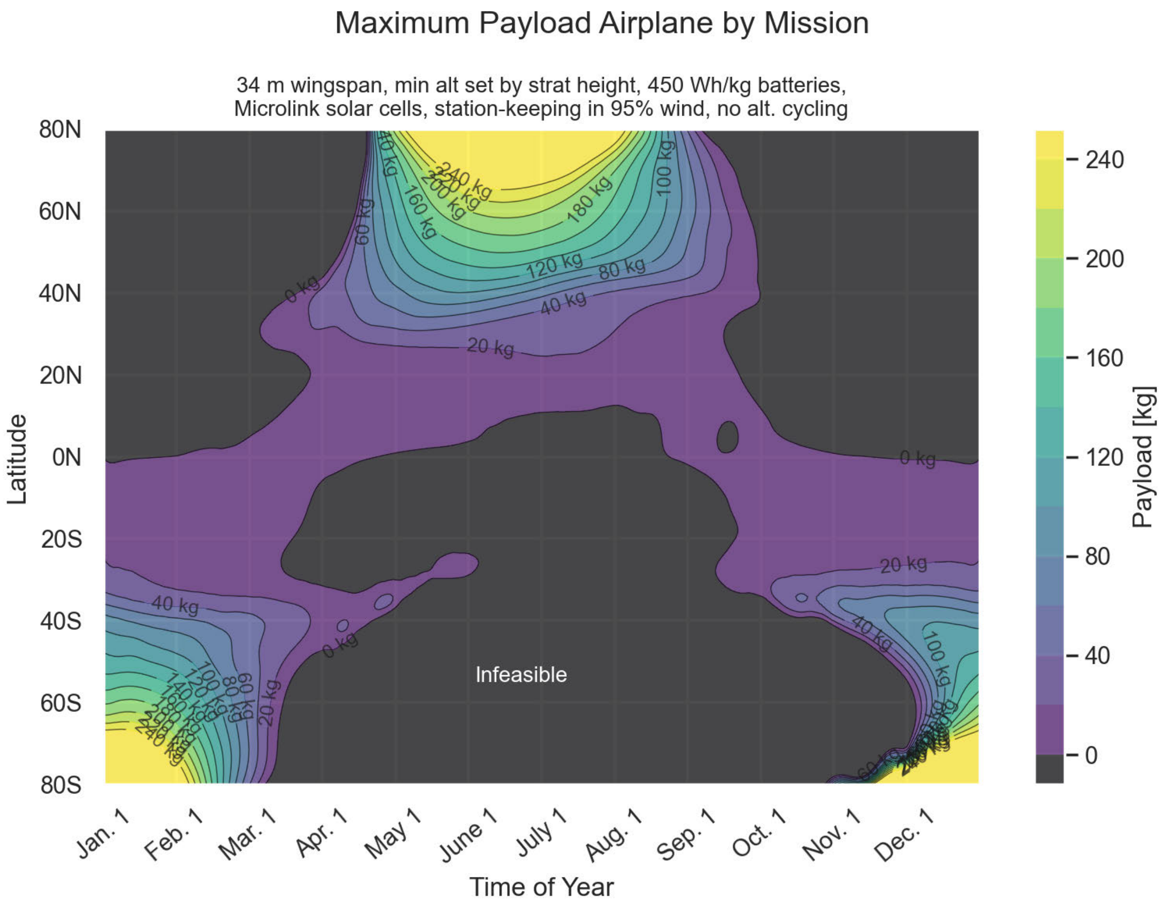


Figure 4-16: Feasibility of a solar aircraft throughout the seasonality-latitude mission space, as measured by the maximum payload that can be carried by a clean-sheet aircraft design with a fixed wingspan of 34 meters. Equivalent to a "flipped problem" of Figure 4-15. Reproduced from Sharpe et al. [13].

This rapid problem reformulation capability is crucial for the early-stage design process, where the design space is not yet well-understood. By allowing the user to quickly explore the design space in a variety of ways, the design tool can be used to rapidly iterate on the design problem, allowing the user to quickly gain insight into the design space and make informed decisions.

Sensitivity Analysis

Another useful framework-level feature that was developed through the Dawn project was the ability to extract first-order sensitivity information of various constraints and parameters. This information is taken from the dual variables of the optimization problem, which indicates how the performance would change if a single constraint or parameter was modified, while holding all others tight. (This is sometimes called a "total derivative" sensitivity.) Because IPOPT is a primal-dual optimizer, these sensitivities can be extracted essentially for free, as they are computed during the optimization process.

Table 4.4 gives a selected subset of these sensitivities for the Dawn design problem. In early design stages, these sensitivities are useful for understanding the design space – as in the Firefly example of Section 4.2.4, it is often not the designer's true intent to optimize wholly for one figure of merit at the expense of all others. In later design stages, these sensitivities can (and should) be used to allocate engineering resources. Example such questions could be: a) whether it more valuable to research battery specific energy improvements or drag reductions, or b) whether an alternative motor, which is more efficient but heavier, results in a net performance gain.

Table 4.4: First-order sensitivities for the point design corresponding to the the baseline mission. Reproduced from Sharpe et al. [13]

Figure of Merit	**Sensitivity of Wingspan**	**Sensitivity of TOGW**
Any added mass	0.22 m/kg	4.17 kg/kg*
Any added power draw	0.010 m/W	0.185 kg/W
Any added drag	0.479 m/N	7.50 kg/N
Battery spec. energy	-0.090 m/(Wh/kg)	-0.960 kg/(Wh/kg)
Battery packing fraction	-0.461 m/(%)	-4.847 kg/(%)
Solar cell efficiency	-0.464 m/(%)	-0.865 kg/(%)
Solar cell area density	12.4 m/(kg/m^2)	149 kg/(kg/m^2)
Solar cell usable wing fraction	-10.6 m/(%)	80.9 kg/(%)
Propeller efficiency	-0.613 m/(%)	-4.72 kg/(%)
Motor efficiency	-0.586 m/(%)	-4.75 kg/(%)
Minimum cruise altitude	5.10 m/km	46.2 kg/km

* Note that this is not equal to the reciprocal of the fixed weight fraction, since this sensitivity assumes that the mass is added uniformly across the aircraft (i.e., spanloaded), while the fixed mass is mostly a point-mass payload.

4.3.5 Computational Reproducibility

A publicly-accessible repository of the code used to generate the Dawn aircraft conceptual design is available via the *Dawn Design Tool* at `https://github.com/peterdsharpe/DawnDesignTool`. We note that this public code has been forked and privately refined in collaboration with the Electra.aero engineering team, so results using this public repository should not be considered exactly representative of any up-to-date design decisions or performance estimates of the as-built aircraft.

4.4 Liquid-Hydrogen-Fueled Long-Haul Transport Aircraft

A third example aircraft design case study conducted using AeroSandbox was a conceptual design study of a liquid-hydrogen-fueled long-haul transport aircraft.

4.4.1 Problem Overview and Background

At the time of writing, aviation is generally estimated to be responsible for roughly 3% of anthropogenic climate impacts, with slight variations in this figure depending on method of accounting[27]. Of this fraction, long-haul aviation (flights with ranges longer than 4,800 km)[28] contributed roughly 35% of the total aviation fuel burn [166]. This is of particular concern, because while short-haul aviation has more readily-available decarbonization pathways (e.g., electrification), long-haul aviation has very few options to decarbonize due to specific energy requirements.

In practice, the two main options for decarbonizing long-haul aviation are synthetic aviation fuels (SAF) and hydrogen-based solutions. While SAF is attractive in the short-term as a potential drop-in solution, the scalability of a SAF-based aviation fuel ecosystem remains an area of concern[29] [167, 168]. Hydrogen, on the other hand, is a more speculative solution, but if scaling SAF proves to be technically, politically, or economically infeasible, hydrogen may be the only remaining option for long-haul decar-

[27] Aviation is roughly 2.5% of global carbon emissions, but the high-altitude nature of emissions cause disproportionately high impact. Also, non-carbon climate impacts (e.g., contrails, NOx, etc.) may have strong impacts on radiative forcing that are not captured in carbon-focused emissions measures.

[28] These mostly include intercontinental flights on wide-body aircraft.

[29] Colloquially, this is often referred to as the "food vs. fuel" debate, as the most economically-viable SAF formulations tend to use food feedstock. This drives up food prices and competes for limited arable land. Estimates of the land area required to sustain a global SAF-based aviation economy are similar in scale to the current arable land of the United States.

bonization.

Hydrogen is best thought of as a "carrier" fuel, similar to a battery, although with much higher specific energy. (For comparison, the specific energy of hydrogen is roughly 120 MJ/kg, while kerosene-based fuels give roughly 43 MJ/kg and lithium batteries give roughly 1 MJ/kg.) Because of this, the emissions associated with hydrogen are, of course, highly dependent on the method of hydrogen production. Currently, the most common method of hydrogen production is steam methane reforming (SMR), which is a process that produces hydrogen from natural gas. This process is not carbon-neutral, but it provides an economical on-ramp for a hydrogen economy while electrolysis technologies and grid decarbonization advance[30] [169]. In the long term, "green hydrogen" produced by electrolysis powered by low-emissions[31] sources (e.g., wind, solar, hydropower, geothermal, nuclear) could dramatically reduce the carbon impact of long-haul aviation.

Hydrogen may be stored in either a high-pressure gaseous (GH_2) or a cryogenic liquid (LH_2) form, each of which come with their own unique challenges. For long-haul aviation, LH_2 is universally preferred due to its much higher effective specific energy (i.e., specific energy with the tank mass included). For example, with LH_2, the tank gravimetric efficiency η_{tank} (essentially, the mass fraction of a filled tank that is usable hydrogen) is usually 50–75% [170]; for GH_2, this is typically 8–12%. The higher volumetric density of LH_2 relative to GH_2 also allows for more compact tank designs, which reduces the drag impact at the aircraft level.

In addition to the grid decarbonization and hydrogen production challenges, hydrogen presents a number of unique operational difficulties at the aircraft-level. Interestingly, the common concern of crash safety is not actually a major one (relative to kerosene-based fuels), as a) in the event of a post-

[30] In the medium term, hydrogen can result in reduced aviation emissions compared to a present-day baseline even without full grid decarbonization. This is especially true if carbon capture utilization and storage (CCUS) is incorporated into the SMR process.

[31] Considering lifecycle emissions

crash fire, hydrogen is a buoyant gas at STP, which directs heat upwards and prevents on-ground fuel pooling; and b) hydrogen burns with minimal soot[32]. In-flight fire is also a risk that can be mitigated to acceptable levels, through a combination of positive-pressure tanks, gas sniffers, and boiloff vents [170]. However, safe hydrogen refueling is challenging and several important open research questions remain [168]. For example, fuel lines must be purged for safety every time the aircraft is refueled, and a cryogenic fuel presents significant limitations on how this can be done[33].

Aircraft performance is also a significant concern when hydrogen is introduced. For example, both LH_2 and GH_2 tanks benefit from a low-surface-area-to-volume ratio, which makes traditional storage within wings impractical. Instead, hydrogen is typically stored in cylindrical or conformal fuselage tanks. Relative to traditional wing storage, this a) loses the benefit of spanloading, leading to increased structural weight in the wing and fuselage, b) can create a significant center-of-gravity shift during fuel burn, depending on configuration, and c) usually yields a (small) aerodynamic penalty due to the increased volume of the fuselage. In addition, tank weight is much higher[34], fuel lines are heavier, and fuel pumping becomes much more complex[35]. Brewer gives one of the most thorough overviews of these opportunities and challenges [170].

On the other hand, the much higher specific energy of hydrogen compared to kerosene results in far smaller required fuel mass fractions[36]. Because of this, it is not obvious whether the net aircraft performance impact of hydrogen is positive or negative relative to existing kerosene-based aircraft, and

[32]In post-crash fires on kerosene-fueled aircraft, smoke inhalation often causes the majority of injuries. Hydrogen combustion produces minimal smoke, mitigating this risk pathway to some extent.

[33]For example, common purge gases like nitrogen and argon will freeze at LH_2 temperatures; other gases, like helium, must be used. The economic feasibility of this at-scale remains an open research question.

[34]In the case of GH_2, this is due to pressurization stress, and in the case of LH_2, this is due to insulation and boil-off considerations.

[35]For example, rotating components in a pump usually cannot be lubricated, because these oils freeze well above the boiling point of LH_2. Hydrogen embrittlement also limits pump life.

[36]For example, even after accounting for increased tank weight, LH_2 offers a *realizable* specific energy that is roughly double that of kerosene-based fuels.

studies find results that vary strongly based on assumptions [168, 169, 171].

In short, the motivation for exploring hydrogen as an aviation fuel is not that it is particularly attractive or easy to work with; instead, it is because a genuine possibility exists that it may be the *only* scalable solution for long-haul aviation decarbonization. Hydrogen converts a problem that may otherwise be impossible to solve (decarbonizing long-haul aviation) into a problem that, while still very challenging, is at least technically solvable (decarbonizing the electricity grid).

4.4.2 Aircraft Design Problem Formulation

In this study, we formulated an aircraft design problem for a hydrogen-fueled long-haul transport aircraft, with the goal of learning more about aircraft performance deltas from kerosene designs. The key design requirements were modeled after specifications of a Boeing 777-300ER, which is a representative modern kerosene-fueled long-haul aircraft. These requirements are given in Table 4.5. In addition, this study assumes that a clean-sheet airplane and engines are used, and that such engines are designed around a 2023 technology level (roughly comparable to a GE9X engine).

Table 4.5: Key design requirements for the LH_2-fueled long-haul transport aircraft design.

Specification	**Value**	**Motivation**
# of Passengers	400	Long-haul demand
Ultimate Range (fully loaded)	> 7,500 nmi	Long-haul demand
Cruise Mach	> 0.75	2x / day turnaround, market acceptance
Wing Span	< 64.8 m (213 ft)	FAA AC 150/5300-13 ICAO Group V: 52 – 65 m
Engine-out Climb Gradient	> 2.4%	Part 25.121
Fuel Source	Hydrogen	

Three key configuration-level decisions were made a priori based on existing literature:

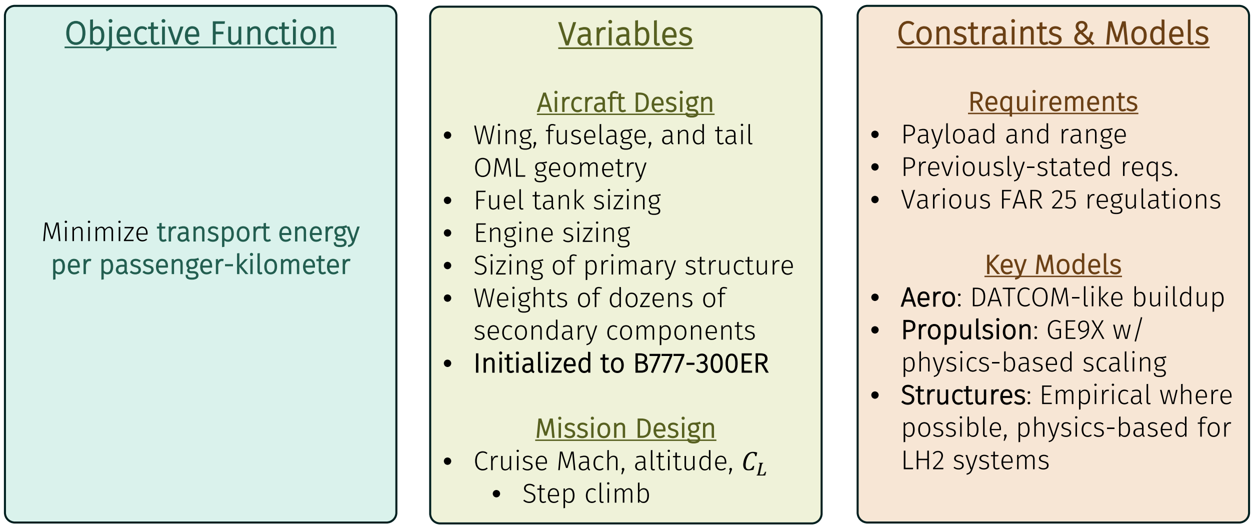
1. Choice of LH_2 fuel, rather than GH_2, due to its significantly higher realizable specific energy.
2. Choice to place fuel in two conformal forward and aft tanks in the fuselage, which results in a smaller center-of-gravity shift during fuel burn compared to a single aft tank[37]. Other tank configurations were considered and not selected for boil-off reasons.
3. Choice to use direct hydrogen combustion, rather than fuel cells, due to much higher specific power.

More details on the motivations for these decisions, and possible alternative ones, are given in the author's contributions to prior work by Gaubatz et al. [168].

[37] Having one aft tank requires a larger horizontal stabilizer in order to trim during the center of gravity shift, costing drag. Also, multiple tanks are needed anyway for FAR 25 certification under current guidance.

From this set of requirements a vehicle design optimization problem was formulated, using the problem setup shown in Figure 4-17. Unlike the Firefly and Dawn problems, this problem was formulated as a single-point design problem, rather than a trajectory optimization problem. This is because the vehicle performance is dominated by cruise performance, and during this phase it is assumed to operate at a constant design lift coefficient and Mach number. (This implies that a step climb is performed throughout cruise as fuel is burned and mass decreases, which is common in practice and described further by Drela in material related to the TASOPT code [16].) Using Breguet-range-like relations, the aircraft's total fuel burn can be computed from this representative optimized cruise operating point.

In the formulation of Figure 4-17, the objective function the *transport energy efficiency*, which is defined as the chemical potential energy of the fuel burned divided by the economically-useful work (measured in passenger-miles) performed by the aircraft. This metric, which is common in green transportation studies, is often used to compare economic viability across both renewable and nonrenewable solutions, which are otherwise difficult to compare directly [167].


Objective Function

Minimize transport energy per passenger-kilometer

Variables

Aircraft Design
- Wing, fuselage, and tail OML geometry
- Fuel tank sizing
- Engine sizing
- Sizing of primary structure
- Weights of dozens of secondary components
- **Initialized to B777-300ER**

Mission Design
- Cruise Mach, altitude, C_L
  - Step climb

Constraints & Models

Requirements
- Payload and range
- Previously-stated reqs.
- Various FAR 25 regulations

Key Models
- **Aero**: DATCOM-like buildup
- **Propulsion**: GE9X w/ physics-based scaling
- **Structures**: Empirical where possible, physics-based for LH2 systems

Figure 4-17: High-level design optimization problem formulation for a liquid-hydrogen-fueled long-haul transport aircraft.

4.4.3 Results

This problem is formulated and solved using AeroSandbox, yielding a point design for a LH_2-fueled transport aircraft. Using the AeroSandbox aircraft geometry stack, a three-view design can be generated and presented to the user, which has been further illustrated to yield Figure 4-18. Overall, the resulting configuration is similar to a Boeing 777-300ER, with a few notable changes: a clean-sheet airframe and engines, much higher realizable fuel specific energy, and a wider cabin that accommodates a 4-5-4 seating arrangement. In Figure 4-18, this substantial increase in fuselage width somewhat visually obscures the large volume occupied by the conformal hydrogen tanks.

Another immediate change that is apparent in Figure 4-18 is the placement of the flight deck, which has no forward visibility. This solves pressurization and access challenges, but requires a camera-based forward visibility system. Existing certification pathways of enhanced flight vision systems (EFVS) and synthetic vision systems (SVS) may be used to certify this IFR-only flight deck configuration, but this is an open research question.

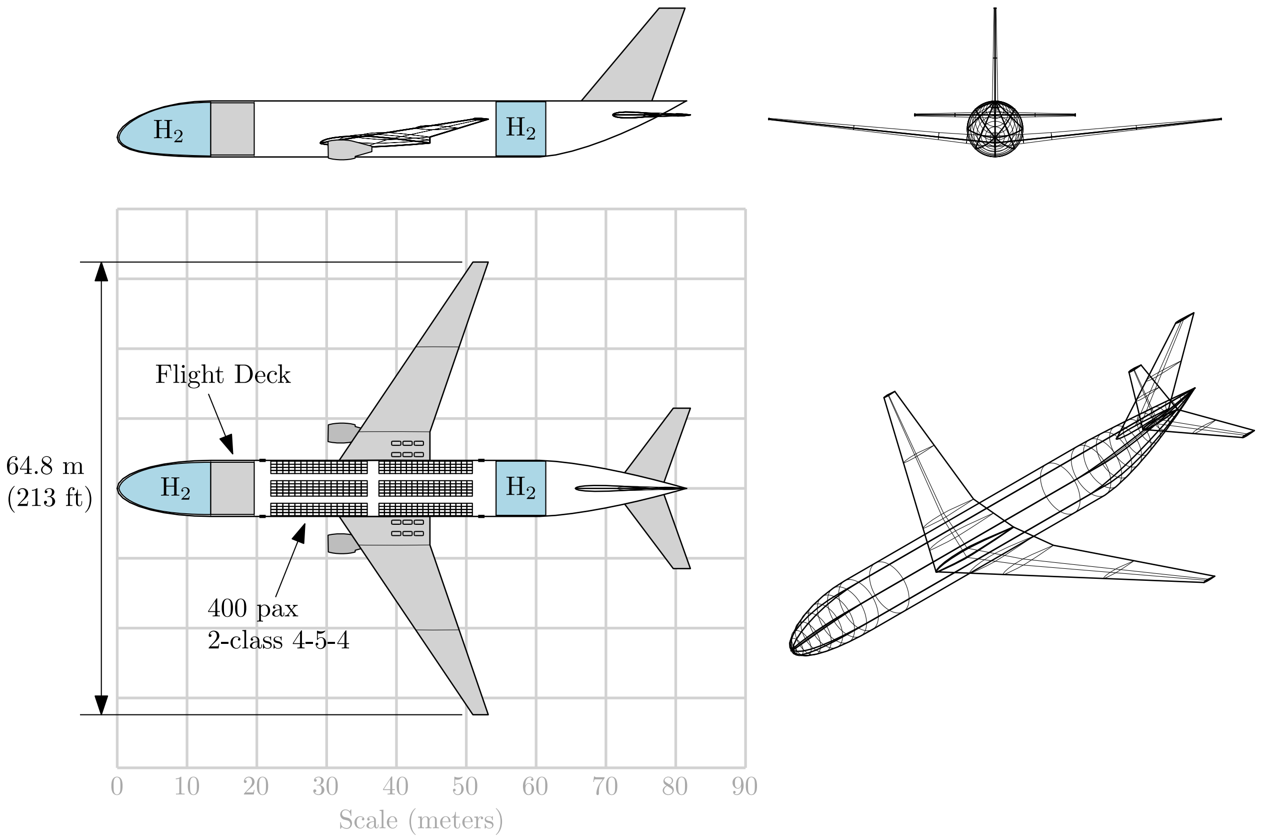


Figure 4-18: Three-view drawing of the LH_2-fueled long-haul transport aircraft design.

The results of the AeroSandbox optimization study can also be quantitatively shown in Table 4.6. In this table, the key performance metrics of the LH_2-fueled aircraft are compared to those of an equivalent kerosene-fueled aircraft, which was optimized using an identical methodology and assumptions. This is useful to isolate the impact of fuel choice on various performance parameters. In addition, the as-built Boeing 777-300ER is shown in the table for comparison, which allows the overall accuracy of the optimization model to be assessed. The higher gross weight of the B777 aircraft is partially explained by its higher ultimate range than the optimized aircraft in this study.

Table 4.6: Key results of the LH_2-fueled long-haul transport aircraft design. Comparisons are made to a kerosene-fueled aircraft optimized using an identical methodology and assumptions, as well as to an as-built Boeing 777-300ER.

	Optimized LH_2 **Airplane**	**Optimized Kerosene Airplane**	**As-Built B777-300ER [172]**
# of passengers	400	400	396
Ultimate range	7,500 nmi	7,500 nmi	8,200 nmi (estimated)
Cruise Mach	0.82	0.83	0.84
Cruise Altitude	36,800 ft	36,900 ft	43,100 ft
Gross Weight	267,800 kg	300,800 kg	351,500 kg
Empty Weight	184,700 kg	150,200 kg	167,800 kg
Fuel Capacity	44,100 kg	111,600 kg	145,500 kg
Wing Span	64.8 m	64.8 m	64.8 m
Length	81.5 m	73.0 m	73.9 m
Lift/Drag	15.5	16.6	-
Fuel Burn	7.94 g/pax-km	20.1 g/pax-km	19.4 to 26.1 g/pax-km
Transport Energy	**0.95 MJ/pax-km**	**0.86 MJ/pax-km**	**0.84 to 1.13 MJ/pax-km**

Notably, Table 4.6 indicates that the transport energy efficiency of the LH_2-fueled aircraft is slightly worse than that of the kerosene-fueled aircraft. Most of this can be attributed to the higher empty weight of the LH_2 aircraft, which cannot take advantage of the spanloading ability of kerosene-fueled aircraft. This leads to higher structural weight in the wing and fuselage.

For a more detailed breakdown of the mass changes as a result of fuel choice, we can inspect the mass budgets of the LH_2 and kerosene aircraft designs directly. Recent versions of AeroSandbox support mass properties data structures that allow visualizations of these budgets to be generated automatically. Figures 4-19 and 4-20 show the mass budgets of the LH_2 and kerosene aircraft designs, respectively. These figures show that the LH_2 aircraft has a much lower fuel mass fraction than the kerosene aircraft; however, it is offset by increased weight in the wing, fuselage, and fuel tanks.

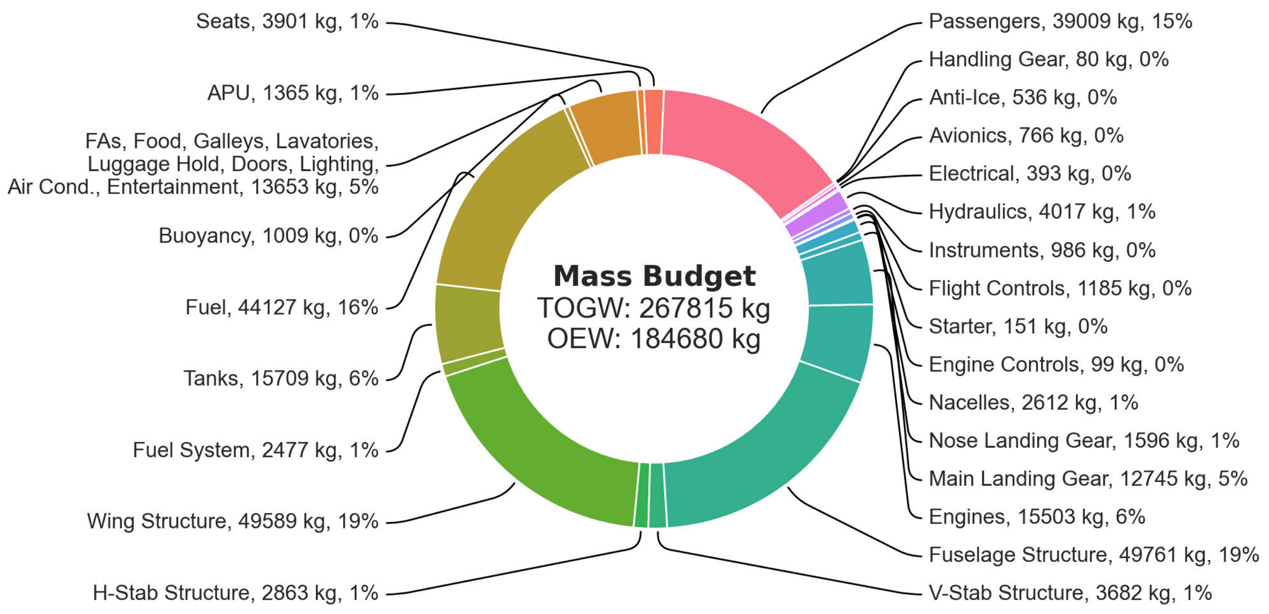


Figure 4-19: Mass budget for the LH_2-fueled aircraft design. Takeoff gross weight (TOGW) and operating empty weight (OEW) are given in the center of the diagram.

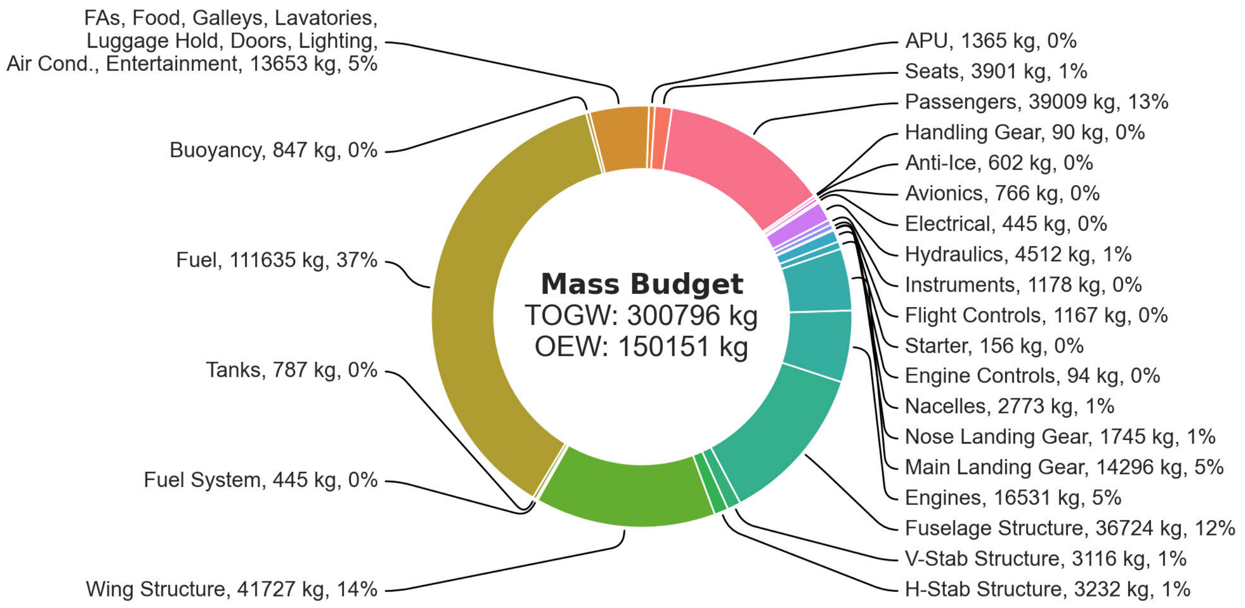


Figure 4-20: Mass budget for an equivalent **kerosene**-fueled aircraft design, which allows comparison against the LH_2 design in Figure 4-19. Takeoff gross weight (TOGW) and operating empty weight (OEW) are given in the center of the diagram.

4.4.4 Performance Sweeps

As with the Firefly example, exploring the impact of a given technology assumption on the design is as simple as changing two lines of code within a procedural-style code transformations framework. Here, the impact of the tank gravimetric efficiency η_{tank} on the transport energy efficiency of the LH_2-fueled aircraft design was explored. The results of this study are shown in Figure 4-21, which shows that the relative performance of the LH_2-fueled aircraft design is highly sensitive to the tank gravimetric efficiency. Given that current tanks have η_{tank} values between 50–75% [170], this implies that LH_2 transport aircraft are roughly competitive with kerosene-fueled aircraft in terms of transport energy efficiency. This chart also demonstrates why GH_2-based aircraft (with \$$\eta_{\text{tank}} \approx 10\%$) are wholly infeasible for long-haul aircraft. Finally, this chart gives a clear explanation as to why estimates for the relative transport energy of hydrogen aircraft vary so widely in the aircraft design literature – this metric is highly sensitive to the assumed tank technology.

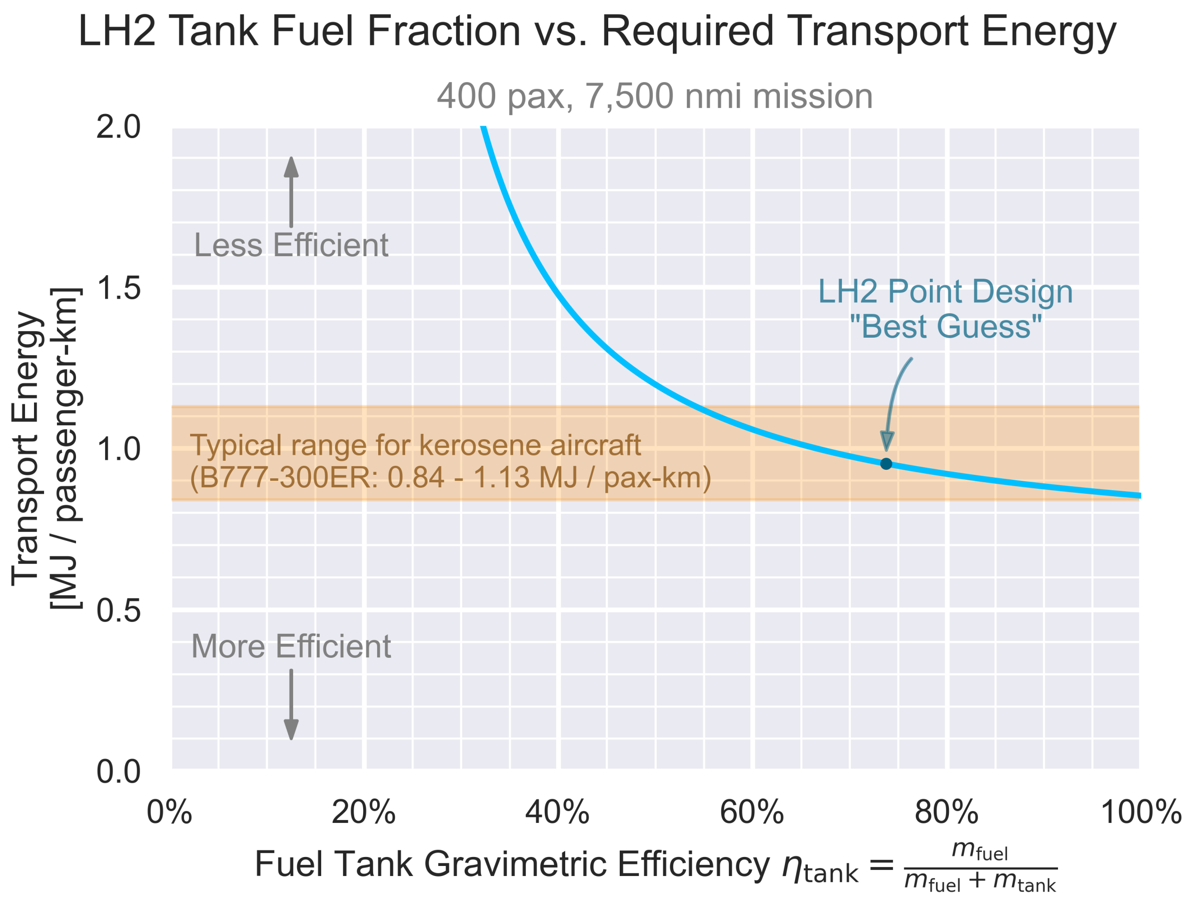


Figure 4-21: Comparison of transport energy efficiency of LH_2- and kerosene-fueled aircraft, designed around the same mission. Relative performance depends strongly on the LH_2 tank gravimetric efficiency, which forms a critical performance metric.

4.4.5 Fleet Design and Market Considerations

Another example of how a rapid design tool can be used is to explore business-level impacts of a technology choice. To demonstrate this, here we show how one might think about covering a market of many long-haul flights with different ranges.

First, consider the payload-range diagram of the LH_2-fueled aircraft design, shown in Figure 4-22. This diagram shows the maximum payload that can be carried by the aircraft as a function of range, a key metric of usefulness for an airline. For example, in this simplified model, the aircraft can carry 400 passengers at its design range of 7,500 nmi. For flights slightly longer than this range, the aircraft must reduce the fuel burn by carrying fewer passengers. For flights shorter than this range, the aircraft is seat-limited to its 400 passenger capacity; however, what is not shown here is that less fuel can be carried, which reduces the aircraft's fuel burn and improves its transport energy efficiency.

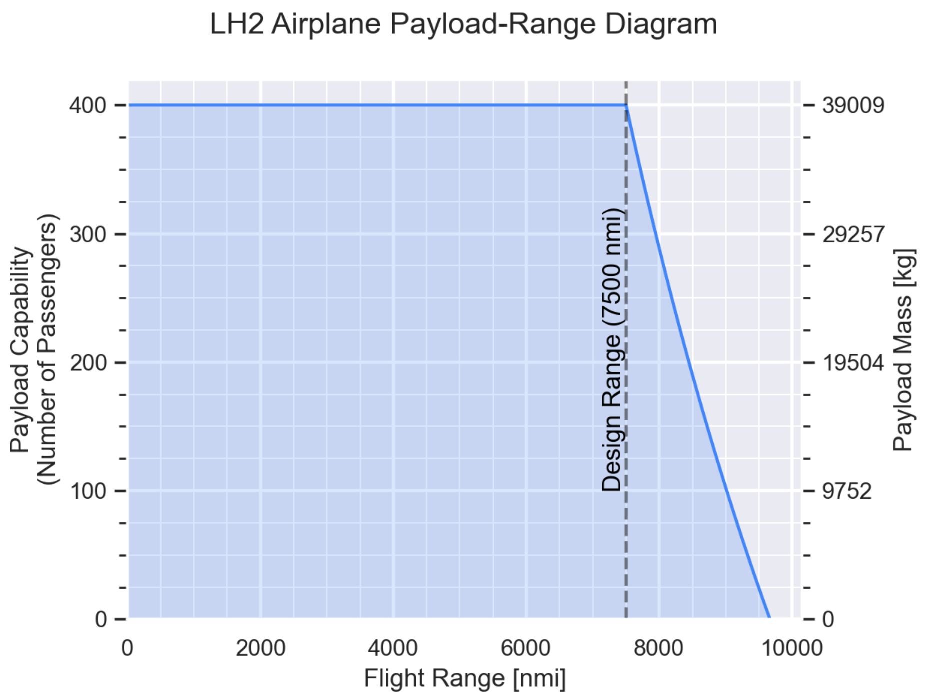
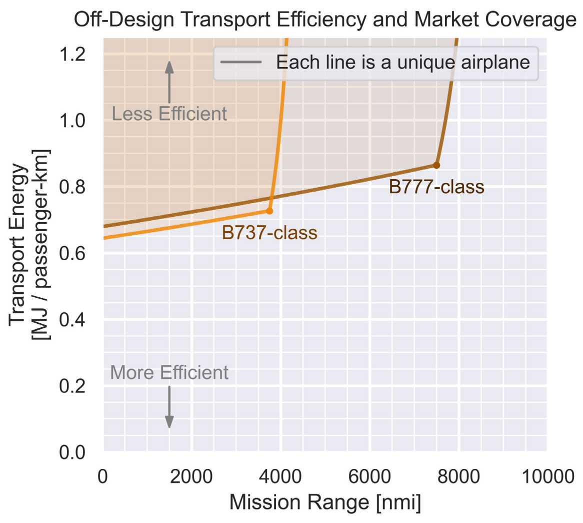
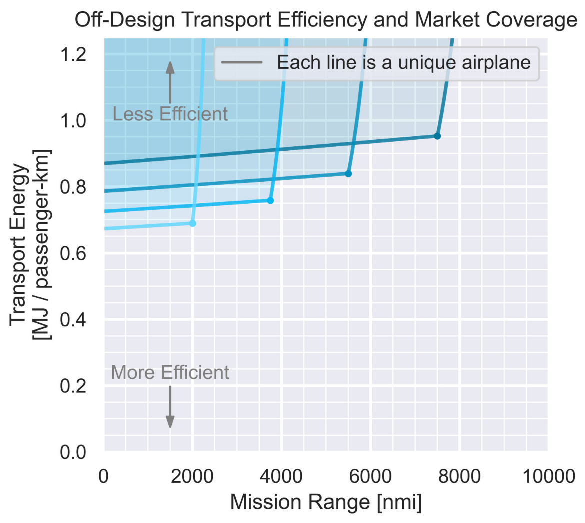


Figure 4-22: Payload-range diagram for the LH_2-fueled aircraft design.

However, this improvement in transport energy efficiency for shorter-than-maximum-range flights is not the same for LH_2- and kerosene-fueled aircraft. This is because the fuel mass fraction of the LH_2 aircraft is much lower than that of the kerosene aircraft, so a hydrogen aircraft achieves less benefit from flying shorter-range flights. This is illustrated in Figure 4-23, where each line shows the transport energy of a particular optimized aircraft design as a function of range. (For each aircraft, the design range is indicated by a dot on the line.) As an operator, the effective transport efficiency *of the combined fleet* is the lower-bound envelope of these curves. Notably, for a kerosene aircraft, there is a relatively small benefit to flying the "right size airplane for a given mission", so the market can be covered with two aircraft without much loss in efficiency. For a hydrogen aircraft, this is not the case, and the market must be segmented more finely to achieve the same efficiency.



(a) **Kerosene**-fueled aircraft fleet



(b) LH_2-fueled aircraft fleet

Figure 4-23: These figures show how multiple individual aircraft (each shown by a line) can be combined to cover a market of long-haul flight segments. In general, aircraft will be more fuel-efficient when flown closer to their design range (indicated by a dot on each line). However, hydrogen-fueled aircraft get less benefit from flying shorter-range flights, because their fuel mass fraction is lower. This drives a need for more aviation market segmentation if a hydrogen transportation system is implemented.

This fleet-design study is just one example of how a rapid design tool can be used to discover the subtle downstream business-level impacts of a technology choice, which may not be otherwise anticipated. This tool can also be used to explore the upstream impacts of a technology; for example, a study by Gaubatz et al. [168] uses this AeroSandbox-based aircraft design tool as one subset of a much broader design study to estimate the grid-level electricity demands of a hydrogen-based long-haul air transportation system.

4.4.6 Computational Reproducibility

A publicly-accessible repository of the hydrogen aircraft design tool is available at `https://github.com/peterdsharpe/transport-aircraft`.

4.5 Other Aircraft Design Case Studies

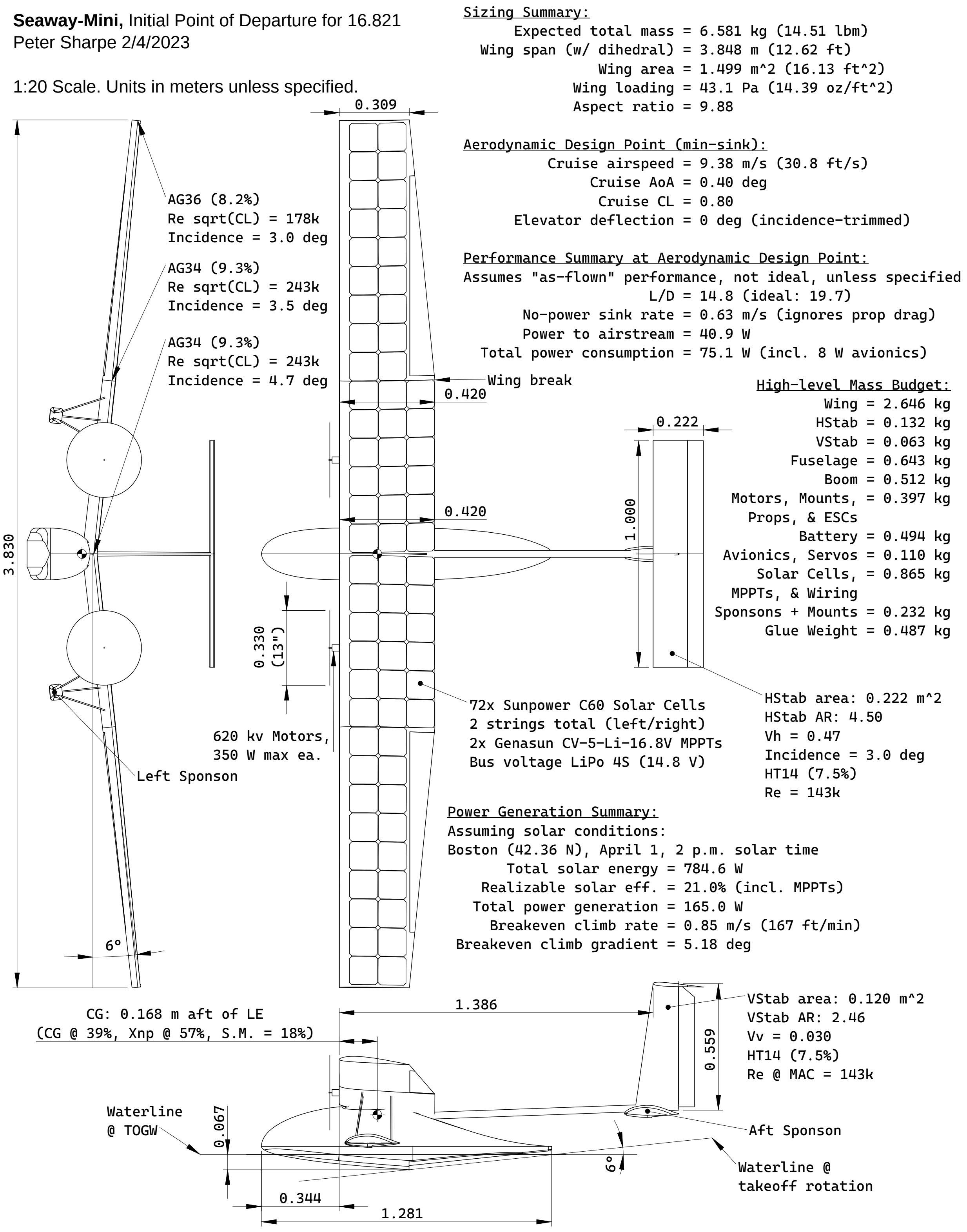
In this final section, we leave brief descriptions and links to other aircraft design case studies that have been conducted using AeroSandbox. For brevity, these case studies are not as thoroughly described in this document as previous examples; however, they serve to provide more possible starting points for the reader interested in implementing their own aircraft design studies based on the work here.

4.5.1 Solar Seaplane

Another aircraft design project that influenced the development of AeroSandbox was the design of a small-scale solar-electric seaplane, which was performed to support teaching work for MIT 16.821: Flight Vehicle Development. This aircraft design aimed to miniaturize a design for a full-scale recreational ultralight aircraft created by students in MIT 16.82: Flight Vehicle Engineering the prior semester. The goal was to roughly recreate the major elements of this previous vehicle, but at a scale that would allow the students to build and test-fly the aircraft within the scope of one semester. As the intended focus of the class was primarily on building techniques rather than design, we elected to give the students a "point of departure" design to jump-start the project.

The design of this aircraft was formulated as an optimization problem in AeroSandbox, which yielded the design illustrated in Figure 4-24. This design was then used as a starting point for the students to build their aircraft, which was successfully flown in a series of test flights at the end of the semester.

AeroSandbox code used to develop this aircraft design is publicly available at `https://github.com/peterdsharpe/solar-seaplane-preliminary-sizing`.



Seaway-Mini, Initial Point of Departure for 16.821
Peter Sharpe 2/4/2023

1:20 Scale. Units in meters unless specified.

```
Sizing Summary:
      Expected total mass = 6.581 kg (14.51 lbm)
   Wing span (w/ dihedral) = 3.848 m (12.62 ft)
                Wing area = 1.499 m^2 (16.13 ft^2)
             Wing loading = 43.1 Pa (14.39 oz/ft^2)
             Aspect ratio = 9.88

Aerodynamic Design Point (min-sink):
          Cruise airspeed = 9.38 m/s (30.8 ft/s)
               Cruise AoA = 0.40 deg
                Cruise CL = 0.80
      Elevator deflection = 0 deg (incidence-trimmed)

Performance Summary at Aerodynamic Design Point:
Assumes "as-flown" performance, not ideal, unless specified
                      L/D = 14.8 (ideal: 19.7)
       No-power sink rate = 0.63 m/s (ignores prop drag)
       Power to airstream = 40.9 W
   Total power consumption = 75.1 W (incl. 8 W avionics)

High-level Mass Budget:
                  Wing = 2.646 kg
                 HStab = 0.132 kg
                 VStab = 0.063 kg
              Fuselage = 0.643 kg
                  Boom = 0.512 kg
       Motors, Mounts, = 0.397 kg
         Props, & ESCs
               Battery = 0.494 kg
      Avionics, Servos = 0.110 kg
          Solar Cells, = 0.865 kg
       MPPTs, & Wiring
    Sponsons + Mounts = 0.232 kg
          Glue Weight = 0.487 kg

Power Generation Summary:
Assuming solar conditions:
Boston (42.36 N), April 1, 2 p.m. solar time
       Total solar energy = 784.6 W
    Realizable solar eff. = 21.0% (incl. MPPTs)
   Total power generation = 165.0 W
     Breakeven climb rate = 0.85 m/s (167 ft/min)
 Breakeven climb gradient = 5.18 deg
```

Figure 4-24: Design drawing of the Solar Surfer solar-electric seaplane.

4.5.2 Feather: Ultra-Lightweight Remote Control Motor-Glider

A final aircraft design project that can serve as a useful code example is that of *Feather*, a project to build a minimum-sink-rate 1-meter-wingspan remote-controlled glider. The goal is to be able to exploit even the very weak low-altitude thermals that occur during early morning hours to stay aloft.

Traditionally, remote-control gliders aimed at thermal performance and within this wingspan range have been discus-launch gliders (DLGs). These aircraft are launched by holding the glider by a wingtip, spinning one's entire body in a circle (about the aircraft's yaw axis) and throwing. This achieves very high launch altitudes (up to 70 meters, in extreme cases), which has several benefits:

- Higher launch altitude yields longer flight times (assuming a constant sink rate), and hence more altitude to find thermals.
- At higher altitudes, thermals are a) wider, b) stronger[38], and c) tend to persist longer, which allows for consistent circling once thermals are found.

All three combined factors make it easier to consistently stay aloft with thermals once some initial altitude is already achieved. DLGs successfully solve this launch altitude problem, but the violent forces during their launch (often exceeding 30 Gs of acceleration) cause some unfavorable side effects:

- The high launch forces require a heavier wing and tailboom (due to yaw moment during derotation immediately after the throw), generally resulting in a higher still-air sink rate.
- The design space for tail geometries is constrained by the need to minimize fuselage boom torsion. For example, vertical stabilizers must be roughly top-bottom symmetric about the fuselage boom axis, which means that $C_{l\beta}$ stability that would ordinarily be provided by a top-mounted vertical stabilizer must be provided by a larger wing dihedral angle.

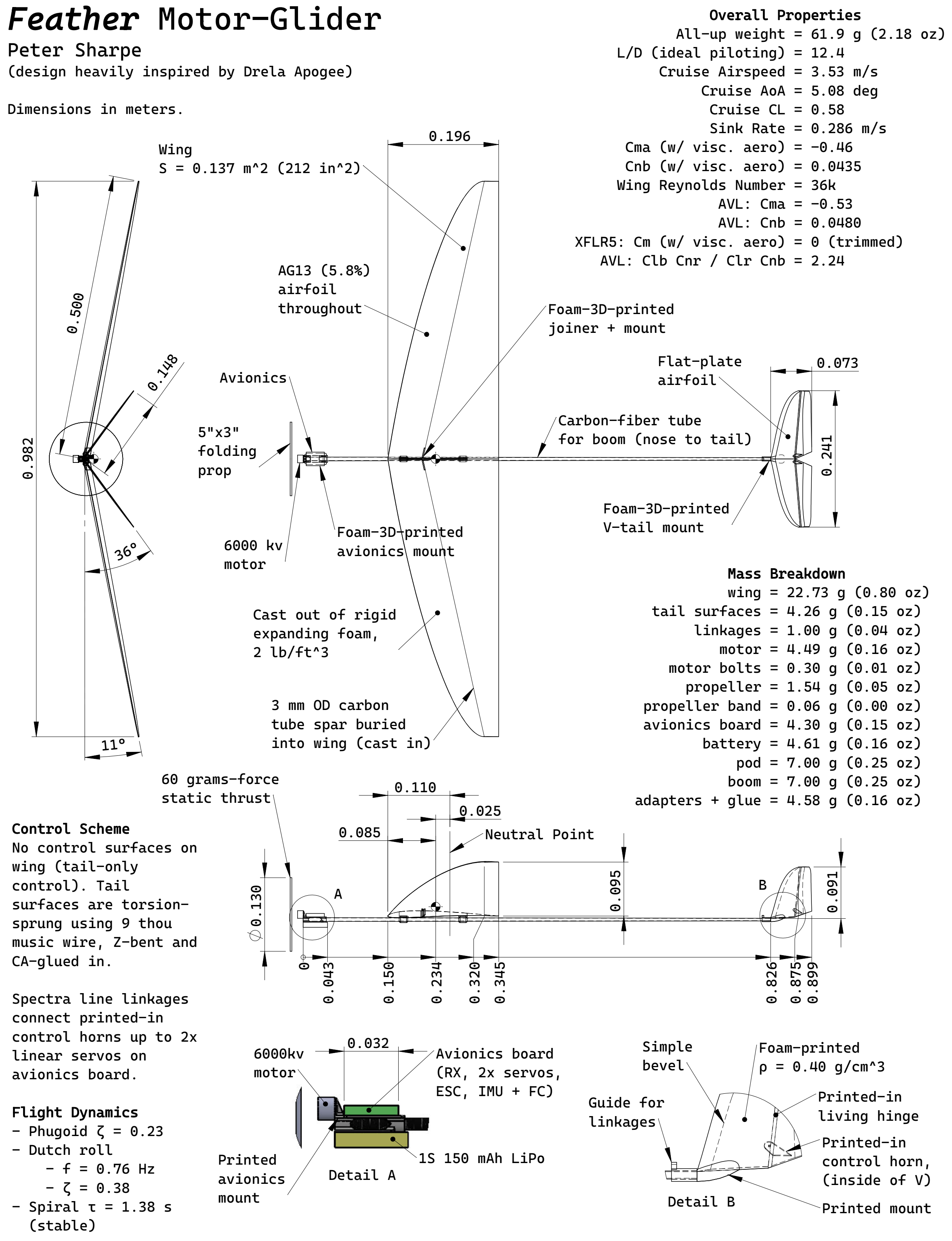
[38]Particularly for low-altitude thermals, within the atmospheric boundary layer.

However, in recent years, electronics have become so miniaturized that it is now possible to solve this launch altitude problem with a tiny electric propulsion unit, rather than by discus launch. The concept of operations is to ascend to an initial altitude using a brushless motor and propeller, and then shut off the powertrain to glide. (The propeller folds backwards against the fuselage for reduced drag.) The hypothesis is that the mass "cost" of this electric propulsion unit is less than the mass cost of the heavier wing and tailboom required for a discus launch. This is the design space that the Feather project aims to explore.

A design optimization problem is formulated with the goal of minimizing the still-air sink rate. General configuration decisions were inspired by the Drela Apogee HLG [173]. The result is the aircraft shown in Figure 4-25, which is currently in the process of being built and flown. The glider overall weighs just 62 grams[39], which is roughly half the weight of a typical DLG of this wingspan. Much of this weight savings is made possible by the avionics subsystem, which weighs just 15.3 grams but includes a LiPo battery, a brushless motor, an ESC, a 5-inch folding propeller, two linear servos, an IMU and microprocessor for gyro stabilization, and a remote-control radio receiver. Still-air sink rate is estimated to be roughly 0.29 m/s (57 ft/min), which is competitive with the best DLGs of this wingspan.

AeroSandbox code used to develop this aircraft design is publicly available at `https://github.com/peterdsharpe/Feather-RC-Glider`.

[39]equivalent weight to roughly 14 sheets of standard copy paper


Feather **Motor-Glider**

Peter Sharpe

(design heavily inspired by Drela Apogee)

Dimensions in meters.

Overall Properties

All-up weight	61.9 g (2.18 oz)
L/D (ideal piloting)	12.4
Cruise Airspeed	3.53 m/s
Cruise AoA	5.08 deg
Cruise CL	0.58
Sink Rate	0.286 m/s
Cma (w/ visc. aero)	-0.46
Cnb (w/ visc. aero)	0.0435
Wing Reynolds Number	36k
AVL: Cma	-0.53
AVL: Cnb	0.0480
XFLR5: Cm (w/ visc. aero)	0 (trimmed)
AVL: Clb Cnr / Clr Cnb	2.24

Mass Breakdown

wing	22.73 g (0.80 oz)
tail surfaces	4.26 g (0.15 oz)
linkages	1.00 g (0.04 oz)
motor	4.49 g (0.16 oz)
motor bolts	0.30 g (0.01 oz)
propeller	1.54 g (0.05 oz)
propeller band	0.06 g (0.00 oz)
avionics board	4.30 g (0.15 oz)
battery	4.61 g (0.16 oz)
pod	7.00 g (0.25 oz)
boom	7.00 g (0.25 oz)
adapters + glue	4.58 g (0.16 oz)

Control Scheme

No control surfaces on wing (tail-only control). Tail surfaces are torsion-sprung using 9 thou music wire, Z-bent and CA-glued in.

Spectra line linkages connect printed-in control horns up to 2x linear servos on avionics board.

Flight Dynamics

- Phugoid ζ = 0.23
- Dutch roll
    - f = 0.76 Hz
    - ζ = 0.38
- Spiral τ = 1.38 s (stable)

Wing
S = 0.137 m^2 (212 in^2)
0.196
0.500
0.148
0.982
36°
11°
AG13 (5.8%) airfoil throughout
Avionics
5"x3" folding prop
6000 kv motor
Foam-3D-printed avionics mount
Foam-3D-printed joiner + mount
Flat-plate airfoil
0.073
Carbon-fiber tube for boom (nose to tail)
0.241
Foam-3D-printed V-tail mount
Cast out of rigid expanding foam, 2 lb/ft^3
3 mm OD carbon tube spar buried into wing (cast in)
60 grams-force static thrust
0.110
0.025
0.085
Neutral Point
⌀0.130
A
B
0.095
0.091
0
0.043
0.150
0.234
0.320
0.345
0.826
0.875
0.899
6000kv motor
0.032
Avionics board (RX, 2x servos, ESC, IMU + FC)
1S 150 mAh LiPo
Printed avionics mount
Detail A
Simple bevel
Foam-printed ρ = 0.40 g/cm^3
Guide for linkages
Printed-in living hinge
Printed-in control horn, (inside of V)
Detail B
Printed mount

Figure 4-25: Design drawing of the Feather ultra-lightweight hand-launched motor-glider.

Chapter 5

Traceable Physics Models

This chapter contributes computational implementations of several key physics models for aircraft design that are traceable within AeroSandbox, the design optimization framework introduced in Section 3.2. The broad motivation for this contribution stems from the observation that "ease-of-model-implementation" has historically proven to be one of the most important factors for determining whether an MDO paradigm can achieve use in industry. To that end, the goals of this contribution are to:

1. Stress-test the feasibility of code transformations in practice—how much added user effort and expertise is required to bring typical engineering analyses into a code-transformations-based MDO tool? To what extent can existing code be used as-is? Finally, the thesis aims to identify any specific computational elements that cause "pain points" when attempting to make an analysis traceable.

2. Jump-start future applied research by providing a set of modular, plug-and-play analyses that can be used to quickly build a variety of aircraft design optimization problems. Since the long-term goal of this research direction is to establish whether the proposed MDO paradigm improves practicality, many practical aircraft design problems must be posed. Creating a set of modular general-purpose building blocks reduces the need to write similar analysis code repeatedly, saving time for designers.

To achieve these goals, this thesis contributes the traceable implementations of the analyses given in Table 5.1. Several other analyses, including structural and propulsive models, were also implemented in the AeroSandbox framework, but are not described in this thesis document for brevity; the interested reader is directed to the AeroSandbox documentation for more information [1, 38]. The set of analyses selected for inclusion in this chapter was deliberately chosen to be mathematically diverse, spanning a wide range of common code patterns in scientific computing. The types of attributes that each analysis is intended to stress-test are given in the right-most column of Table 5.1.

Table 5.1: A list of aircraft design analyses that the thesis implements within a code transformations framework. The middle column lists the non-traceable tools for each analysis that are commonly used in industry today. The right-most column lists the computational attributes that each analysis is intended to stress-test.

Analysis To Contribute	**Non-Traceable Analogue**	**Tests tracing through...**
Vortex-Lattice Method Aerodynamics Analysis	AVL [148]	An aerospace geometry engine and discretization; large, vectorized matrix methods, like linear solves
Nonlinear Lifting Line Aerodynamics Analysis	Phillips and Snyder [174], Reid [175], Weissinger [176]	Nonlinear systems of equations (i.e., implicit), which often lead to value-dependent code execution (via convergence loops)
Workbook-style Aerodynamics Buildup	USAF Digital DATCOM [177]	Table lookups, large amounts of conditional logic (yielding a wide, branching graph), and scalar-heavy math
Rigid-Body Equations of Motion	ASWING (dynamics) [178]	Ordinary differential equations, which are often implemented in a loop-heavy way (yielding a deep graph)

These traceable implementations offer value within the context of the thesis itself (in stress-testing the practicality of code transformations), but they also have value as a standalone contribution to the aircraft design community—even outside of design optimization. Because of the mixed-backend numerics library described in Section 3.4.1, these analyses can be used independently from the design optimization context of AeroSandbox if desired. In such cases, however, they still can retain certain runtime benefits from the code transformations framework, if desired by the user. For example, a traceable workbook-style aerodynamics buildup would seamlessly enable GPU-accelerated evaluations and automatic vectorization, offering significant speedups; an example application might be real-time performance estimation for model predictive controllers or flight simulation.

5.1 Vortex-Lattice Method Aerodynamics Analysis

5.1.1 Method Overview

The vortex-lattice method (VLM) is a low-fidelity aerodynamics analysis used to model the inviscid 3D flow field around a system of lifting surfaces (e.g., wings). It is one of the most common conceptual-level aerodynamics analyses used in aircraft design, as it is computationally inexpensive and interpretable. Common tools that implement the VLM include AVL [148] and XFLR5 [179].

A VLM analysis is based on classical potential-flow theory. Because this flow field model is a linear partial differential equation, it can be quickly solved using a boundary-element method representation by superimposing Green's-function kernels. These kernels model disturbances in the flow field that are induced by the lifting surfaces. In a VLM analysis, these kernels are modeled as a collection of *horseshoe vortices* that are distributed along the wing in a regular lattice pattern (distributed in both spanwise and chordwise directions). A single horseshoe vortex is illustrated in Figure 5-1. Each horseshoe vortex is a connected polyline of uniform-strength vortex filaments, with three segments: a bound vortex on the wing, and two trailing legs extending downstream to infinity[1]. (In practice, a fourth leg consisting of a far-downstream "starting vortex" can be imagined to close the horseshoe vortex, which forms a ring vortex and thus satisfies the Helmholtz vortex theorems.) The Kutta condition is naturally satisfied, as the only place where the wing can shed vorticity is at the trailing edge (due to placement of the trailing legs).

[1]These trailing legs extend to the far-field Trefftz plane. In theory, these trailing legs should follow the local flow direction (and hence be "force-free" by the Kutta-Joukowski theorem), but often they are simply extended directly backwards which simplifies induced velocity computation and removes the need for an iterative wake relaxation.

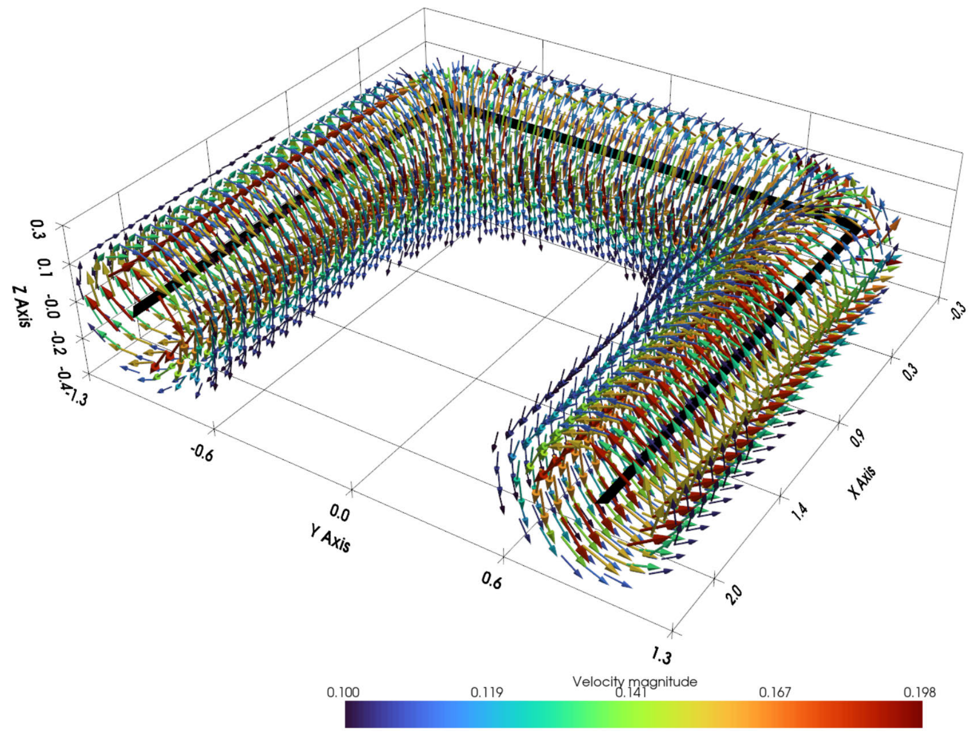


Figure 5-1: Illustration of a horseshoe vortex (black) and the induced fluid velocity field around it, which is the fundamental building block of a VLM analysis. In reality the induced velocity field has global influence, but it is truncated here for conceptual clarity.

Each horseshoe vortex has an initially-unknown strength, and the vorticity associated with each vortex creates an induced velocity that affects the global flowfield. To solve for these N unknown vortex strengths, N constraints are needed. A convenient choice is to impose a *flow-tangency* (also called *no-penetration*) boundary condition associated with each horseshoe vortex, where the flow velocity normal to each horseshoe vortex is zero. It is not immediately obvious at which location this flow-tangency condition (called the *collocation point*) should be imposed, relative to each horseshoe vortex. As it turns out, the best choice is to discretize the wing into quadrilateral panels, then place the bound leg at the quarter-chord point of each panel, and place the collocation point at the three-quarter-chord point. This choice results in higher-order convergence with respect to discretization resolution than any other choice, with derivation for this given by Katz and Plotkin [180]. (One way to intuitively understand this reasoning is that the VLM is essentially a 3D analogue of 2D thin airfoil theory.)

Because of the linearity of the governing equations, the unknown horseshoe vortex strengths can be solved as a linear system of equations. Due to the global influence of each vortex on the flowfield, this linear system of equations is dense and asymmetric, so it is typically solved using LU factorization. The solution to this linear system gives the vortex strengths, which can then be used to reconstruct the flowfield around the lifting surfaces. The vortex strengths are also conveniently equal to the local difference in pressure coefficient between the top and bottom surfaces of the wing, which can be used to visually interpret the flow field. Lift force computation is usually performed using the Kutta-Joukowski theorem on each bound leg. Drag force calculation (which only includes induced drag, as a VLM is inviscid) can be accurately performed using a Trefftz plane wake integral. Further details on this analysis formulation are available in work by Katz and Plotkin [180] and Drela [154].

5.1.2 Implementation

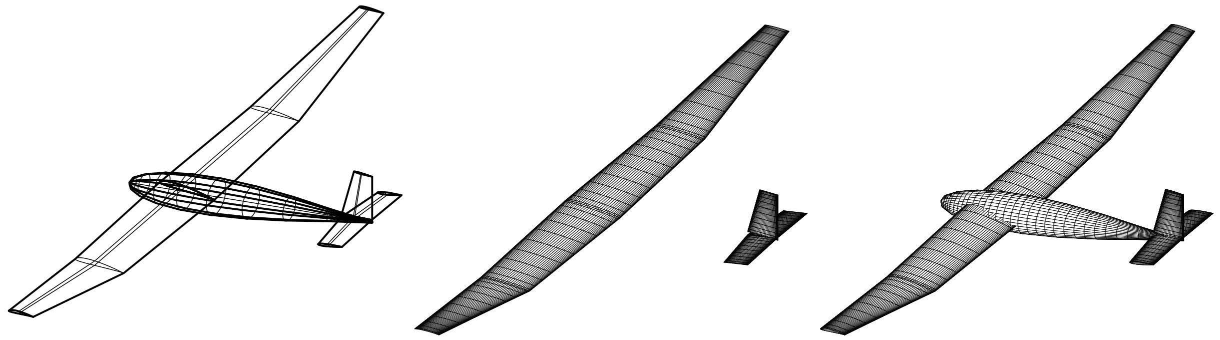
Discretization

Based on this theory, a vortex lattice method was implemented into AeroSandbox as a traceable analysis. One of tasks when implementing this analysis is discretizing the geometry (i.e., generate a mesh) in a code-transformations compatible way. Because discretization of a high-level geometry representation is such a common task in engineering analysis, an aircraft geometry stack was built within AeroSandbox to facilitate this. This stack allows one to represent an aircraft geometry at a conceptual level (i.e., within a hierarchical data structure that is not tied to any specific analysis), and then to generate a variety of degenerate geometry representations from this conceptual geometry. These degenerate geometry representations could be a series of 1D beams (for structural analysis), a mean camber line geometry representation (for a VLM aerodynamics analysis), a 3D panel geometry representation (for visualization or for CFD analysis), or any other representation that is needed for a specific analysis. Examples of such degenerate geometry representations that are possible are illustrated in Figure 5-2. This capability to separate the concept of geometry from its representation makes it much easier to implement multi-physics analysis that may be required in MDO problems.

Conceptually, the ideas behind this geometry stack are inspired by the degenerate geometry representation capabilities of OpenVSP [94], with the difference that this is built on top of a traceable numerics core. Because of this, these meshes gain all the properties enabled by code transformations, such as differentiable meshing. Furthermore, if analyses are built on top of this geometry stack, the analysis can be end-to-end differentiable throughout both discretization and solution.

This unified, vertically-integrated workflow contrasts with many current design optimization workflows, which instead usually treat meshing and solving as separate black-box tools. In cases where end-to-end gradients are required (i.e., aerodynamic shape optimization with respect to a set of design variables), the usual strategy is to compute gradients for each process and then later to stitch them together. For example, a meshing tool might use forward-mode automatic differentiation[2], while a solver might use a discrete adjoint of the governing PDE to obtain mesh gradients. Stitching these gradients together is a perfectly viable solution from a computational perspective, but as a practical matter it often requires a fair amount of boilerplate code from the user. In contrast, unifying the meshing and solution processes within a code transformations framework eliminates the need for the user to manually interact with component partials, lessening the expertise required to build a differentiable analysis.

[2]or, if the mesh has a deterministic connectivity (i.e., hyperbolic marching from a surface), one could use a method with similar properties, like complex-step differentiation or finite-differencing


(a) Concept-level geometry representation, corresponding to the raw data structures within the AeroSandbox geometry stack.

(b) Mean camber line degenerate geometry representation, which is used during VLM analysis.

(c) 3D panel degenerate geometry representation, which can be used in other aerodynamic analyses.

Figure 5-2: Illustration of several possible degenerate geometry representations produced by the AeroSandbox geometry stack. This removes the need for individual disciplinary analyses to re-implement their own meshing tools, creating a clearer API for new analysis tool development.

Solution Methodology Considerations

With meshing complete, the governing system of equations described in Section 5.1.1 can be implemented. Within the context of a code transformations framework, an interesting question is whether this system should be solved explicitly or implicitly. To clarify the difference here:

- In an **explicit** formulation, the matrix and right-hand-side vector of the linear system of equations are explicitly constructed within the analysis code, and the system is solved using a direct solver. In the context of a broader optimization problem encompassing such a model, this means that a) the system of governing equations is solved exactly at each iteration, and b) no optimization variables or constraints are added to the solve.

- In an **implicit** formulation, the horseshoe vortex strengths are posed as optimization variables, and the governing equations are implemented in residual form and constrained (in the top-level optimization problem) to be driven to zero. This is analogous to a simultaneous-analysis-and-design MDO problem formulation (shown in Figure 4-5). Here, the matrix and right-hand-side vector of the linear system are never explicitly constructed within the analysis code (though of course, the optimization problem constraint Jacobian will essentially compute this same information). In the context of a broader optimization problem encompassing such a model, this means that a) the system of governing equations is solved approximately at each iteration, and b) optimization variables and constraints are added to the solve.

Both methods have advantages. The principal advantage of the explicit formulation is that incorporating this into a design optimization framework can yield more stable convergence, since "correct" (i.e., zero-residual) solutions are guaranteed at each iteration. This can be especially important in the context of a multidisciplinary optimization problem, where this VLM analysis might be combined with other analyses. Here, a fully coupled implicit system might become numerically unstable (e.g., numerically-stiff), leading to convergence difficulties.

On the other hand, the implicit formulation has the potential to be much faster, in the context of a broader optimization problem. This is because the optimizer is solving for primal feasibility (i.e., constraint violation) and dual feasibility (i.e., optimality conditions) simultaneously. An intuitive way to think about this is that the optimizer is not wasting CPU cycles by solving the analysis accurately during early iterations, when the solution is far from the optimum. Instead, the optimizer is solving the analysis just enough to ensure that the constraints are satisfied, and then moving on to the next iteration. This can lead to significant speedups (up to an order of magnitude, in the author's experience) in the context of a broader optimization problem, especially when the analysis is expensive to solve.

There are also some cases where this implicit solution can be favorable in augmented analyses. For example, if a panel method is coupled with an integral boundary layer model (e.g., in XFoil), solving the problem as a coupled system rather than a segregated disciplinary solve can resolve singularity issues that might otherwise appear [95, 181–184].

In the case of the present VLM implementation, both formulations are offered, with the explicit solution being the default. This is because the explicit solution is more stable and easier to debug, and because the VLM analysis is relatively inexpensive to solve. However, the implicit solution is also available, and can be used if the user desires faster convergence in the context of a broader optimization problem.

5.1.3 Example Results

Figure 5-3 shows the results of a VLM analysis performed using AeroSandbox, using a generic glider design and a prescribed aerodynamic operating point. At the moderately high paneling resolution shown here ($N = 700$ panels), solution takes about 0.25 seconds on a laptop-grade CPU. The surface color shows the vortex strength (or equivalently, ΔC_p across the wing), while the streamlines show the flow field.

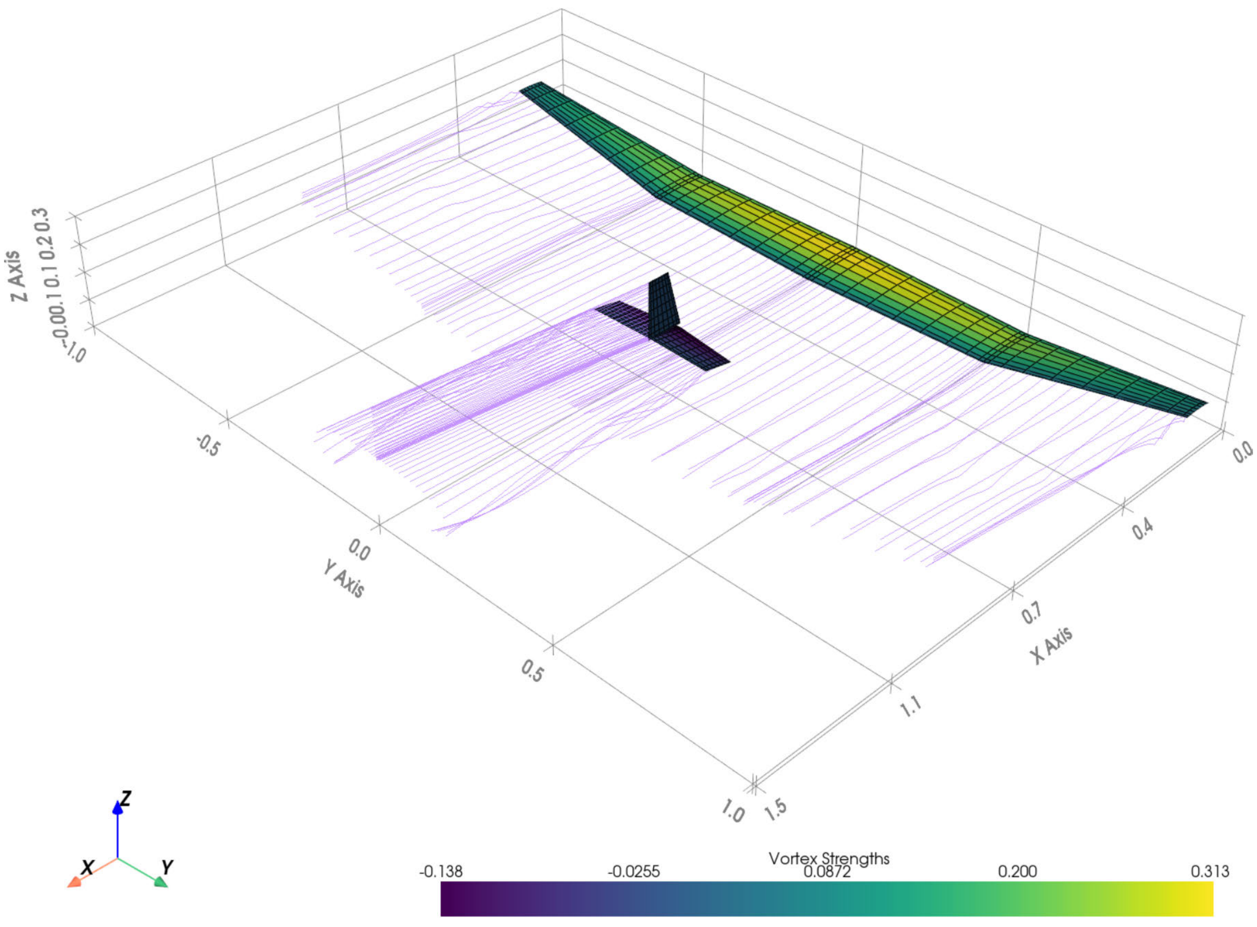


Figure 5-3: AeroSandbox VLM analysis results for a simple glider geometry at a prescribed aerodynamic operating point. Surface color shows the vortex strength (or equivalently, ΔC_p across the wing). Streamlines show the flow field.

Quantitative validation results for this VLM analysis are shown later in Section 5.4, where the VLM can be cross-compared not only against external tools but also against the other plug-and-play aerodynamic analyses developed in this chapter. Notably, while the VLM method is quite accurate for computing induced drag, it has no mechanism to compute profile drag without augmentation by another aerodynamic method, like the buildup described in Section 5.3; hence, the VLM method should not be used by itself for performance analysis. It is, however, more than adequate for flight dynamics analysis, as there are few other low-fidelity aerodynamics methods that handle local velocity effects (e.g., dynamic derivatives, aerodynamic damping) as accurately.

5.1.4 Aerodynamic Shape Optimization

To demonstrate how this VLM tool can be incorporated into a broader optimization problem, we can formulate a classic aerodynamic shape optimization problem and compare our result to a known solution.

Here, we aim to find the minimum-induced-drag wing planform, by optimizing a wing's chord distribution. We assume:

- A fixed total lift
- A fixed wing area
- A fixed span
- An untwisted, uncambered, unswept, thin, planar wing
- Potential flow (inviscid, incompressible, irrotational, and steady)

For this fixed-span, fixed-lift case, theory shows that the minimum-induced-drag wing has an elliptical lift distribution. With the additional assumptions above, the corresponding minimum-induced-drag wing will also have an elliptical *chord* distribution [185].

This problem can be posed in AeroSandbox syntax using the plug-and-play VLM analysis model described in the previous section. We supply an initial guess of a simple rectangular (i.e. untapered) wing, and optimize the chord distribution. We do not know the correct angle of attack α to achieve our specified lift coefficient *a priori*, so this becomes an additional optimization variable. The VLM is implemented, here using just one spanwise panel for each unknown chord variable—typically, a higher resolution (~ 4 panels per optimization variable) would be used for numerical stability, but this is used to demonstrate numerical robustness of the implementation here.

Surprisingly, the entire optimization problem can be written in less than 40 lines of code, allowing it to be shown in its entirety in Listing 4. This serves as a good illustration of the motivation behind developing these plug-and-play models for a code transformations framework—the modularity allows complex optimization problems to be posed with minimal code.

```python
import aerosandbox as asb
import aerosandbox.numpy as np

opti = asb.Opti()  # Initialize an optimization environment.

N = 16  # Spanwise resolution
y = np.sinspace(0, 1, N, reverse_spacing=True)  # Spanwise locations along the wing.
chords = opti.variable(init_guess=1 / 8, n_vars=N, lower_bound=0)  # Chord dist.
wing = asb.Wing(  # Defines the wing geometry.
    symmetric=True,
    xsecs=[  # Cross sections ("XSecs") of the wing
        asb.WingXSec(
            xyz_le=[  # Location of each cross-section's leading edge
                -0.25 * chords[i],  # This keeps the quarter-chord-line straight.
                y[i],  # Our (known) span locations for each section.
                0
            ],
            chord=chords[i],
        )
        for i in range(N)
    ]
)

aero = asb.VortexLatticeMethod(  # Compute aerodynamics using the VLM analysis
    airplane=asb.Airplane(wings=[wing]),  # The geometry to analyze
    op_point=asb.OperatingPoint(  # Aerodynamic operating condition
        velocity=1,  # A fixed velocity; unimportant due to nondimensionalization.
        alpha=opti.variable(init_guess=5, lower_bound=0, upper_bound=30)  # Angle of attack
    ),
    spanwise_resolution=1,  # Uses one panel per wing cross-section
).run()

opti.subject_to([
    aero["CL"] == 1,  # We want a fixed lift coefficient
    wing.area() == 0.25,  # We want a fixed wing area
])

opti.minimize(aero["CD"])
sol = opti.solve()
```

Listing 4: AeroSandbox code to optimize the chord distribution of a wing to minimize induced drag.

The optimization problem is solved to tolerance within 18 IPOPT iterations, corresponding to around 10 seconds of wall-clock time on a laptop CPU. This yields the wing geometry shown in Figure 5-4, which is indeed elliptical in chord distribution. To confirm, the chord distribution can be directly compared to an elliptical distribution as shown in Figure 5-5. Very slight differences appear due to the fact that the vortex lattice method does not make the assumption of locally-2D flow. With the added consideration of 3D relief, chords near the tip must be slightly wider than elliptical to achieve elliptical loading [186].

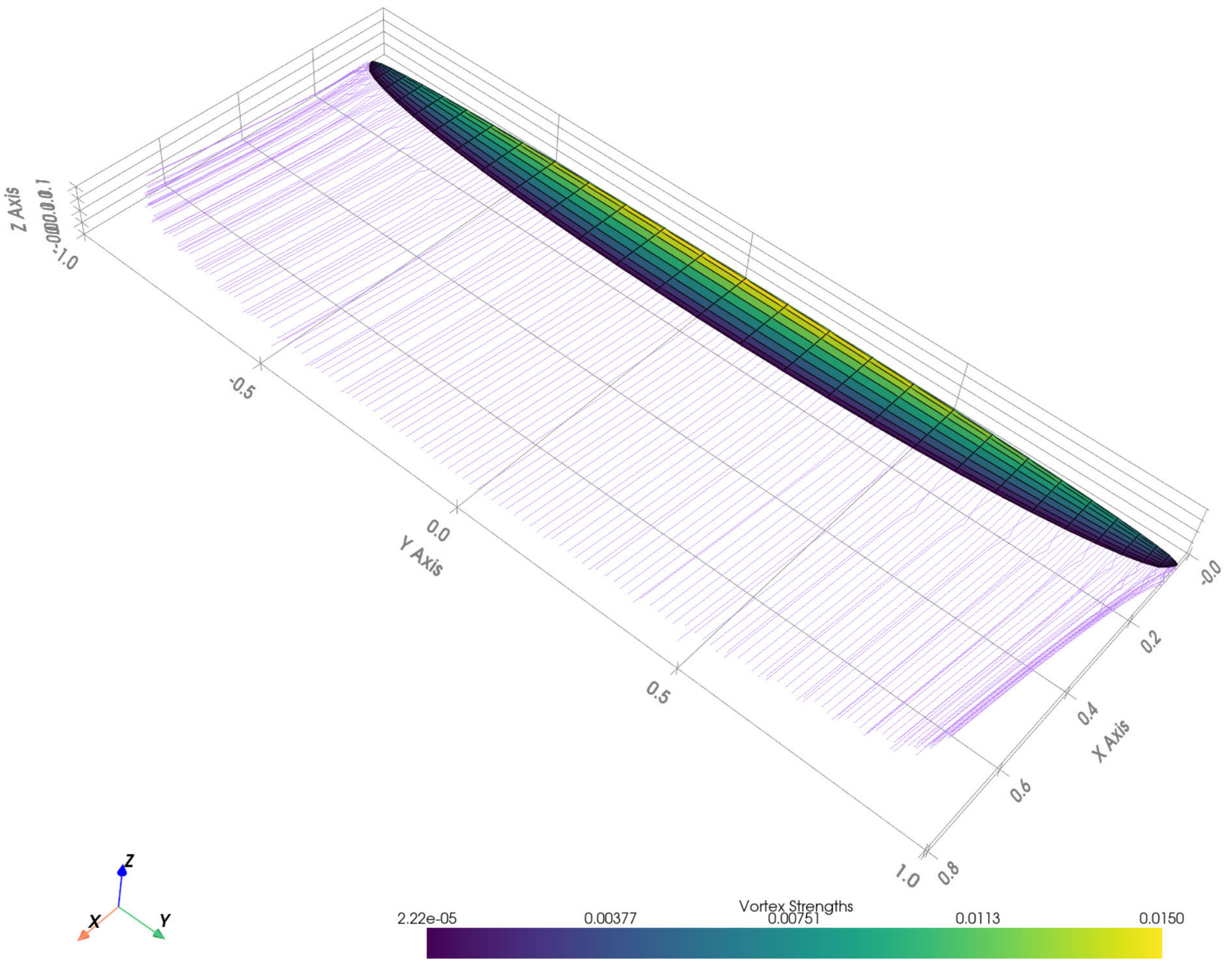


Figure 5-4: Result of the AeroSandbox VLM analysis-based aerodynamic shape optimization problem. The optimized wing has an elliptical chord distribution.

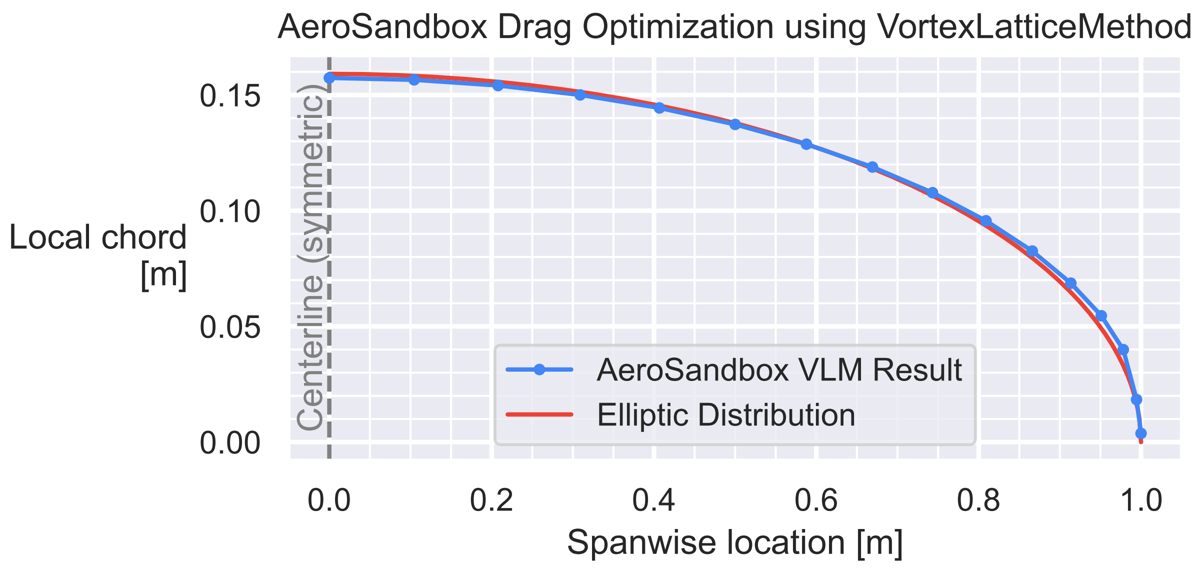


Figure 5-5: Comparison of the optimized chord distribution to the theoretical elliptical chord distribution.

5.2 Nonlinear Lifting Line Aerodynamics Analysis

5.2.1 Method Overview

While the VLM described in Section 5.1 is a useful tool in conceptual analysis, its inviscid nature means that it cannot capture many important aerodynamic effects. This includes profile drag effects, but also more subtle effects like nonlinear sectional lift curve slopes and local stall. Another downside of the VLM is that capturing camber effects requires a substantial number of chordwise panels, which quickly increases the size of the linear system[3] without meaningfully improving the induced drag prediction (as this is dominated by spanwise effects).

One way to address both of these limitations is to use a lifting-line method instead of a VLM. A

[3]The solve time of this linear system scales as $\mathcal{O}(N^3)$ with the number of panels, so keeping the number of panels low is quite valuable.

modern lifting-line method is best conceptually understood as a vortex lattice method with several key similarities and differences:

1. First, just as with the VLM, the lifting-line method uses a potential flow model with horseshoe vortices to model the flow changes induced by the vehicle.

2. Another difference is that in a lifting-line method the governing equation no longer comes from a flow-tangency condition. Instead, the constraint relationship comes from equating the lift force generated by the vortex (via the Kutta-Joukowski theorem) to the lift force computed using a 2D sectional model (i.e., airfoil-level aerodynamic models) and the local angle of attack. This local angle of attack includes the effects of induced downwash from the other vortices in the system, capturing 3D inviscid effects.

3. A notable difference is that in a lifting line method only one chordwise panel is required, because camber effects are not captured by direct flow tangency over the curved surface.

This strategy is related to prior work to generalize the lifting-line method by Reid [175], Phillips and Snyder [174], and ultimately has roots in work by Weissinger [176]. The key insight is that the lift force generated by a vortex can be related to the lift force computed using 2D sectional data by a constraint relationship. This constraint relationship allows the lifting-line method to leverage the accuracy of 2D sectional data, which can include camber effects. In almost all existing lifting-line methods, the sectional data is approximated using an affine C_L-α relationship (possibly with a nonzero C_L at $\alpha = 0$ to capture camber effects), because this reduces the governing system of equations to a single linear solve (just as with the VLM). This is computationally faster to solve than the true nonlinear problem. However, this also loses substantial accuracy in lift prediction, especially in cases where nonlinearities in the lift curve are important, such as near transitional Reynolds numbers and in high-lift cases. In many cases, this may make the traditional linear lifting-line method unsuitable.

Because of this, the present work implements *both* a linear and nonlinear lifting line method, with the latter retaining the full (nonlinear) lift curve slope behavior. This is done by reformulating the method to be implicit, with governing equations converted into nonlinear residual form.

In both the linear and nonlinear methods, 2D sectional data is provided using *NeuralFoil*, a custom physics-informed machine learning method described in Chapter 7. Using NeuralFoil to obtain sectional data has negligible error compared to XFoil solutions, and it has the important property of retaining C^1-continuity that allows for gradient-based rootfinding methods to be used in the nonlinear case. With the linear method, the NeuralFoil evaluation is numerically differentiated about the geometric angle of attack of each wing cross section[4] to construct an affine $C_L(\alpha)$ dependency. Critically, because of this local linearization step about the geometric angle of attack, even the linear lifting-line method in the implementation here can capture limited stall effects—this is a unique feature of the present work compared to other lifting-line implementations.

The overall lifting line methodology presented in this section represents modern thinking about the theoretical formulation of lifting-line methods, as given by Phillips and Snyder [174] and Reid [175]; this is a generalization of the historical lifting-line formulation originally given by Prandtl in the early 20th century. The general formulation presented here has been shown to work well for non-straight lifting lines, which occurs in wings with sweep or dihedral [174, 175, 187, 188].

[4]In other words, the angle of attack with respect to the freestream velocity vector (which can be computed explicitly), not the local perturbed fluid velocity vector (which is an implicit function of the initially-unknown vortex strengths)

5.2.2 Implementation

The linear version is implemented similarly to the VLM implementation, where both an explicit and an implicit solve are possible. On the other hand, the fully nonlinear lifting-line method is currently only implemented as an implicit analysis.

Numerical discretization is performed by thin-surface meshing with a chordwise paneling resolution of one. The quarter-chord lines of these panels can then be connected to obtain a lifting line for the wing.

5.2.3 Example Results

Figure 5-6 shows analogous lifting-line analysis results for the same glider example as the VLM analysis. The reduction in discretization resolution due to the elimination of chordwise refinement is immediately apparent from the lower number of discrete panels. This leads to much shorter asymptotic runtimes, especially for vehicles with complex systems of multiple interacting lifting surfaces.

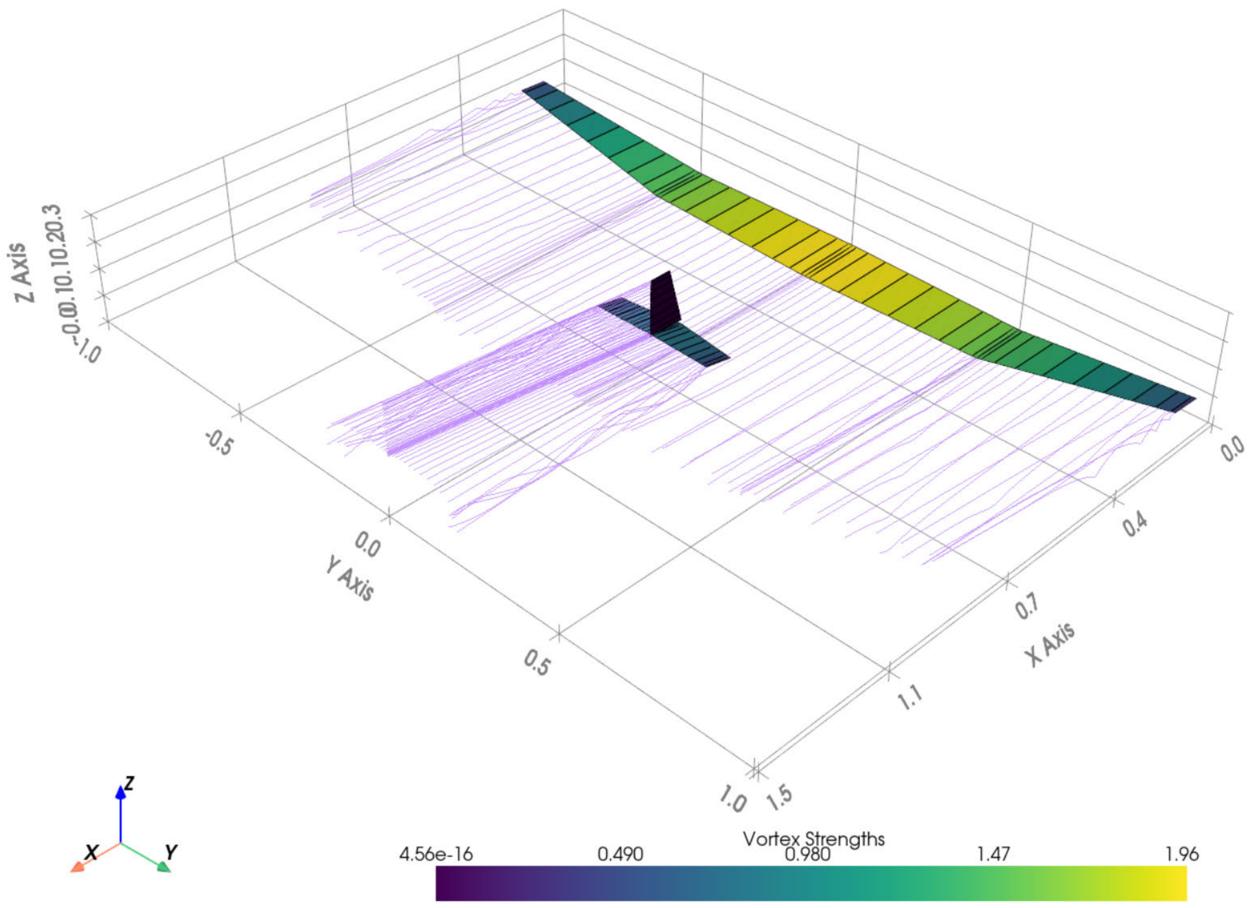


Figure 5-6: AeroSandbox lifting-line analysis results for a simple glider geometry at a prescribed aerodynamic operating point. Surface color shows the vortex strength (or equivalently, ΔC_p across the wing). Streamlines show the flow field.

As with the VLM, quantitative validation results for this lifting-line analysis are shown later in Section 5.4. In general, the lifting-line method is expected to be more accurate than the VLM for aerodynamic performance prediction purposes (i.e., lift, drag, and moment prediction), due of the incorporation of 2D sectional data. The lifting line also generally leads to more interpretable results, as the local angle of attack, lift coefficient, downwash, and other properties of interest can be directly returned by the analysis. (In a VLM, these quantities must be obtained by integration along the chord, which introduces error and creates extra postprocessing work depending on the data structure of the mesh.)

On the other hand, the VLM has advantages in low-aspect-ratio cases (as spanwise flow becomes more significant, invalidating the force-free wake assumption on the straight trailing vortices), in other highly-3D flow fields, and for angular rate stability derivatives (as the VLM can better capture chord-wise variations in local velocity).

5.2.4 Multidisciplinary Coupling Case Study: Aerostructural Analysis

The lifting line model developed in this section provides a good opportunity to demonstrate the ability to easily implement coupled multidisciplinary analyses within AeroSandbox. In this case, we can demonstrate a simple static aerostructural analysis of a wing, where the lifting line model is connected to a 6-DoF Euler-Bernoulli beam model.

Here, we study a wing with a fixed jig shape, which is shown in gray in Figure 5-7. This jig shape involves a tapered wing with an aspect ratio of 12, a straight quarter-chord, no dihedral, and a constant DAE-11 airfoil. The spanwise chord distribution of the wing $c(y)$ follows the proportionality relation below:

$$c(y) \propto \sqrt{1 - \frac{y}{b/2}} \qquad \text{for } 0 \leq y \leq \frac{b}{2} \tag{5.1}$$

where: $c(y)$ = The chord length at spanwise location y, where $y = 0$ is the centerline.

b = The wing span.

The structural properties of the wing are modeled as varying as a function of the local chord, with proportionality relations below. Constants are chosen such that the wing is quite flexible, which allows for a more interesting aerostructural interaction.

$$\begin{aligned} EI(y) &\propto c(y)^3 \\ GJ(y) &\propto c(y)^3 \end{aligned} \tag{5.2}$$

where: $EI(y)$ = The bending stiffness of the wing at spanwise location y.

$GJ(y)$ = The torsional stiffness of the wing at spanwise location y.

The wing structural equations have Dirichlet boundary conditions at the root and Neumann conditions at the tip, consistent with a cantilevered symmetric wing. These equations model both bending and torsion effects on the wing. For simplicity, only aerodynamic loads are considered in the analysis, with no additional external loads (e.g., gravity).

The aerodynamic operating point is at an angle of attack of $5°$ (measured at the fixed wing root), with a freestream velocity high enough to produce interesting aeroelastic effects.

The resulting aerostructural analysis is implemented in AeroSandbox by coupling the lifting-line analysis to the Euler-Bernoulli beam model. Both models use an identical discretization, which is facilitated through the degenerate geometry representations created by the AeroSandbox geometry stack (as described in Section 5.1.2). Coupling is implemented through an all-at-once approach, where the governing equations for the aerodynamic and structural models are solved simultaneously.

This solve quickly obtains an aerostructural equilibrium condition in just 3 iterations, which is sensible as both the aerodynamic and structural models are almost-linear. Figure 5-7 shows the deformed wing shape in blue, relative to the original jig shape in gray. The wing has a noticeable upward deflection and washout at the tip, which is expected due to the aerodynamic lift distribution. Figure 5-8 shows both the aerodynamic and structural quantities of interest as a function of the spanwise location y.

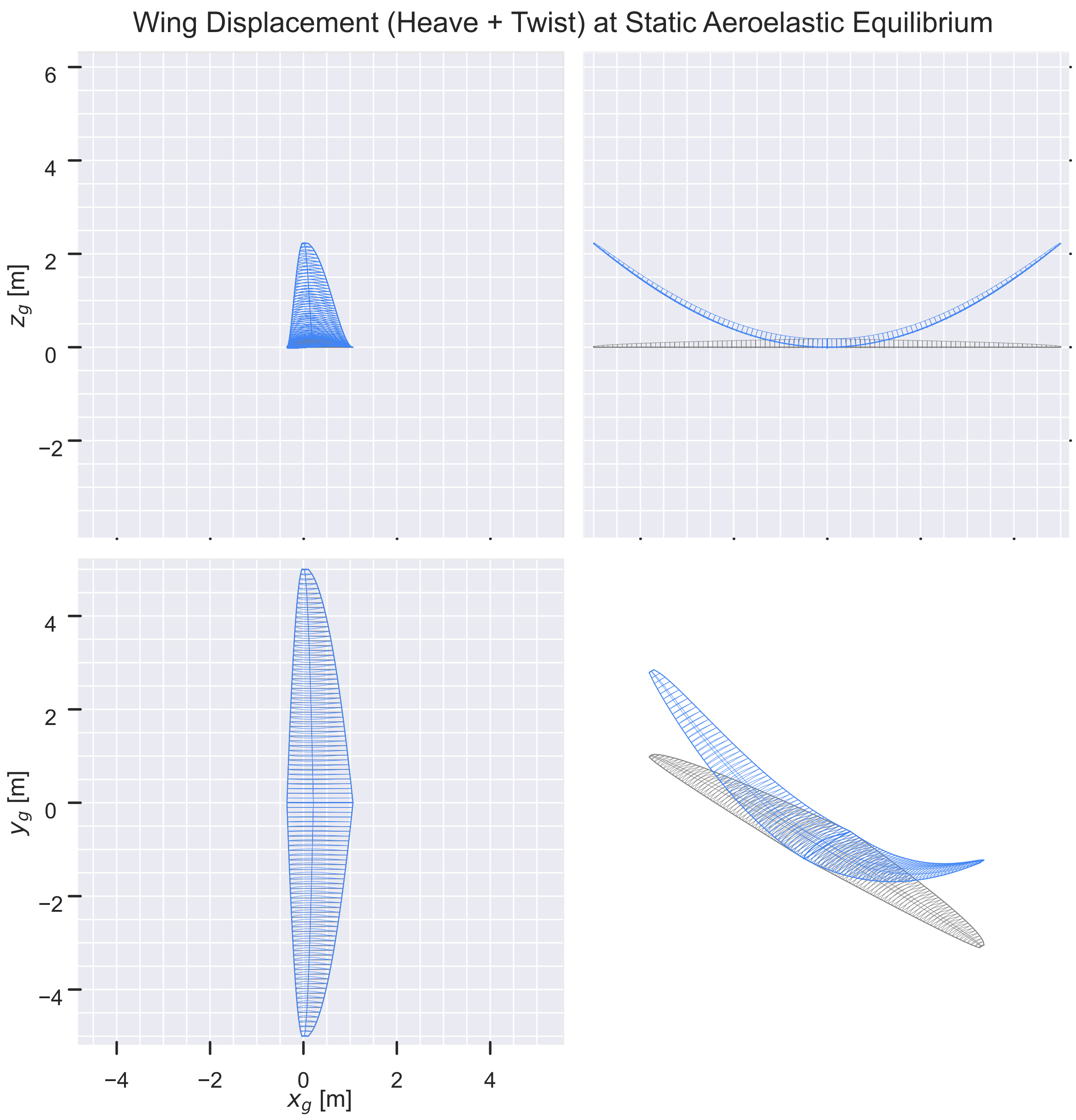


Figure 5-7: Fully-coupled aerostructural analysis results for a simple wing geometry. The deformed wing at static aerostructural equilibrium is shown in blue, with the original jig shape shown in gray.

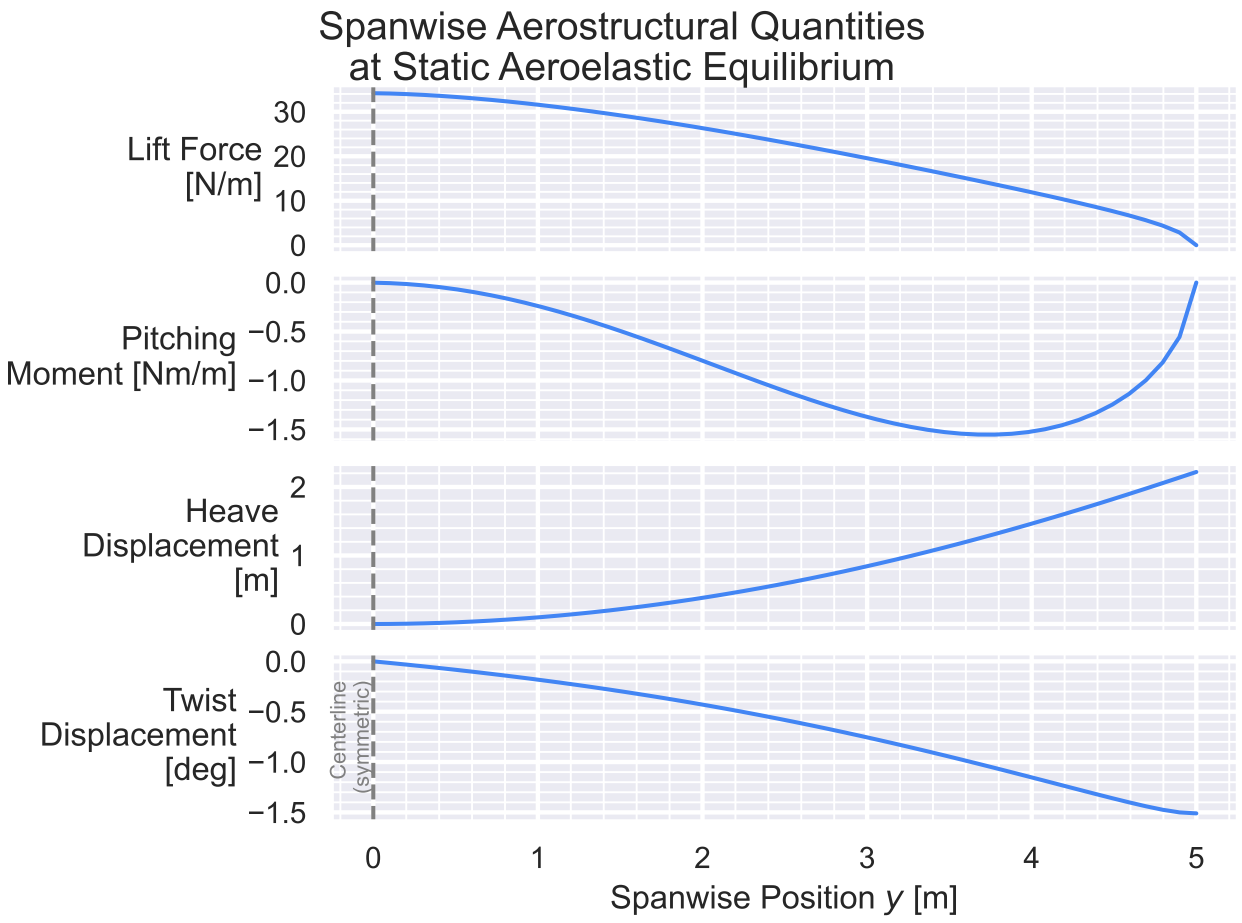


Figure 5-8: Aerodynamic and structural quantities of interest as a function of the spanwise location y. Pitching moment is measured about the wing's local quarter-chord axis.

5.3 AeroBuildup: A Workbook-style Aerodynamics Buildup

This section includes content from the author's contributions to the AeroSandbox documentation [38].

5.3.1 Method Overview and Implementation

While the VLM and lifting-line methods in Sections 5.1 and 5.2 provide conceptual-level aerodynamics analyses with strong first-principles grounding, they are limited in their ability to capture many important aerodynamic effects. For example, compressibility effects, fuselage aerodynamics, and high-angle-of-attack aerodynamics all go beyond the capabilities of these methods. Furthermore, these methods can have scalability challenges in combined-vehicle-and-trajectory optimization problems, depending on problem formulation. For example, the *Firefly* design problem of Section 4.2 requires aerodynamic solutions to be computed at *each* of 200 unique points along the flight trajectory, at each iteration. Computing these solutions using a VLM or lifting-line method[5] increases the total optimization runtime to the point where real-time interactive design is no longer attractive (i.e., an hour or longer), limiting the tool's utility in gathering conceptual design intuition.

To address these problems and provide a practical alternative, AeroSandbox includes a rapid, vectorized aerodynamics engine called AeroBuildup. AeroBuildup's methodology essentially uses a workbook-style buildup to compute aerodynamic forces, moments, and (optionally) stability derivatives. It attempts to model viscous and compressible effects on wings and fuselages across any orientation (360 degrees of α, β) and arbitrary vehicle angular rates (p, q, r). Just as with the VLM and lifting-line methods, the input is both the vehicle geometry[6] and an aerodynamic operating point[7].

AeroBuildup combines a variety of both theoretical and empirical models, inspired heavily by the methodology used by USAF Digital DATCOM [177]. (Indeed, many of the submodels are directly shared with Digital DATCOM.) Compared to DATCOM, however, AeroBuildup has several advantages that make it more favorable as an analysis to use within a larger aircraft design optimization problem:

1. AeroBuildup aims to *smoothly* model across a wide variety of flight regimes; the model returns C^1-continuous aerodynamic results that span from subsonic to supersonic speeds, from attached to

[5]The astute reader will notice that, in the case of a linear method where geometry and aerodynamic operating point can be separated (like the VLM), one possible strategy to improve this scaling is by saving the LU-factorization and back-substituting unique right-hand-side vectors. However, no analogous strategy exists if either the method is fully nonlinear or if the coefficient matrix depends on the operating point, as is the case with the lifting line implementation here.

[6]The vehicle geometry includes a reference point, which is used for moments and stability derivatives.

[7]An operating point consists of the local atmospheric conditions, airspeed, orientation (angle of attack α and sideslip angle β), and angular rates (p, q, and r about the three principal aircraft body axes).

separated flow, and from the lowest to highest Reynolds numbers. This contrasts with DATCOM, which requires the user to select submodels appropriate to the flight regime (e.g., small-scale transonic flight), and produces discontinuous results in some cases (e.g., as soon as Mach number crosses a threshold, DATCOM instantly switches to a different wave drag model, which causes a discontinuity in the drag prediction).

2. AeroBuildup makes minimal assumptions about the vehicle configuration. For example, with AeroBuildup any number of wings and fuselages may be specified, and these may be placed in any location; DATCOM forbids certain configurations, such as those with multiple fuselages. As another example, all lifting surfaces in AeroBuildup are simply "Wing" objects–the user isn't required to provide any extra semantic meaning (i.e., whether a surface is a horizontal vs. vertical stabilizer) that would change physics modeling. This contrasts with DATCOM, which requires explicit labeling. The end result is that AeroBuildup enables optimization with respect to parameters that are simply not possible with DATCOM[8].

3. AeroBuildup is end-to-end vectorized, which means that multiple aerodynamic operating points can be analyzed simultaneously with hardware-level (SIMD) parallelism. This is especially advantageous in a semi-empirical model like AeroBuildup, as many of the submodels consist of scalar-heavy analytical equations where vectorization buys significant speedups. Because of this vectorization, AeroBuildup is fast enough to be used in combined-vehicle-and-trajectory optimization problems, as demonstrated in Sections 4.2 and 4.3.

4. AeroBuildup is implemented in an accessible high-level language, Python, which makes it easier

[8]For example, imagine an airplane with a V-tail, where we wish to optimize the V-tail dihedral angle. DATCOM requires this to be entered as either a horizontal or vertical stabilizer, but it is not clear at what V-tail dihedral angle that switch should be made. This distinction changes DATCOM's results, but of course in reality the actual flow physics should not depend on the semantic label that we give it.

for end-users to modify or extend the analysis with problem-specific considerations. The AeroBuildup codebase is also modular and hierarchical, with heavy code re-use. For example, the same lines of code are executed for aerodynamics analysis of all wing-like components; this contrasts with DATCOM, which uses specialized code for different surfaces (e.g., a horizontal stabilizer vs. canard). This gives AeroBuildup a concise codebase with clear, isolated scopes, minimizing the logic that a new user must understand in order to make a modification.

The basic philosophy of AeroBuildup's approach revolves around a) hierarchical decomposition of the vehicle geometry, b) aerodynamic modeling of individual pieces, and then c) corrections for interactional effects between those pieces. For example, AeroBuildup computes aerodynamics on a per-wing and per-fuselage level, and further breaks down each of these into sectional aerodynamics.

For each wing section, the freestream velocity is computed in the local frame, with streamwise, normal, and crossflow velocities. A rotation-induced local flow velocity is also added to this, which later enables downstream computation of dynamic stability derivatives. This approximate local flow information is used to compute a set of *tentative* sectional aerodynamics (including viscous and compressible effects) using NeuralFoil, a physics-informed machine learning model described in Chapter 7. These tentative sectional aerodynamics are then corrected for the wing's self-induced downwash, for 3D compressible effects, and for other 3D effects (e.g., the unsweeping of a lifting line near the root of a swept wing). Induced drag is computed by assuming the wake flows back to the Trefftz plane by projecting the wing's lifting line in the direction of the freestream velocity. Induced drag is computed at an overall-aircraft-level rather than a wing-level, so wakes between separate lifting surfaces are correctly merged for these purposes. After these corrections, the sectional aerodynamics are integrated along the span to obtain the total aerodynamic forces and moments.

Currently, a notable omission of the AeroBuildup model is the influence of downwash from *other* wings for the purposes of lift computation. This has the benefit of improving computational speed by eliminating a lifting-line-like linear solve, but it can lead to inaccuracies in the presence of strong interactions between lifting surfaces. (For example, when analyzing a conventional-configuration airplane using AeroBuildup, the effectiveness of the horizontal stabilizer will usually be slightly overestimated due to the missing downwash influence of the main wing[9].) This modeling decision ultimately comes down to a complexity vs. accuracy tradeoff, and future versions may offer to include these interactional effects as an option.

Fuselage modeling generally follows methods described by Jorgensen [189, 190] and MIL-HDBK-762 [191]. More precisely, inviscid results are theoretically grounded in slender body theory [154], while viscous effects are modeled using an infinite-length crossflow analogy with finite-length corrections. The AeroSandbox geometry stack admits both circular and non-circular fuselages [10], and the dependence of crossflow separation location on local fuselage curvature (i.e., corners) is modeled. Wake separation, base drag, and any associated extra lift generation is handled using empirical methods fitted to wind tunnel results on a series of typical body shapes. Likewise, local velocity overspeed and any associated wave drag in transonic cases is computed using empirical relations. These compressible considerations depend on a variety of fuselage shape factors, but most notably on the effective fineness ratio of the pressure-drop region (i.e., roughly the forward half) of the fuselage.

AeroBuildup is a fully explicit calculation method, which means that a result is guaranteed in bounded computational time, without any non-convergence issues. However, the accuracy of this result is strongly dependent on the flight regime of the case in question. For example, AeroBuildup

[9]One way to conceptually think about this is that the effective lift-curve-slope of the horizontal stabilizer, $C_{L\alpha,h}$, should be slightly reduced due to main-wing downwash; but this effect is not captured.

[10]Precisely, fuselage cross sections may be superellipse curves with any positive value of the shape parameter.

is expected to be most accurate for typical flight conditions where theory is sound and experimental data for validation sub-models is widely available. Generally, this well-behaved region includes flows without regions of massive separation or strong shocks. (AeroBuildup also remains relatively accurate in cases with low and transitional Reynolds numbers, due to viscous methods described in Chapter 7.) Quantitative validation results for AeroBuildup against other methods are shown later in Section 5.4.

On the other hand, cases at extreme flight conditions (e.g., $\alpha = 90°$, $M_\infty = 1.00$) will be much less accurate. However, even in these cases, effort has been taken such that the results from AeroBuildup are at least order-of-magnitude correct. This improves optimization robustness, as even very strange intermediate optimization points will allow the optimizer to gain gradient information and make progress back to the region of reasonable designs.

Beyond optimization applications, however, this property of bounded computational time gives AeroBuildup in various situations that require real-time results. For example, AeroBuildup can be used as a rapid aerodynamics model within flight simulation or real-time control.

5.3.2 Example Results

To illustrate AeroBuildup's ability to rapidly compute aerodynamic results, consider the case of the *MIT Firefly* air vehicle, which is described further in Section 4.2. The geometry of this air vehicle is shown in Figure 4-6 using the AeroSandbox geometry stack. Firefly was selected to illustrate some of AeroBuildup's capabilities because its mission CONOPS (Figure 4-2) subjects the vehicle to an uncommonly large range of interesting aerodynamic operating conditions. These include:

1. High-angle-of-attack flight, which occurs immediately after deployment from the host vehicle. The vehicle may be tumbling at this point, and must passively stabilize to controlled flight.

2. High-transonic flight, which occurs during the initial boost phase of the vehicle's trajectory. The initial Mach number at launch is 0.8, and depending on the combination of trajectory constraints that the user specifies, Mach numbers slightly over Mach 1 may be encountered during flight.

3. Low-Reynolds-number flight, which occurs at all phases but especially at high altitude. For example, at the apogee of the trajectory shown in Figure 4-7, the flow Reynolds number referenced to Firefly's mean aerodynamic chord is just 53×10^3.

Because of these considerations, Firefly is an excellent test case to show the viability of AeroBuildup to span these various flight regimes in a single model without requiring user intervention. In these cases, the following reference quantities are used, which are derived from the projected area, mean aerodynamic chord, and span of the vehicle's main wing:

$$S_{\text{ref}} = 0.02335 \text{ m}^2 \qquad c_{\text{ref}} = 0.05391 \text{ m} \qquad b_{\text{ref}} = 0.4800 \text{ m} \tag{5.3}$$

Traditional Aerodynamics Polars

To begin, we can compute the traditional static aerodynamic polars, which is an "alpha sweep": lift, drag, and moment coefficients as a function of the angle of attack α. In Figure 5-9, we show the AeroBuildup-computed polars for the MIT Firefly air vehicle, at a representative flight condition of sea level flight at Mach 0.15. This is comparable to the aerodynamic conditions on the Firefly vehicle at the end of its nominal trajectory[11], as shown in Figure 4-7. For this analysis, control surface deflections of zero are used.

[11]Firefly's optimized trajectory during the glide phases predictably has a nearly-constant indicated airspeed, of around 86 KIAS (44 m/s at sea level). Slight variations occur due to Reynolds effects.

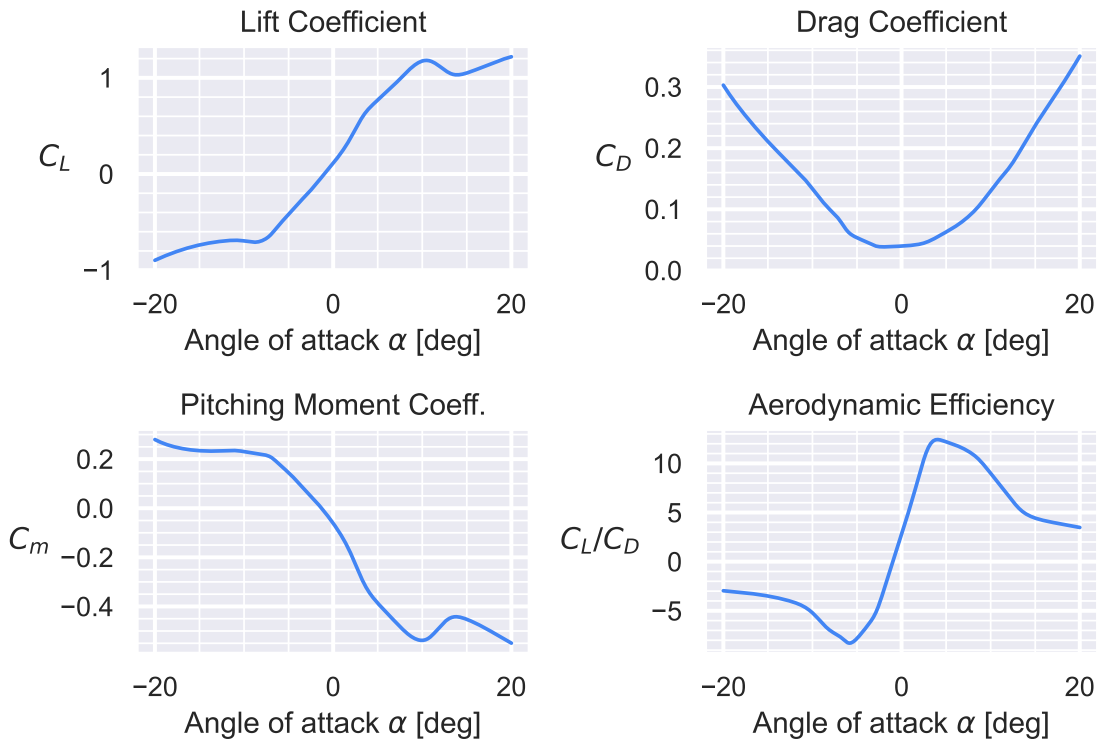


Figure 5-9: Standard static aerodynamic polars for the MIT Firefly air vehicle, computed using AeroBuildup. Analysis is an angle of attack (α) sweep at Mach 0.15 and sea level conditions. Control surfaces are not deflected.

To generate the smooth plots in Figure 5-9, 1,000 unique aerodynamic operating points were analyzed with AeroBuildup (each at a different α). This process is benchmarked at 3.13 seconds in total on a laptop-grade CPU, or around 3 milliseconds per operating point. This is achievable due to data-level vectorization, allowing for instruction-level parallelism (SIMD) to be automatically performed. (To illustrate this, a single non-vectorized analysis takes roughly 1 second.) In either case, however, this speed combined with the ability to model viscous and compressible effects makes AeroBuildup useful for rapid aerodynamic assessment of new concepts or for aerodynamic database generation.

Figure 5-9 demonstrates that AeroBuildup captures several nuanced aerodynamic effects. For example, the lift curve captures both positive and negative stall conditions, and a lift-curve-slope nonlinearity during the attached flow regime near $\alpha = 4°$ occurs due to sudden early upper-surface turbulent transition along the main wing. Likewise, in the moment polar, we observe a general curving-down nonlinearity, which is ultimately a consequence of the vertical displacement between the center of gravity and the neutral point. The effect of tail stall is also visible near $\alpha = 13°$ in the moment polar, which causes a slight destabilizing effect. We also note that the moment polar shows that the vehicle requires a modest pitch-up input in order to trim at the target lift coefficient during glide ($C_L \approx 1$), which matches flight test reports by Gaubatz [12].

Stability Derivative Calculation

AeroBuildup also supports the ability to compute stability derivatives (both static and dynamic), due to its local flow velocity computation. By default, this is computed by evaluating finite differences of the aerodynamic operating point with respect to angle of attack, sideslip, and angular rates (α, β, p, q, r). This approach is the fastest strategy, because these additional analyses can be added to the vectorized evaluation enabling negligible overhead.

As an alternative to finite-differences, an interesting capability that results from the code transformations paradigm is the ability to *directly* compute stability derivatives using automatic differentiation on AeroBuildup itself. Where this gets particularly unique is if this stability derivative then gets used in an optimization problem, as higher-order automatic differentiation may be required. For example, imagine we formulate an aircraft design optimization problem where the objective function is to minimize $C_{m\alpha}$. Here, we might compute $C_{m\alpha}$ by applying automatic differentiation to AeroBuildup (1 derivative). Then, we compute an exact Hessian of $C_{m\alpha}$ with respect to the design variables (2 more derivatives), to facilitate gradient-based optimization. In total, we may be performing triple-nested automatic differentiation of the original AeroBuildup analysis.

Because the operator space of the AeroSandbox numerics stack is closed under differentiation, implementing this higher-order AD is surprisingly straightforward, and has been successfully tested. This fascinating capability offers unique capabilities when optimizing with respect to the evaluated gradients of an analysis, which has applications far beyond aerodynamics. Nevertheless, because AeroBuildup's implementation grants such an unusually large advantage to vectorized evaluation, simple finite-differencing is the default strategy used for stability derivative computation.

In addition to stability derivatives, other quantities of interest, like the location of the aircraft's neutral point, are computed as well. For Firefly at the Mach 0.15, sea level flight condition, AeroBuildup computes a neutral point located at 49% of the main wing's root chord, or $x_{\text{np}} = 241$ mm aft of a datum placed at the vehicle's nose. For comparison, Gaubatz [12] assessed this same geometry using a vortex-lattice code (AVL [148]), which produced an estimate for the neutral point that is slightly farther aft[12], at $x_{\text{np}} = 243$ mm. Gaubatz's subsequent wind tunnel test campaign produced results[13] that would indicate a neutral point at $x_{\text{np}} = 236$ mm.

In any case, this indicates that the AeroBuildup result is quite consistent with both experiment and other conceptual-level aerodynamics tools. This is a particularly useful comparison, as proper computation of this neutral point reflects the fact that AeroBuildup's potential-flow-based fuselage aerodynamic computations include the destabilizing contributions of the large fuselage, which is not captured by wing aerodynamics alone.

[12]Note that in Gaubatz [12], the reference datum is placed at the vehicle's tail rather than nose, so measurements must be subtracted from the vehicle length of 460 mm to convert coordinates to aircraft geometry axes with a nose datum. Note that the vehicle length of 460 mm is shorter than the bounding box of 480 mm, due to the inclusion of an external rocket igniter (briefly shown in Figure 4-2).

[13]See stability derivatives reported in Table 3.2 in Gaubatz [12].

High-Angle-of-Attack Flight

Because AeroBuildup uses sectional data that models post-stall behavior (obtained with a tool called NeuralFoil, described in Chapter 7), it is capable of estimating aerodynamic coefficients at extreme angles of attack and sideslip angles. This is a particularly useful capability when designing the Firefly vehicle, which must passively stabilize from a tumbling condition after deployment from the host vehicle.

For example, Figure 5-10 gives a "3D" version of the traditional aerodynamic polars from Figure 5-9, where these are extended to include sideslip angle β as well as angle of attack α, both out to $[-90^\circ, +90^\circ]$. In this study, the MIT Firefly air vehicle is analyzed at sea-level Mach 0.15 conditions.

Of particular interest is the "moment flow" diagram in the lower left of Figure 5-10, which shows how the yawing and pitching moment coefficients (C_n and C_m) vary throughout the α, β phase space; arrows and streamlines indicate the direction that passive aerodynamic moments will tend to point the nose, from any initial vehicle orientation. This mapping is similar to a "phase portrait" in dynamical systems theory, though inertial effects are neglected in this strictly-static analysis. Nevertheless, this provides an excellent initial map of the vehicle's passive stability characteristics from upset flight conditions, even those well-past stall. For example, the moment flow diagram indicates that Firefly is passively stable roughly in the regime where $\beta \in [-30^\circ, +30^\circ]$ and for all α, but not outside of this. This may indicate that active control inputs are required to recover from such upset conditions.

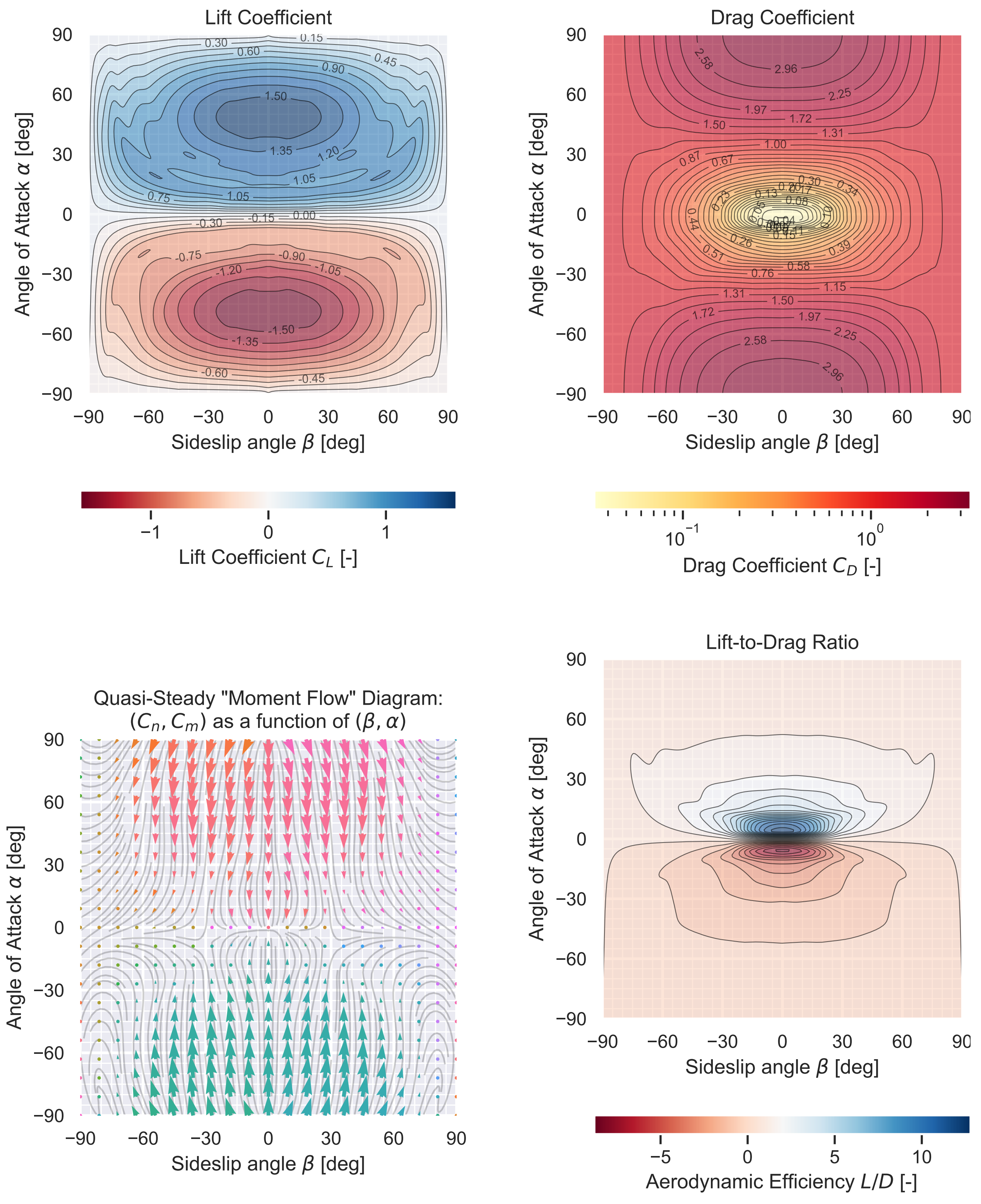


Figure 5-10: A "3D" version of standard aerodynamic polars, where lift, drag, moment (i.e., C_n and C_m), and aerodynamic efficiency are given as a function of *both* angle of attack α and sideslip angle β. Analysis is for the MIT Firefly air vehicle at sea-level Mach 0.15 conditions, with α and β varied. Aerodynamics computed with the AeroBuildup physics model, and given for an extreme range of $\alpha, \beta \in [-90^\circ, +90^\circ]$ to demonstrate post-stall modeling capability. Control surfaces are undeflected in all cases.

This kind of information about post-stall handling qualities of a candidate airframe would ordinarily not be available until well beyond the conceptual design phase. However, AeroBuildup's rapid aerodynamic analysis capability allows for this kind of analysis to be performed in a matter of seconds, enabling the designer to make informed decisions about the vehicle's passive stability characteristics early in the design process.

Transonic and Low-Reynolds Flight

Firefly uniquely operates at both low Reynolds numbers and transonic speeds, which are two flight regimes that are traditionally difficult to model with aerodynamics tools. Moreover, the *combination* of these two regimes presents further challenges, as a) limited validation data is available for such conditions, and b) these two conditions create aerodynamic design drivers that are directly opposed to each other, such as the target pressure distribution in pressure recovery zones. Further details on this inherent conflict of transonic and low-Reynolds design drivers are given by Drela [192]. In short, this conflict makes it challenging for a designer to know which high-level aerodynamic effects will dominate the vehicle's performance.

AeroBuildup is capable of modeling these effects simultaneously, as shown in Figure 5-11. Here, the MIT Firefly air vehicle is analyzed across a range of Mach numbers M_∞ and angles of attack α. In all cases, atmospheric conditions are taken at Firefly's launch altitude of 30,000 ft. Because of this, Reynolds number also changes as a function of Mach number. In these results, several notable aerodynamic phenomena are captured:

1. Near the zero-speed condition ($M_\infty \leq 0.05$), the Reynolds number also drops quite low, which precipitates a rise in the drag coefficient C_D and a sudden reduction in the lift-curve slope $C_{L\alpha}$ as the suction-side boundary layer cannot transition.

2. At low subsonic conditions, the expected Prandtl-Glauert-like compressibility effects on the overall pressure distribution are visible, which cause the lift curve slope $C_{L\alpha}$ to rise with increasing Mach.
3. At transonic speeds ($M_\infty \approx 0.8$), the onset of transonic effects is clearly visible. The Mach number at which these effects begin, M_{crit}, is a strong function of local pressure drop, and hence, angle of attack. This cross-dependency is successfully captured by AeroBuildup. After such speeds, lift drops markedly due to a buffet model, and drag rises due to the onset of wave drag. This shock effect occurs both at positive and negative angles of attack, but the wing's ability to sustain a larger pressure drop on the suction-side surface causes these transonic effects to be more pronounced at positive angles of attack.

This provides a unique view into this combination of nuanced aerodynamic effects that is rare to obtain at the conceptual phase. Because the impact of both transonic and low-Reynolds effects can be considered together, the designer can make a meaningful prediction about which set of competing aerodynamic design drivers is likely to dominate the vehicle's performance.

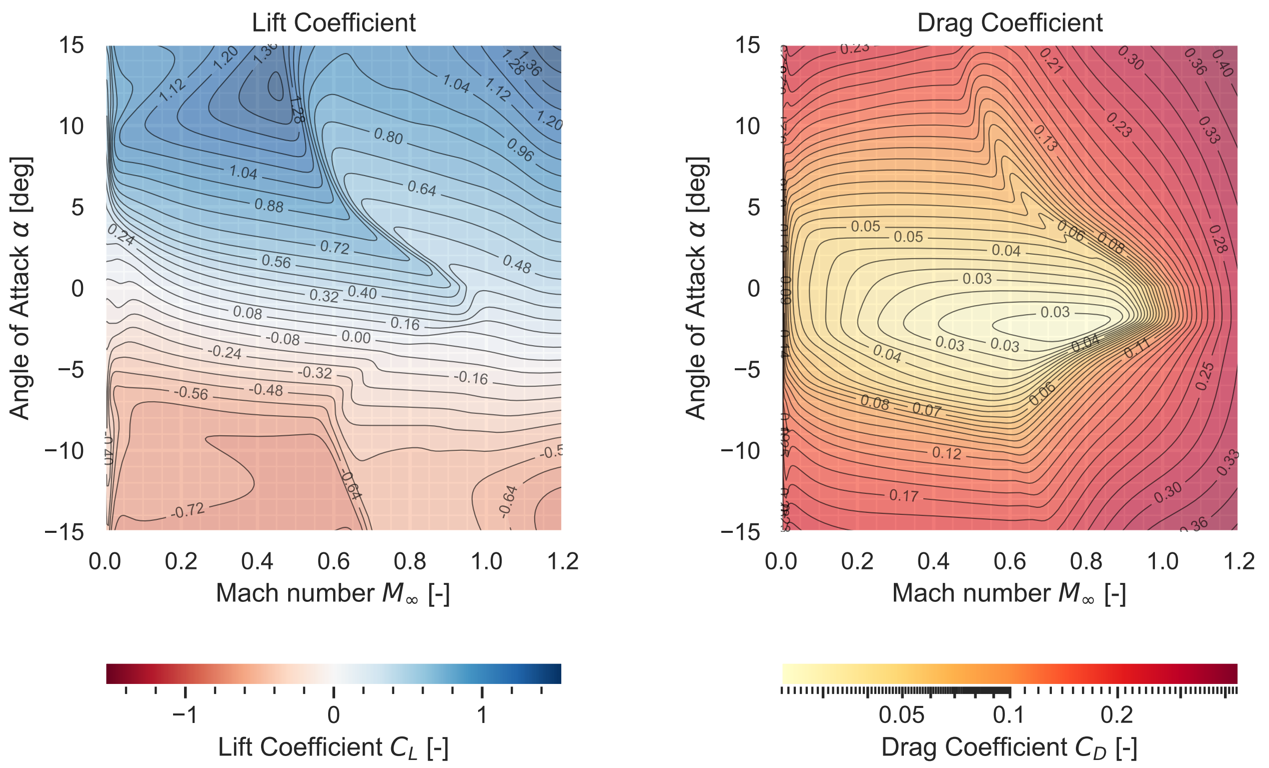


Figure 5-11: Transonic and low-Reynolds-number aerodynamic effects can be simultaneously captured by AeroBuildup. Here, the MIT Firefly air vehicle is analyzed across a range of Mach numbers M_∞ and angles of attack α. In all cases, atmospheric conditions are taken at Firefly's launch altitude of 30,000 ft. Because of this, Reynolds number also indirectly changes as a function of Mach number.

5.4 Cross-Validation of AeroSandbox Aerodynamics Analyses

To assess the accuracy of the three AeroSandbox aerodynamics analyses presented in Sections 5.1, 5.2, and 5.3, a series of cross-validation studies were performed.

5.4.1 Flying Wing Validation Study

The first such validation study is based on a flying wing geometry described in NACA Report RM-A50K27 [15], which provides wind tunnel data as a point of comparison. This AeroSandbox validation case study was originally organized by John Yost [193], and was subsequently refined in collaboration with the author.

The geometry used in this study is described fully by Tinling and Kolk [15], and is also visualized in Figure 5-12. The wing has a 35° quarter-chord sweep, a span of $b = 3.098$ m, an aspect ratio of $\mathcal{R} = 10$, and a taper ratio of $\lambda = 0.5$. A constant NACA 64_1A012 airfoil is used along the entire span, with no twist. The wing is analyzed at a Reynolds number of $\text{Re}_c = 2 \times 10^6$ to match the wind tunnel conditions, corresponding to 91.3 m/s ($M_\infty = 0.26$) at sea level.

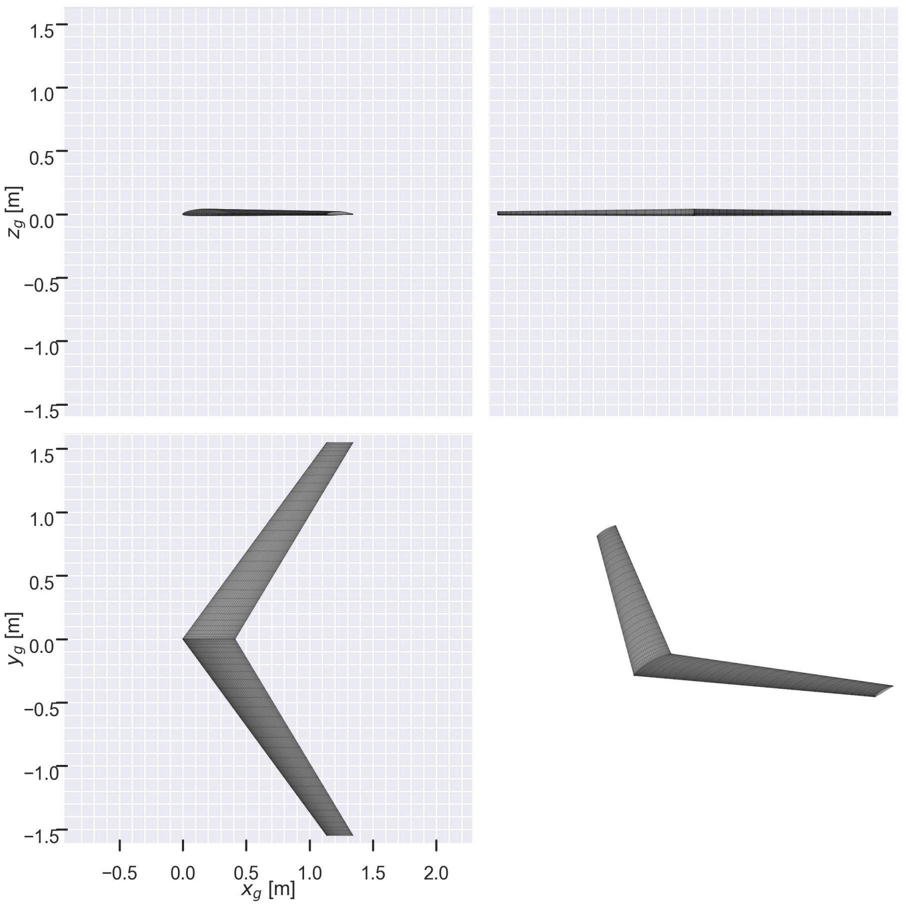


Figure 5-12: Three-view of the flying wing aircraft geometry used in the AeroSandbox aerodynamics validation study. Based on geometry used in wind tunnel experiements in NACA Report RM-A50K27 [15].

Using this geometry and freestream conditions, we perform an angle of attack sweep using the three AeroSandbox aerodynamics analyses, to compare to wind tunnel experiment. In addition, several external conceptual-level aerodynamics tools are used for comparison, including:

- AVL [148] (using XFoil [95] for 2D sectional data), a vortex lattice method that is widely used in conceptual aircraft design.
- OpenVSP's VSPAero solver [94], which is a 3D panel method combined with form-factor-based empirical profile drag buildups.
- LORAAX [194], a fully-coupled 3D panel and 2D (strip-wise) integral boundary layer method. An iterative solution is used to obtain a force-free wake.

AVL [148] (using XFoil [95] for 2D sectional data), OpenVSP's VSPAero conceptual aerodynamics tool [94], and LORAAX, a 3D panel and integral boundary layer code. The results of this comparison are shown in Figure 5-13.

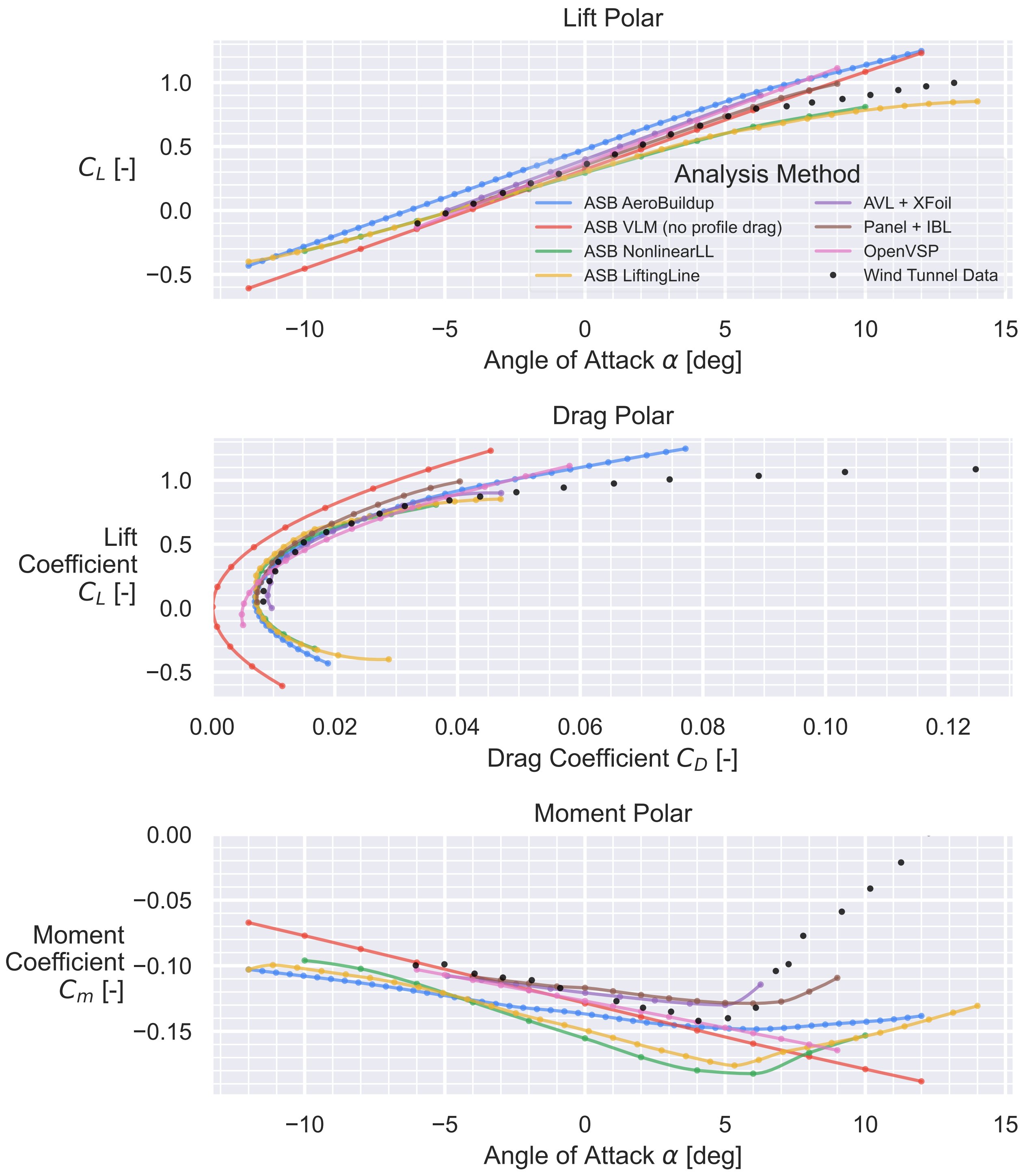


Figure 5-13: Comparison of aerodynamic polars for the flying wing geometry at $\text{Re}_c = 2 \times 10^6$. The AeroSandbox VLM, lifting-line, and AeroBuildup analyses are compared to wind tunnel data and other conceptual-level aerodynamics tools. The wind tunnel data is reproduced from NACA Report RM-A50K27 [15].

In the lift polar of Figure 5-13, we observe that the AeroSandbox lifting line models (both quasi-linear and fully nonlinear) capture the nonlinear $C_L(\alpha)$ dependency of the wind tunnel data quite well. Likewise, the AeroBuildup method captures the shape of the lift curve reasonably well. Interestingly, the computational lifting-line methods tend to err low on this highly-swept-wing case while the AeroBuildup method tends to err high, and for exactly opposite reasons. To explain further, consider that although the true lifting line of a highly-swept wing will tend to follow the geometric sweep of the wing, it will tend to unsweep near the centerline due to the influence of the opposite wing half. A computational lifting-line method places the aerodynamic lifting line exactly at the locus of geometric quarter-chord-points, which then underestimates the lift at the centerline. (While there *should* be a dip in the lift distribution at the centerline, a computational lifting-line method will predict a more-extreme dip than reality.) On the other hand, AeroBuildup assumes a uniform downwash[14], which results in no dip in the lift distribution near the centerline. Of course, reality is somewhere in between these two limit cases. The ASB vortex lattice method estimates the lift quite well when flow is fully attached, but as is the case with any (linear) vortex lattice method, it struggles to capture the nonlinearities in the lift curve slope near stall.

In the drag polar of Figure 5-13, the AeroSandbox lifting line models and AeroBuildup both achieve excellent agreement with the wind tunnel data. This is notable, because this polar is typically the most useful metric of the overall aerodynamic performance of the aircraft. The AeroSandbox vortex lattice method does not include any profile drag handling, and is thus not expected to match other results – as discussed in Section 5.1, the primary intended use of the VLM is for flight dynamics analysis rather than performance prediction.

In the moment polar of Figure 5-13, we see that the AeroSandbox vortex lattice method achieves very close agreement with wind tunnel data until separation occurs. This is expected, as vortex lattice methods

[14]This is usually a reasonable rough assumption for low-induced-drag wings with near-elliptical lift distributions, but in a highly-swept wing case this is violated.

have a long and successful history of use for conceptual stability and control analysis. The AeroSandbox lifting line methods achieve good agreement on the static stability derivative $C_{m\alpha}$, but a small constant error in the C_m value itself remains. This is also likely due to underestimation of the lift at the centerline, as discussed in the lift polar. The AeroBuildup method achieves closer agreement to wind tunnel data, but it predicts a more gradual onset of stall than the sudden separation seen in experiment.

One of the notable findings of this validation study is that the quasi-linear lifting line and the fully-nonlinear lifting line yield very similar results across lift, drag, and moment polars. Because of this, the quasi-linear lifting line is recommended for use in conceptual design, as it is much faster to compute. The fully-nonlinear lifting line may be useful in specific cases with extensive and important near-stall behavior.

Nevertheless, in all cases here, the discrepancies discussed are well below the threshold of accuracy needed for conceptual aircraft design purposes. (Mostly, the deviations discussed in detail here are presented for theoretical interest.) In particular, the accurate results for the C_D-C_L polar highlight the utility of these methods for initial aircraft performance prediction.

5.4.2 Sailplane Cross-Validation Study

The second test included here is a cross-validation study of the AeroSandbox aerodynamics analyses on a more complex geometry that includes multiple wings (including a non-planar wing) and a fuselage. This study analyzes aerodynamic performance of a small remote-control-scale sailplane, which is drawn in Figure 5-14. The main wing has a span $b = 2$ m, a projected area of $S = 0.292$ m^2, and polyhedral with outer sections angled up $11.3°$. The horizontal stabilizer is mounted with a geometric decalage of $5°$ with respect to the main wing, which keeps the vehicle in trim at zero control surface deflections. The fuselage geometry is a canted body of revolution, and is somewhat thicker than a typical sailplane to test whether fuselage $C_{m\alpha}$ and $C_{n\beta}$ effects are well-captured.

Performance is assessed at sea-level conditions with a freestream velocity of 15 m/s, which corresponds to a relatively low Reynolds number of $\text{Re}_c = 155 \times 10^3$ referenced to the mean aerodynamic chord of the main wing.

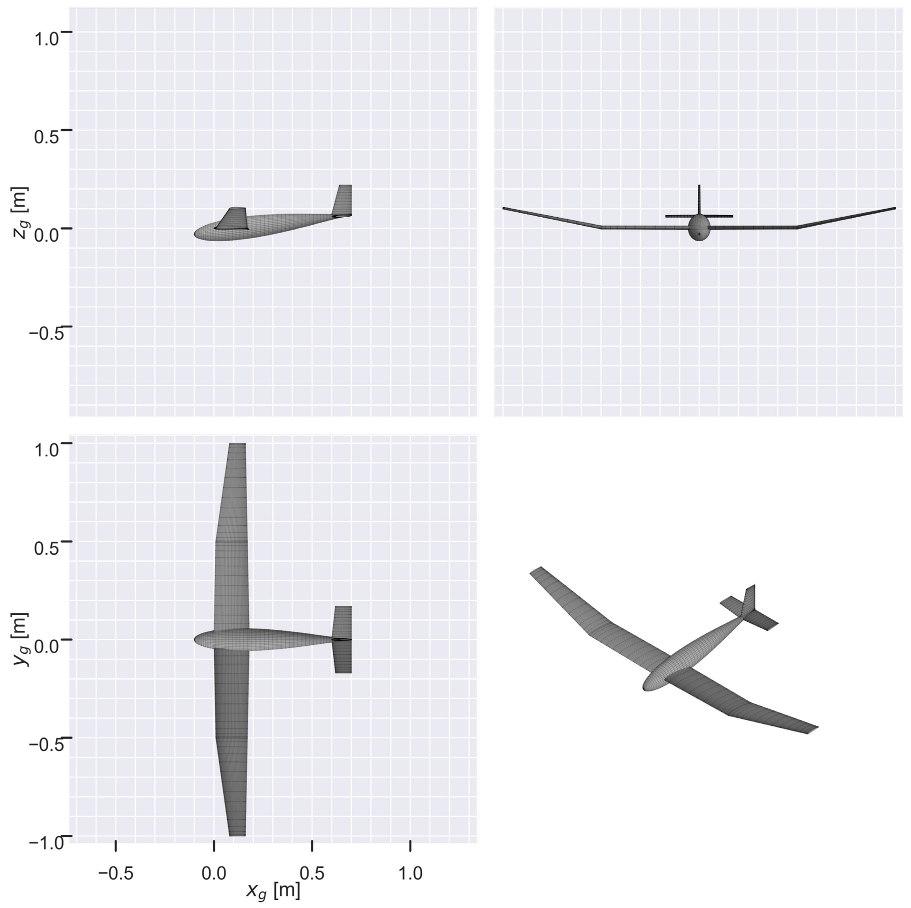


Figure 5-14: Three-view of the glider aircraft geometry used in the AeroSandbox aerodynamics cross-validation study.

Figure 5-15 shows the aerodynamic polars for the glider geometry, computed using the AeroSandbox VLM, lifting-line, and AeroBuildup analyses. In general, trends are relatively similar to the flying wing case. On lift, good agreement is achieved between all three models. As before, the lifting line and AeroBuildup models able to capture nonlinear lift curve effects that the VLM misses.

On the drag polar, the VLM model once again only reports induced drag (as no profile drag model is included), and thus is not expected to match the other models. The lifting line and AeroBuildup models both agree quite closely on their drag predictions across a range of lift coefficients.

On moment prediction, the lifting line and vortex lattice methods agree reasonably well within the range of angles of attacks that have mostly attached flow ($\alpha \in [-5^\circ, +12^\circ]$). The AeroBuildup method generally predicts a more-negative $C_{m\alpha}$ derivative; this is likely erroneous, and occurs due to the method's inability to capture the downwash effects of the main wing on the horizontal stabilizer. This is a known limitation of the AeroBuildup method, and is a tradeoff of speed vs. accuracy that could be made in future iterations of the method.

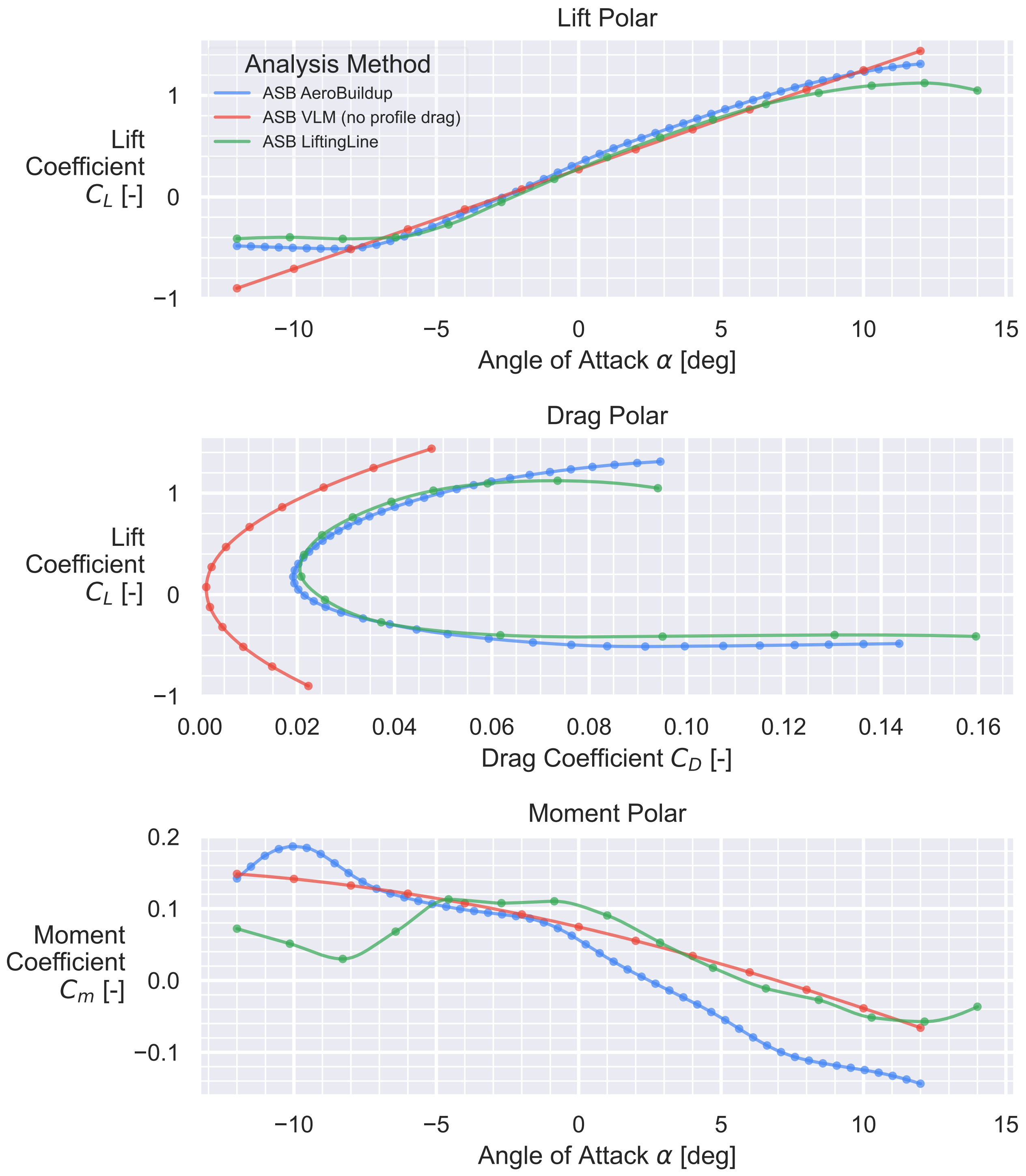


Figure 5-15: Comparison of aerodynamic polars for the glider geometry of Figure 5-14. Geometries are assessed at flight conditions corresponding to $\text{Re}_c = 155 \times 10^3$. The AeroSandbox VLM, lifting-line, and AeroBuildup analyses are cross-compared to assess agreement.

In general, the validation studies in this section demonstrate that the AeroSandbox aerodynamics analyses generally agree with wind tunnel data, external analysis tools, and each other. These studies also indicate that different methods have different strengths and weaknesses: the vortex lattice method is useful for moment and stability derivative estimation, but AeroBuildup and lifting line methods have an advantage in force prediction and performance analysis.

5.4.3 Computational Reproducibility

All code and aircraft geometries used in these aerodynamic validation studies are publicly available within the tutorials section of the broader AeroSandbox repository, available at `https://github.com/peterdsharpe/AeroSandbox/tree/master/tutorial`.

5.5 Rigid-Body Equations of Motion

Code-transformations-based MDO framework are particularly well-suited for solving combined vehicle-and-trajectory optimization problems, as demonstrated with several of the aircraft design case studies of Chapter 4. Fundamentally, this is because the trajectory side of the problem (assuming transcription using direct collocation) forms a large-and-sparse constraint Jacobian, while the vehicle design part of the problem generally forms a small-and-dense constraint Jacobian. A framework that facilitates automatic sparsity detection can naturally deal with this varying sparsity structure without user intervention, which saves significant engineering time.

Implementation fundamentally involves three steps: a) parameterizing the system into a state vector, b) defining the equations of motion, and b) transcribing the problem into an optimization framework (i.e., discretization). All three steps can be quite tedious, and are often error-prone (and difficult to debug) due to their complexity. For example, consider the example equations of motion in Equations 5.4-

5.7; a single sign error will result in silently-incorrect results. Implementing this trajectory optimization problem manually may be necessary or desirable on problems with unique dynamics (e.g., contact dynamics during landing gear touchdown), but for many aerospace problems, much of the code is boilerplate.

Because of this, this thesis implements a general-purpose flight dynamics stack that gives the user the option to bypass this manual process. One challenge of such a general-purpose effort is that different aircraft design problems will require different levels of fidelity in the equations of motion. To handle this, the AeroSandbox dynamics stack uses a series of classes with nested inheritance, starting from the highest-allowed fidelity model, which are the 3D rigid-body equations of motion given in Equations 5.4 to 5.7. This approach allows the user to select the level of fidelity that is appropriate for their problem, while minimizing the size and maintenance cost of this codebase through code reuse. Another benefit is that all classes share the same API, which enables the user to easily swap between different levels of fidelity.

Example: General Newtonian Rigid-Body Equations of Motion
For a rigid 3D object with:

- Mass m and body-axis inertia tensor **I**, in a gravity field with acceleration g in the $+\hat{z}_e$ (Earth-frame) direction
- Position $[x_e, y_e, z_e]$ in the Earth frame
- Velocity $[u, v, w]$ in the body frame
- Orientation given by roll-pitch-yaw Euler angles $[\phi, \theta, \psi]$
- Angular velocity $[p, q, r]$ in the body frame
- Forces $[X, Y, Z]$ and moments $[L, M, N]$ in the body frame

Motion is governed by the following 12 coupled first-order differential equations, adapted from Drela [154]:

$$\begin{aligned}
\dot{x}_e &= (\cos\theta\cos\psi)u + (\sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi)v + (\cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi)w \\
\dot{y}_e &= (\cos\theta\sin\psi)u + (\sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi)v + (\cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi)w \\
\dot{z}_e &= (-\sin\theta)u + (\sin\phi\cos\theta)v + (\cos\phi\cos\theta)w
\end{aligned} \tag{5.4}$$

$$\begin{aligned}
\dot{\phi} &= p + q\sin\phi\tan\theta + r\cos\phi\tan\theta \\
\dot{\theta} &= q\cos\phi - r\sin\phi \\
\dot{\psi} &= \frac{q\sin\phi}{\cos\theta} + \frac{r\cos\phi}{\cos\theta}
\end{aligned} \tag{5.5}$$

$$\begin{aligned}
X &= m(\dot{u} + qw - rv + g\sin\theta) \\
Y &= m(\dot{v} + ru - pw - g\sin\phi\cos\theta) \\
Z &= m(\dot{w} + pv - qu - g\cos\phi\cos\theta)
\end{aligned} \tag{5.6}$$

$$\begin{aligned}
L &= I_{xx}\dot{p} + I_{xy}\dot{q} + I_{xz}\dot{r} + (I_{zz} - I_{yy})qr + I_{yz}(q^2 - r^2) + I_{xz}pq - I_{xy}pr \\
M &= I_{xy}\dot{p} + I_{yy}\dot{q} + I_{yz}\dot{r} + (I_{xx} - I_{zz})rp + I_{xz}(r^2 - p^2) + I_{xy}qr - I_{yz}qp \\
N &= I_{xz}\dot{p} + I_{yz}\dot{q} + I_{zz}\dot{r} + (I_{zz} - I_{xx})pq + I_{xy}(p^2 - q^2) + I_{yz}rp - I_{xz}rq
\end{aligned} \tag{5.7}$$

From the starting point of Equations 5.4-5.7, the AeroSandbox dynamics stack offers users a series of simplifying assumptions that can be applied to reduce the complexity of the equations of motion. A wide variety of possible assumptions are offered, which can generally be grouped as follows:

- **Dimensionality and symmetry**: Starting from the full 3D equations of motion, users can reduce to 2D or 1D equations of motion. Even a reduction to two dimensions (e.g., a range-altitude position parameterization, and assuming left-right vehicle symmetry) offers an enormous reduction in complexity, since this eliminates almost all of the gyroscopic, centrifugal, and Coriolis terms.

- **State parameterization**: Depending on what the "fastest relevant flight dynamics mode" is for the problem, the state vector can be parameterized in different ways. For example:

    - With no simplifications beyond equations 5.4-5.7, the vehicle's state vector has 12 elements (position and velocity, each in both a translational and rotational sense; each with 3 components). Here, the instantaneous control inputs to the system are forces and moments. This is referred to as an *inertial* parameterization, where full motion (e.g., tumbling) can be captured.

- The first possible simplification is to make a *point-mass* assumption, where the rotational dynamics (Equations 5.5 and 5.7) are neglected. Instead, now the instantaneous control inputs are orientation: possibly the Euler angles $[\phi, \theta, \psi]$, but in a fixed-wing aircraft context this would more likely be parameterized using the aerodynamic orientation angles $[\alpha, \beta]$. This essentially assumes that vehicle can be "pointed" in any desired direction nearly-instantaneously, since this effect (analogous to a short-period mode in aircraft flight dynamics) is much faster than the dynamics of optimization interest[15]. For example, this is the parameterization used in the Firefly rocket-UAV study of Section 4.2, where the performance dynamics of interest (e.g., the boost-glide trajectory of Figure 4-7) are much slower than the vehicle's orientation dynamics (e.g., the vehicle's short-period and Dutch roll modes, both of which are on the order of 10 Hz).
- A second further simplification is to also eliminate the translational velocity dynamics (Equation 5.6). Now, the instantaneous control inputs include not only orientation, but also the velocity itself. This essentially assumes that the vehicle can be flown at any desired velocity, and that the dynamics of this velocity change are much faster than the dynamics of optimization interest. Stated another way, this assumes that transient changes in kinetic energy are so small that they can be neglected. For example, this is the parameterization used in the Dawn solar aircraft study of Section 4.3, where the major dynamics of interest

[15]This is essentially a spectral separation assumption.

(e.g., solar flux as it varies over 24 hours) are much slower than the transient dynamics of the vehicle's velocity (e.g., phugoid mode).

- **Velocity parameterization**: In all 2D and 3D cases, the velocity vector can be parameterized in either Cartesian coordinates or spherical coordinates[16]. (In spherical coordinates, velocity is parameterized by a magnitude V; a flight path angle γ; and a bearing angle, sometimes called *track*.) The underlying physics are of course identical, but both parameterizations have unique advantages in a discretized setting:
  - Cartesian velocity parameterization yields simpler equations of motion, and it eliminates a singularity at $V = 0$; this makes it useful for optimizing aerobatic maneuvers.
  - Spherical velocity parameterization, on the other hand, offers greatly reduced discretization error due to the ability to use symplectic (energy-preserving) integrators. Here, the velocity magnitude maps directly on to the kinetic energy, so exact energy conservation can be enforced at the parameterization level. This is particularly important for problems involving high-L/D airplanes, as the even a miniscule error in the direction of the lift force while flying a curved trajectory will cause it to perform nonzero work, leading to noticeable energy conservation error. The inherent energy conservation of spherical velocity parameterization makes it superior for performance optimization problems, since the (adversarial) optimizer is less able to exploit discretization error to find a solution that is not physically realizable. For this reason, both the Firefly and Dawn studies use spherical velocity parameterization.

Choosing an appropriate parameterization can easily change the wall-clock solve time of a trajectory optimization problem by an order of magnitude. For example, if the Dawn solar aircraft design

[16]In 2D cases, this corresponds to polar coordinates.

optimization problem were to be parameterized using the full 12-element state vector, the dynamics would span many orders of magnitude in time scale – with a short-period mode on the order of 1 Hz, and solar dynamics on the order of 24 hours. This situation corresponds mathematically to a stiff ODE system, requiring extremely fine time resolution to remain numerically stable. By choosing a parameterization that eliminates the fast dynamics (and offering a shared API that makes such simplification straightforward), the optimization problem becomes much faster to solve and has better convergence properties. In total, the AeroSandbox dynamics stack offers 8 different "off-the-shelf" classes for the user to parameterized vehicle dynamics, each with different combinations of choices from the list of possible assumptions above; this allows the user to identify the appropriate level of fidelity for this problem.

In addition to parameterization, transcription of the dynamics into an optimization formulation is another common user pain point that the AeroSandbox dynamics stack aims to address. Both direct collocation[17] and single-shooting transcription methods are supported. Empirically, direct collocation methods tend to result in optimization problems that not only solve faster but also have more stable convergence, so in general these are preferred. However, single-shooting methods enable the use of adaptive-step integrators, which can be useful in problems where stiff dynamics cannot be avoided (e.g., chemical species kinetics in a rocket engine plume, as demonstrated by Mathesius [152]).

Under either a direct collocation or single-shooting approach, a variety of integrators are available to the user. These include Runge-Kutta methods, forward- and backward-Euler methods, a trapezoidal method, methods based on Simpson's rule, and a cubic reconstruction method. This wide gamut of integrators lets the user select an appropriate integrator, based on speed vs. accuracy and smoothness vs. discontinuity tradeoffs.

[17] Described in more detail by Kelly [153]

5.5.1 Trajectory Optimization Case Study: Low-Altitude Air Racing Optimization

While the *Firefly* and *Dawn* case studies of Chapter 4 demonstrate the usage of the AeroSandbox dynamics stack more broadly for combined vehicle-and-trajectory optimization, this section gives a more focused case study that demonstrates the deep capabilities of the dynamics stack developed in Section 5.5.

This aircraft trajectory optimization case study focuses on fast, low-altitude flight through mountainous terrain. Ultimately, this nap-of-the-earth flight is often used for evading radar detection while reaching an objective. This task, recently popularized in prior art by Cruise et al. [195], is a common and challenging scenario in military aviation.

The task in this case study is to fly a fighter jet between two points 75 km (40 nmi) apart, located near Riffe Lake, Washington, USA. The aircraft's aerodynamics and propulsion physics are loosely modeled after an F/A-18 in full afterburner, ultimately yielding in an average Mach number of 0.95 in the resulting trajectory. This scenario replicates a mission in Microsoft's recent Flight Simulator 2020, which allows results to be compared[18] against known fastest times.

[18] Flight physics will vary slightly between this case study and the Flight Simulator 2020 model, but broader strategic decisions like whether to fly over or around a mountain make for a meaningful comparison.

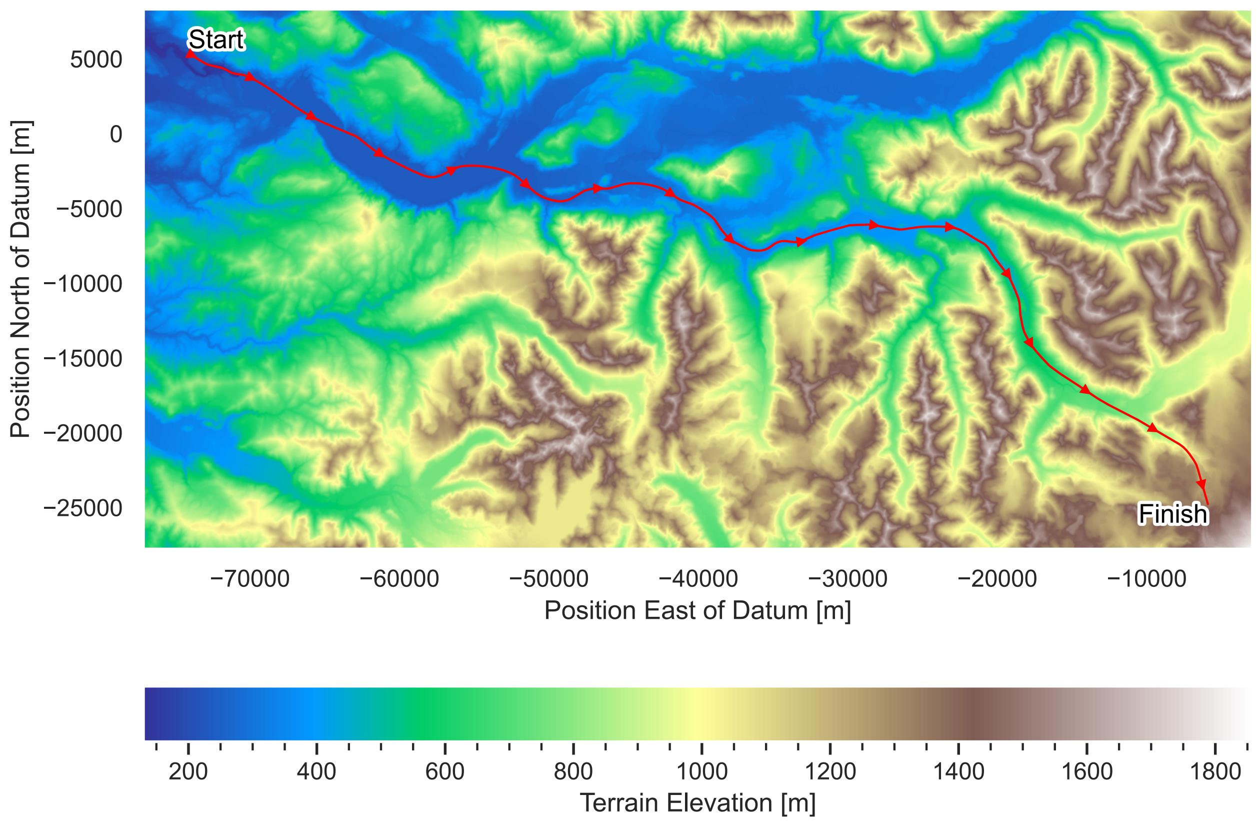


Figure 5-16: Top-down visualization of the optimized trajectory for the air racing optimization case study, which uses the AeroSandbox dynamics stack. Trajectory follows riverbeds and valleys closely to maintain low-altitude flight.

The objective function of this optimization problem is a combination of flying as low as possible and as fast as possible. A (slight) regularization penalty is also added for aggressive control inputs, which encourages more human-like piloting. Quantitatively, the objective function is given by:

$$\text{minimize} \left(\frac{\text{Total flight duration}}{(240\text{ seconds})}\right)^2 + \left(\frac{\text{Time-averaged altitude (MSL)}}{(30\text{ feet})}\right) + 10^{-3} \cdot w \tag{5.8}$$

where the wiggliness w is obtained by integrating the squared second-time-derivative of each control input[19], and summing them.

The vehicle dynamics are parameterized using the AeroSandbox dynamics stack; the selected parameterization is 3D, uses point-mass equations of motion, and uses a spherical velocity representation. The optimization problem is transcribed using direct collocation, and cubic spline reconstruction is used for integration. This reconstruction is discrete, creating a cubic local function representation of each integrand over each interval based on an overlapping 4-point stencil[20]; this reconstruction is then exactly integrated over each interval.

In addition to the constraints obtained via direct collocation of the vehicle dynamics, the following additional constraints are added to the optimization problem:

- Initial and final state constraints.

- A path constraint that the altitude must be at least 15 meters (49 feet) above the terrain elevation. This elevation is obtained via a digital elevation model (DEM) of the terrain produced by the U.S. Geological Survey at 1 arc-second (approx. 30 m) resolution [196]. To implement this in a way that is C^1-continuous with respect to collocation node location, the elevation model is represented as an interpolated 2D hermite spline using the AeroSandbox numerics stack [197].

- The track angle may not deviate from the overall goal direction (i.e., start-to-finish) by more than 75°. This prevents the optimizer from flying in circles in low-altitude regions, which can cause spurious local minima.

- The vehicle's G-loading must remain in the range $[-0.5\,\mathrm{G}, +9.0\,\mathrm{G}]$ at all times, due to both

[19]Control inputs include the angle of attack α, the bank angle ϕ, and the throttle setting. Inputs are nondimensionalized by reference values before integration into the w measure.

[20]This stencil uses the two endpoints of each interval, and one point beyond either side of the interval.

structural and human factors limitations[21].

This optimization problem is nonlinear and also highly nonconvex – a simple example of this nonconvexity is the multiple local minima that can occur if the optimizer flies to the left or right of a mountain. To mitigate this, the optimization problem is warm-started using an initial guess from a simple 2D-grid relaxation using an A* search algorithm[22].

The resulting trajectory is shown in a top view in Figure 5-16, and in 3D views in Figures 5-17 and 5-18. The trajectory is highly dynamic, with the vehicle navigating a series of quick sequential max-G turns in opposite directions to follow the terrain. The trajectory tends to follow low-altitude areas, like river beds and valleys, which shows the influence of the objective function. The mean altitude above ground level (AGL) throughout the optimized flight trajectory is just 18 meters (59 feet), which is remarkably close to the 15 meter minimum altitude constraint.

To achieve this low-altitude flight, the trajectory optimization algorithm naturally "discovers" a low-altitude "ridge-crossing" maneuver, which involves an inverted high-G turn. This is shown in Figure 5-18. This maneuver, which is also used in real-world military nap-of-the-earth flight, allows lower altitudes to be maintained by exploiting the much-higher positive G limit of the vehicle compared to its negative G limit. This is a good example of how the trajectory optimization algorithm can discover novel and unexpected solutions to complex problems, even without any explicit user guidance.

[21]One motivation for this rather-modest negative G limit is to test whether the optimizer can discover "ridge-crossing" maneuvers, as demonstrated in Figure 5-18.

[22]Another good simple choice here would be to initialize using a 3D Dubins path, as demonstrated by Vana et al. [198].

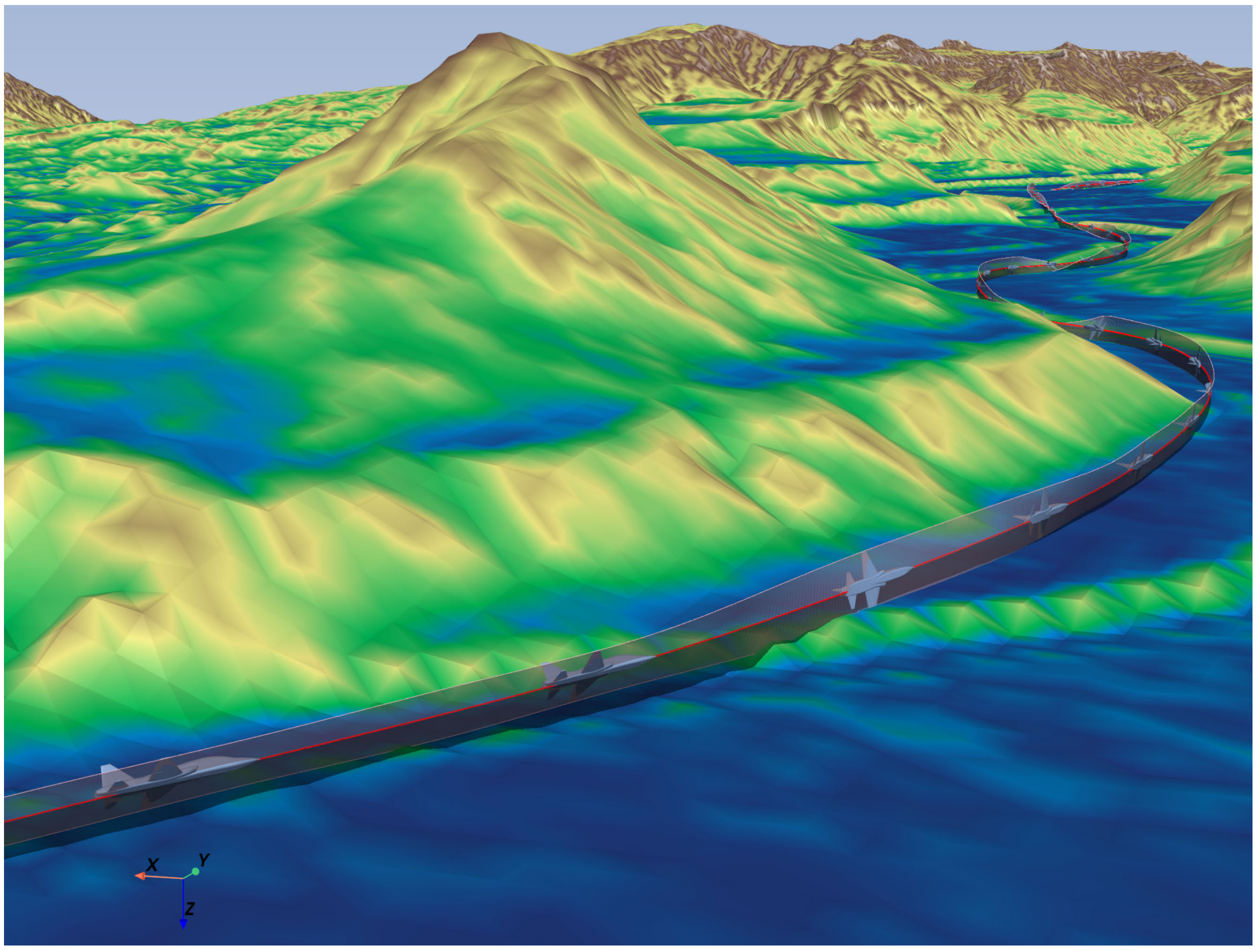


Figure 5-17: 3D visualization of the optimized trajectory for the air racing optimization case study, which uses the AeroSandbox dynamics stack for both numerical formulation and visualization. Here, the vehicle navigates a series of quick sequential max-G turns in opposite directions to follow the terrain. In this figure, the aircraft and wingtip ribbon are drawn at 70x scale for visibility; in some places, this scaling-up causes a visual artifact where the wingtips appear to barely intersect the ground.

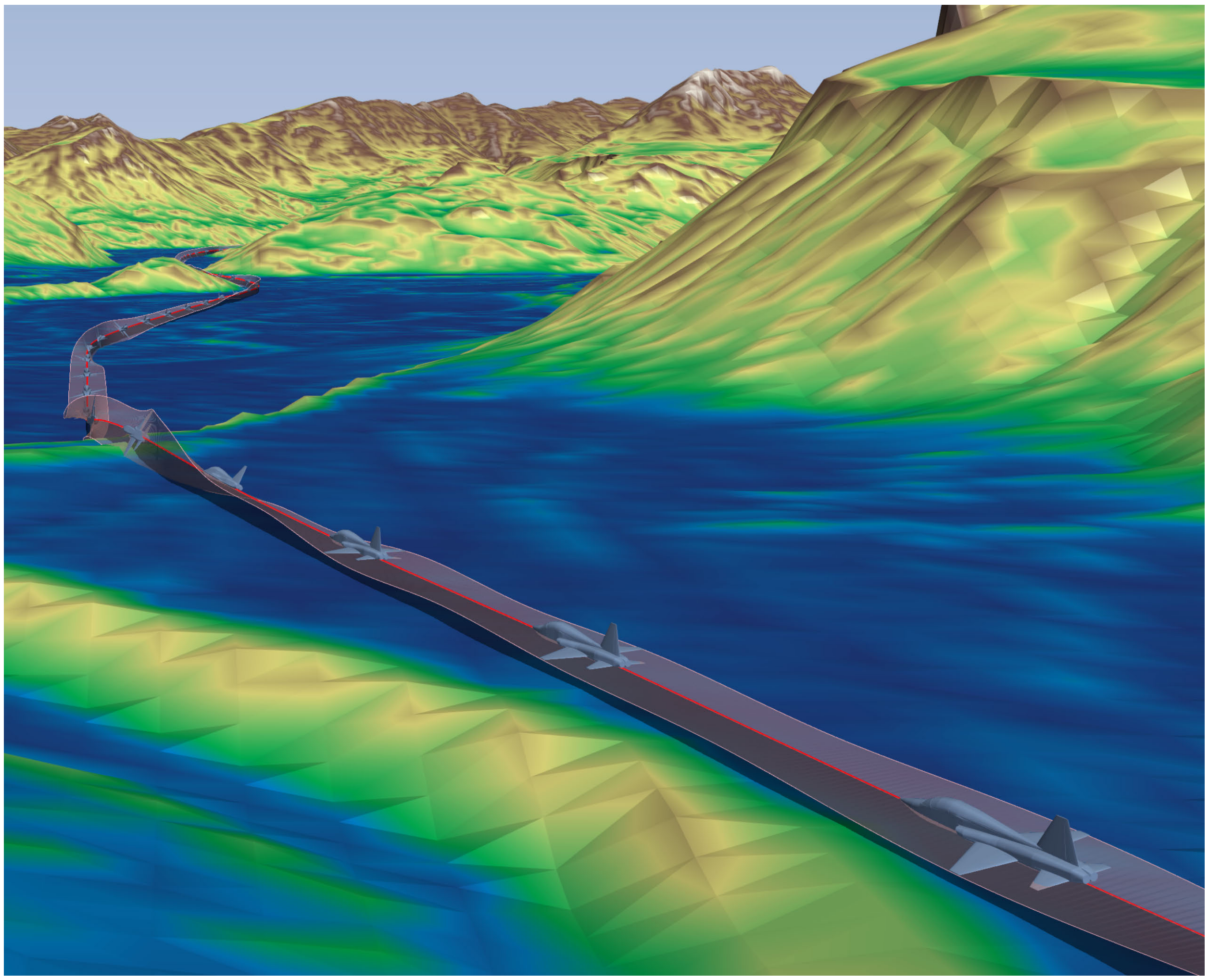


Figure 5-18: 3D visualization of the optimized trajectory for the air racing optimization case study, which uses the AeroSandbox dynamics stack for both numerical formulation and visualization. Here, we show that the trajectory optimization algorithm naturally "discovers" a low-altitude ridge-crossing maneuver, which involves an inverted high-G turn. This is a common maneuver practiced in military nap-of-the-earth flight. In this figure, the aircraft and wingtip ribbon are drawn at 70x scale for visibility; in some places, this scaling-up causes a visual artifact where the wingtips appear to barely intersect the ground.

In short, this case study demonstrates the deep capabilities of the AeroSandbox dynamics stack to model complex vehicle dynamics with challenging constraints. When combined with the more performance-oriented case studies in Chapter 4, these examples show how a code-transformations-based MDO framework facilitates flexible modeling up and down the fidelity ladder.

The code (including data, optimization problem formulation, and visualizations) needed to reproduce this case study is publicly available at `https://github.com/peterdsharpe/air-racing-optimization`. This source also includes links to video demonstrations of the optimized trajectory for further inspection.

Chapter 6

Sparsity Tracing via NaN-Propagation

A key challenge in applying code transformations is handling external black-box (i.e., non-traceable) function calls within the analysis graph. While code transformations can apply techniques like automatic differentiation and automated sparsity detection to accelerate traceable code, these efficiency gains are lost when opaque black-box functions are present. Here, gradient calculation methods must invariably fall back to slower techniques, such as finite-differencing or complex-step derivative computation [133]. The end result is that patching a black-box model into a code-transformations-based MDO tool using simple finite-differencing usually leads to poor performance.

To take a first step towards improving black-box handling in code transformation frameworks, we can ask ourselves what the minimum amount of information needed to achieve meaningful acceleration on black-boxes would be. In an ideal case, we would like to have a fast way to directly evaluate the Jacobian of the black-box function at some given point in the input space. This would allow the black-box function to be directly patched into the computational graph as its own primitive operator. Unfortunately, though some black-box functions provide direct gradient information, most do not.

If we accept that the Jacobian of a black-box function will need to be computed with finite differencing to support gradient-based optimization, the next-best option would be to obtain information that allows us to compute this faster. One example of such information is the sparsity pattern of the Jacobian of the black-box function. This sparsity pattern essentially gives a map of which inputs have the potential to affect each output. Armed with this information, we can perform Jacobian compression via coloring [2, 71, 105]. This allows the Jacobian to be calculated with fewer function evaluations, even in the case of simple finite-differencing.

6.1 Existing Approaches

Currently, the most common approach to obtain the sparsity pattern of a black-box function is a *gradient-based* method. First, the (dense) Jacobian is constructed via finite differencing, evaluated at some particular point in the input space that is assumed to be representative. (In the context of an optimization framework, this input point usually corresponds to the initial guess.) Then, a critical assumption is made: it is assumed that any zero entries in this evaluated Jacobian are indicative of a zero entry in the true Jacobian – in other words, sparsity seen at one point indicates sparsity at all points. Due to this assumption, the sparsity pattern constructed with this method is only an estimate, not a guarantee.

The fact that this reconstructed sparsity pattern is only an estimate is inevitable for black-box functions. To illustrate why, consider a piecewise function, where input x_j influences output f_i when x_j is within a certain interval but not outside of it. It is possible to construct a pathological black-box function where this region of influence is arbitrarily small, such that it will nearly never be detected by direct sampling of the Jacobian. Such cases will result in a *false negative* in the estimated sparsity pattern: cases where x_j and f_i are believed to be independent, when in reality they are not.

This kind of branching code execution is perhaps the most common cause of such false negatives (often via conditional expressions) in the context of engineering analysis. However, there are other relevant failure modes as well. One possible cause of false negatives is if the representative input point yields a partial derivative of zero only due to coincidence, rather than due to mathematical structure. One simple illustrative example is the function $f(x) = x^2$, if the evaluation point of $x = 0$ is chosen. Here, a central finite difference will see zero gradient and incorrectly infer no dependency. On most functions of practical interest, such points are usually rare within the input space[1]. However, in an optimization context, the distribution of user-specified initial guesses is not uniform, and in fact adversarial: users will preferentially supply near-optimal initial guesses, which have near-zero gradients (if the problem is unconstrained). Because of this, and as demonstrated later in Section 6.6, this effect can contribute to a non-negligible rate of false negatives in practice.

False positives, where x_j and y_i are believed to be related when they are not, are also possible, though rarer. A practical example where this could occur is with a black-box function that has truncation error,

[1]In most practical functions $\vec{f}(\vec{x}) : \mathbb{R}^m \to \mathbb{R}^n$, the dimensionality of all manifolds where $\partial f_i / \partial x_j = 0$ is less than the dimensionality of the input $\vec{x}$. In such cases, *almost all* points will be free of coincidental zeros.

perhaps due to a nonlinear solver or numerical integrator that uses a value-dependent convergence criteria. In such cases, changing one input may alter the number of iterations required for convergence. This can cause changes in the truncation error of all outputs—even those that are mathematically-unrelated to that input. Thus, changes in this truncation error (i.e., a differing number of iterations) may yield a nonzero Jacobian entry, even if the math that the code aims to represent has no such dependency. Other false positive scenarios, such as stochastic functions, are possible but relatively uncommon in engineering analysis. In either the false-negative or false-positive case, an interesting observation is that the sparsity of the function *in code* may not match the sparsity of the mathematical model the code aims to represent.

For reasons related to Jacobian compression (described later via demonstration in Section 6.2.3), the harm caused by a false-negative and a false-positive are very unequal – false negatives are far worse. The ultimate effect of a false-positive is that subsequent gradient calculations will be slightly slower, though still numerically correct. On the other hand, the ultimate effect of a false-negative is that gradient calculations can be incorrect, due to erroneous Jacobian compression. Worse yet, these incorrect results can occur silently. In the context of design optimization, an incorrect gradient might only be detected much later in the development effort, when an optimizer consistently cannot converge (as the KKT conditions cannot be satisfied). By this point in the code development process, the combined optimization model is usually much more complex, and finding and eliminating an incorrect-gradient error becomes enormously tedious. Because of this, a new method for black-box sparsity estimation that eliminates false-negative failure modes offers significant value to the end-user, even if it comes at the cost of false-positive failure modes. In other words, the focus of black-box sparsity estimation within this context should be to build up a *conservative* estimate of the sparsity pattern that trades away specificity in favor of sensitivity.

6.2 NaN-Propagation Overview and Demonstration

This thesis contribution introduces a novel technique that we call *NaN propagation* to trace these input-output dependencies through black-box functions. The name refers to "not-a-number" (NaN), which is a special value in floating-point arithmetic that is often used to represent undefined, missing, or otherwise unrepresentable values. The proposed technique exploits the fact that Not-a-Number (NaN) values are universally propagated through floating-point numerical computations; colloquially, they tend to "contaminate" any calculation that is given a NaN input. For example, a dyadic[2] operation with a just one NaN operand will return NaN. This behavior is explicitly part of math library APIs that follow the IEEE 754 standard [199], first established in 1985. Because nearly all floating-point math libraries created since the advent of IEEE 754 follow it, this presents a fascinating way to potentially trace sparsity in a way that is essentially independent of math library or programming language. By systematically contaminating inputs to a black-box function with NaN values and observing which outputs become NaN, an estimate of the sparsity pattern can be reconstructed.

As a high-level comparison against the existing technique described in this section, this proposed new technique:

- Eliminates a false-negative failure mode, namely coincidental zero gradients.
- Adds a new false-positive failure mode, for functions with internal mathematical cancellation.
- Is either equivalent in runtime speed, or potentially much faster due to *short-circuiting operators*. This consideration is primarily determined by behavior of the math library used by the black-box function.

[2]referring to a function that accepts two input parameters

- Is compatible with most black-box functions, but not all, due to possible internal handling of NaN values. However, this incompatibility is usually easily and immediately detected, allowing the user to fall back to the existing technique.

This new technique, and strategies and mitigations around these four high-level tradeoffs, are the focus of the remainder of this chapter.

6.2.1 Example Black-Box Function: Wing Weight Model in TASOPT

Some of the more nuanced details of the proposed NaN-propagation technique are most clearly explained through demonstration. To do this, we leverage one of the constituent submodels within *TASOPT*, an aircraft design optimization suite for tube-and-wing transport aircraft developed by Drela [16]. The submodel we use is a wing weight model [3], with the major modeling considerations shown in Figure 6-1. The model is relatively detailed, with 38 inputs and 37 outputs. Inputs include geometric parameters, material properties, and configuration-level decisions (e.g., the number and location of engines, or the presence and dimensions of a strut). Outputs are the weights of various wing components, metrics describing the weight distribution, stresses at key points, and other quantities of interest.

[3] In TASOPT, this model is named `surfw`.

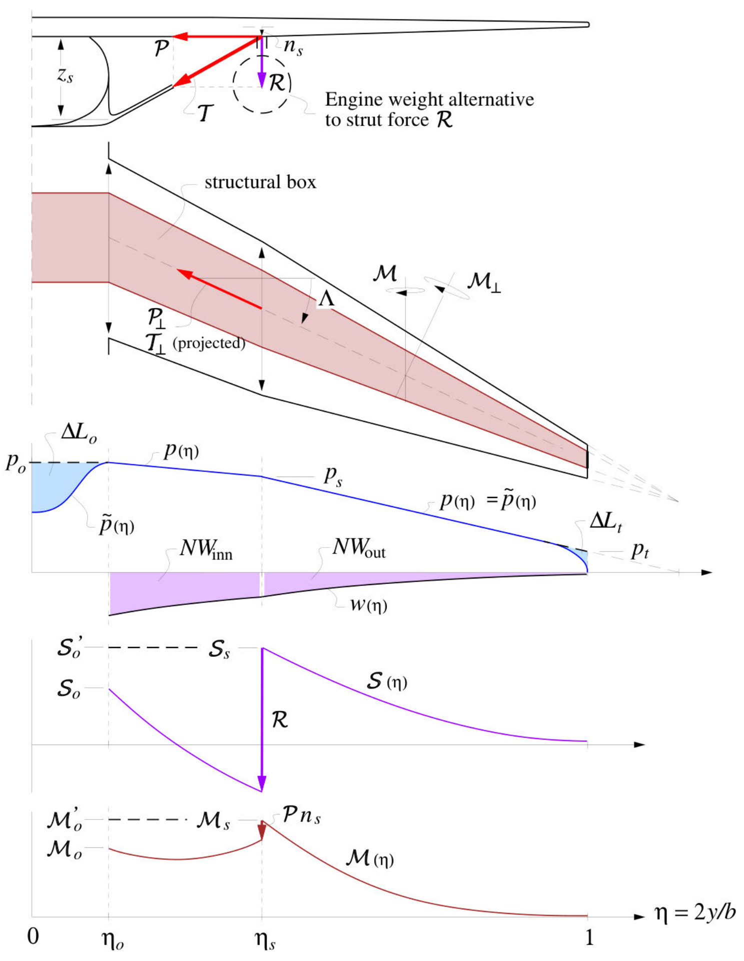


Figure 6-1: Illustration of TASOPT's wing weight model, reproduced from Drela [16]. The model uses an Euler-Bernoulli beam model to compute shear and moment distribution, and accepts a relatively large set of geometric parameters as inputs.

For the purposes of this demonstration, the code implementation of this model is accessed through *TASOPT.jl*, which is a transpilation of the original Fortran code into Julia performed by Prakash et al. [200]. We aim to estimate the sparsity pattern of this function from within Python while accessing this model only through a Python-Julia interface, making this a true black-box function written in an entirely separate programming language.

6.2.2 Sparsity Evaluation

The first step in the NaN-propagation technique is to obtain a representative input. In the context of an optimization problem, the initial guess provides a natural choice:

```
input_vector = [114115.099, 37.533, 10.697, ..., 2700.0, 817.0]
```

Next, we create a set of "contaminated" inputs, essentially by merging this input with one-hot encoded NaN values:

```
contaminated_input_vectors = [
                [       NaN, 37.533, 10.697, ..., 2700.0, 817.0],
                [114115.099,    NaN, 10.697, ..., 2700.0, 817.0],
                [114115.099, 37.533,    NaN, ..., 2700.0, 817.0],
                ...,
                [114115.099, 37.533, 10.697, ...,    NaN, 817.0],
                [114115.099, 37.533, 10.697, ..., 2700.0,   NaN],
]
```

Finally, we evaluate the black-box function at each of these contaminated inputs, and observe which outputs become NaN. For example, the output for one such contaminated input is shown below:

```
contaminated_outputs[18] = [NaN, 3475012.2, NaN, ..., 170545.5, NaN]
```

This simple procedure yields the complete bipartite graph of which inputs and outputs are linked, and this can be interpreted as a sparsity pattern. For the demonstration case study, the sparsity pattern is shown in Figure 6-2.

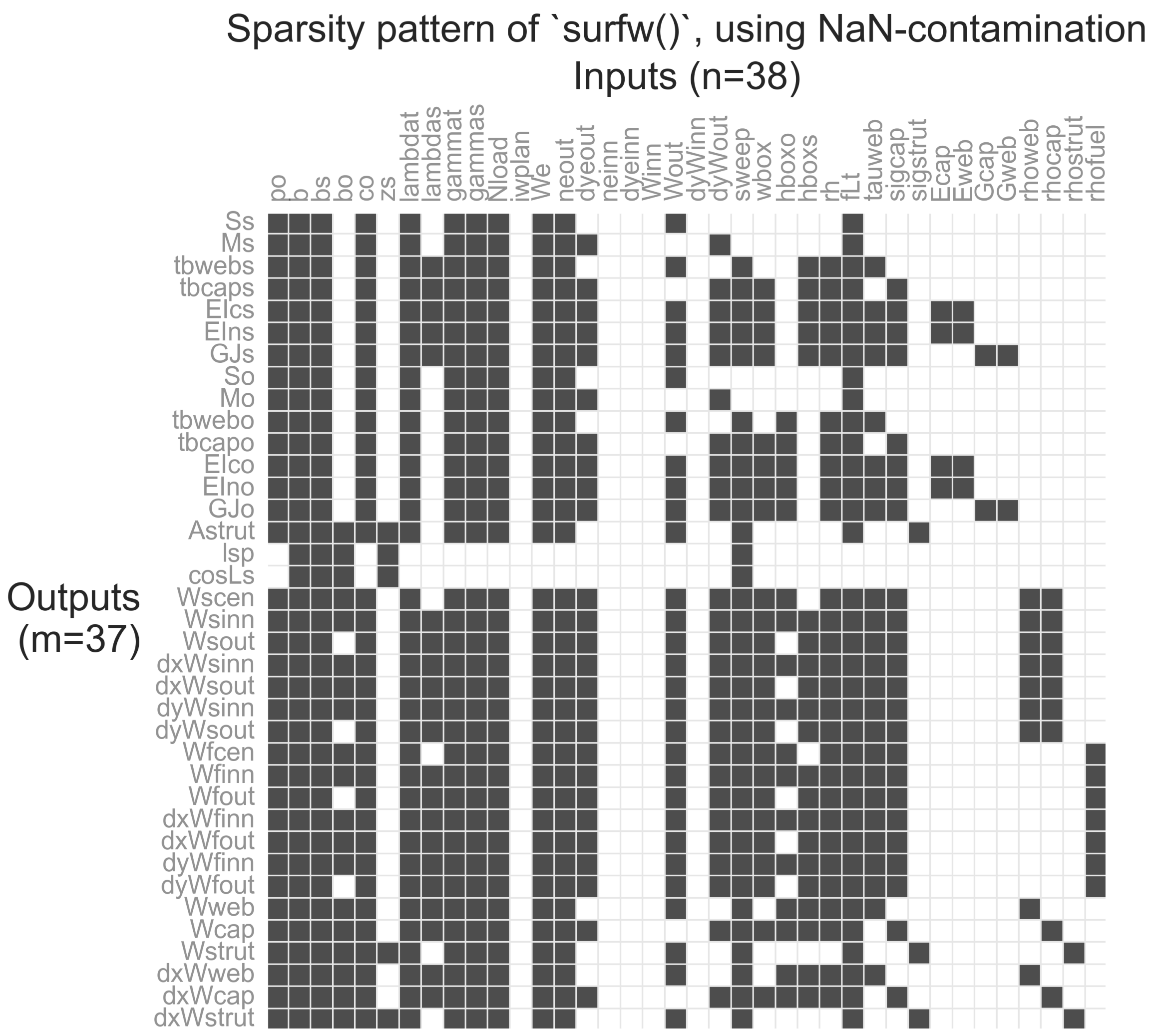


Figure 6-2: Sparsity pattern of the TASOPT wing weight model, estimated using NaN propagation. In this visualization, gray squares indicate nonzero entries in the Jacobian, while white squares indicate zero entries.

A key result is that this NaN-contamination technique eliminates the possible false-negative failure mode of coincidental zero gradients that can occur with existing gradient-based methods. Consider the example case discussed in Section 6.1, where the sparsity of the function $f(x) = x^2$ is traced at $x = 0$. Here, a NaN-contamination technique will still detect the dependency between x and $f(x)$, while existing methods will not. Because the potential consequence of a false-negative is so large, this represents a significant advantage over existing methods.

6.2.3 Jacobian Compression

This sparsity pattern can be used to enable gradient computation accelerations via simultaneous evaluation. More precisely, this involves Jacobian compression across columns, since the gradients of this black-box will later be obtained with finite-differencing ("forward-mode"-like behavior). Briefly, the steps to achieve this are:

1. Convert the sparsity pattern into an undirected graph representation where each node corresponds to an input (i.e., a column in Figure 6-2), and the presence of each edge indicates that the corresponding pair of inputs has overlapping sparsity. Conveniently, the adjacency matrix of this graph can be obtained by computing the Gramian of the sparsity pattern. If the sparsity pattern is represented as the binary matrix S, then the adjacency matrix of the graph is the binary matrix $S^T S \neq 0$.

2. Apply a vertex coloring algorithm to this graph. This is a well-studied problem with many efficient approximate algorithms available [201].

Visually, the result of this process can be shown with the compressed Jacobian of Figure 6-3. Here, several columns show a combination of multiple inputs, and gradients with respect to these inputs can be safely computed simultaneously due to structural independence. The improvement in gradient computation speed is case-dependent, but in this demonstration, a reduction from 38 inputs to 25 effective inputs is achieved. Because gradient computation runtime scales linearly with the number of inputs in a finite-difference (or complex-step) context, this means that all future gradient calculations are roughly $38/25 \approx 1.52$x faster than a dense Jacobian construction.

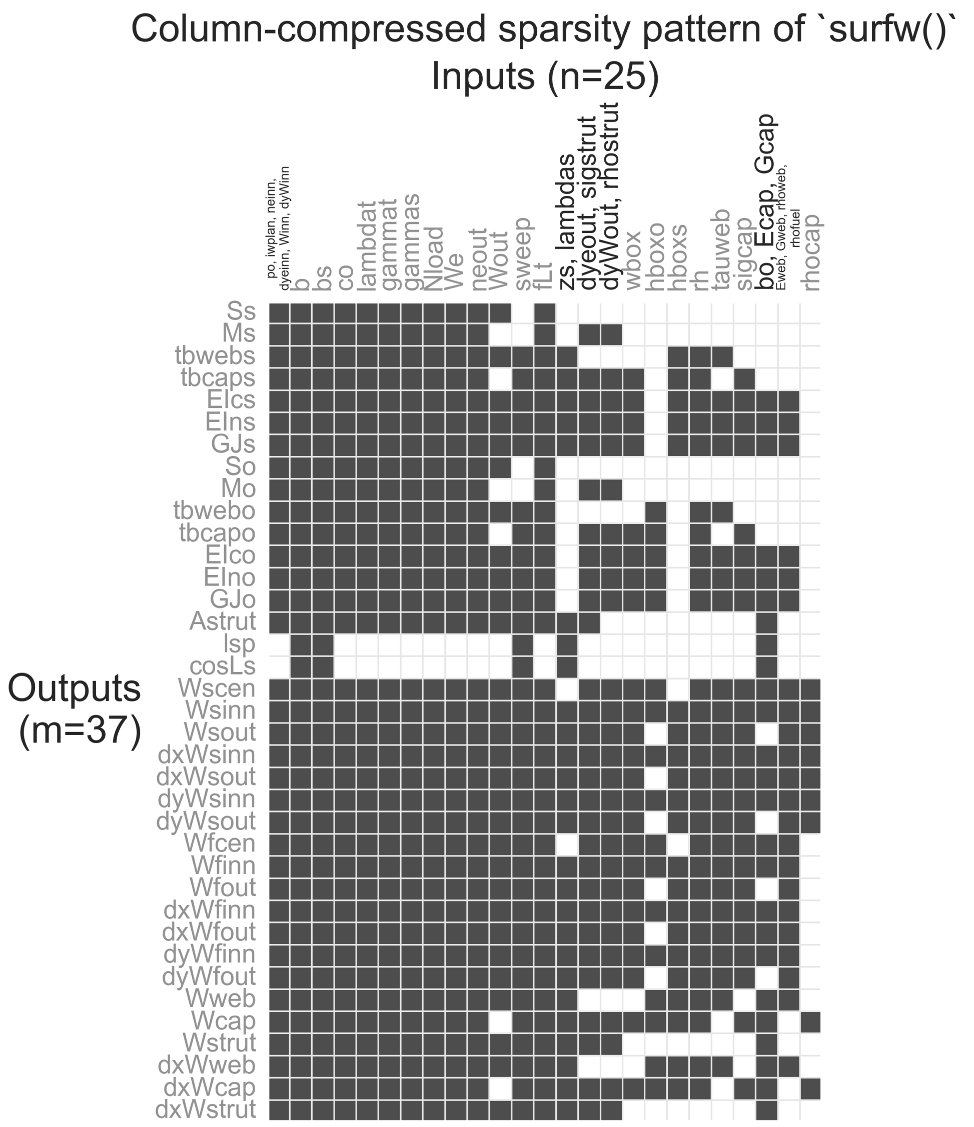


Figure 6-3: Compressed sparsity pattern of the TASOPT wing weight model, obtained by applying vertex coloring to the sparsity pattern of Figure 6-2.

6.3 Speed Considerations

This gradient acceleration comes at a small upfront computational cost, since performing the sparsity trace requires a number of function evaluations equal to the number of inputs. This is true for both existing methods (described in Section 6.1) and for NaN-contamination. This cost is nearly always worthwhile in the context of design optimization: for the TASOPT wing model, a sparsity trace "pays for itself" in runtime if the subsequent optimization process requires more than three gradient evaluations.

In some cases, the runtime of a NaN-contamination-based sparsity trace will be essentially equal to that of standard finite-difference-based sparsity estimate. Intuitively, this makes sense, because both approaches require the same number of black-box function evaluations. However, in other cases, NaN-contamination can actually be significantly faster due to operator *short-circuiting*. In this context, short-circuiting refers to operators that will check for NaN input values before performing computation; if any are found, the computation is skipped and a NaN is immediately returned instead. Programming languages and math libraries vary widely in whether and how they implement NaN-short-circuiting, so the magnitude of this potential speed-up is highly case-dependent.

6.4 Potential Limitations

While NaN-contamination eliminates a notable false-negative failure mode compared to existing methods, it also has several possible pitfalls that merit discussion. Some of these pitfalls are shared with existing methods, while others are unique to this technique.

6.4.1 Mathematical False-Positives

One limitation of NaN-contamination is that it can introduce new false-positive failure modes. To illustrate one possible mechanism, consider a scenario where we attempt to NaN-propagate through a black-box function written in code as $f(x) = x - x$. If x is real-valued, then the correct result of a sparsity trace is that the input x and the output $f(x)$ are structurally unrelated. However, a NaN-contamination technique will indicate a possible dependency here, as NaN values do not subtractively cancel. In contrast, existing gradient-based methods will correctly identify independence. More generally, this kind of false-positive can occur in any function where self-cancellation leads to structural independence, such as $f(x) = \sin(x)^2 + \cos(x)^2$ or $f(x) = x^0$.

In some ways, this false-positive is an unavoidable result of the same properties that allow NaN-contamination to eliminate false negatives due to coincidental zero gradients. In theory, global knowledge of this function as a computational graph could allow this self-cancellation to be detected and avoided. However, this would require a level of introspection that is fundamentally incompatible with the black-box nature of the function.

6.4.2 Internal NaN Handling

NaN-contamination is not compatible with all black-box functions. In particular, black-box functions that internally handle NaN values may not allow propagation. Most commonly in this scenario, the function will instead raise an error and halt the computation. This behavior is easily and immediately detected, and the sparsity detection routine can fall back to existing gradient-based methods in these cases.

A more serious false-negative failure mode can occur if the program actively overwrites NaN outputs without raising an error, as this can result in false negatives in the sparsity trace. Fortunately, this overwriting behavior is quite rare in engineering analysis code today. The main way this can occur is if the black-box code returns special flag values (sometimes called *magic numbers*) to signal invalid computation, though this is generally considered an anti-pattern and avoided in numeric code. Even in code that has this overwriting behavior upon seeing a NaN result, most codes will at least raise a warning to the user.

Another possible false-positive failure mode can occur if operators within the black-box are overly aggressive about propagating NaN values. For example, consider a simple matrix-vector multiplication of the form $\vec{f}(\vec{x}) = \mathbf{A}\vec{x}$. If a single element of the matrix A_{ij} is NaN, this should result in a single

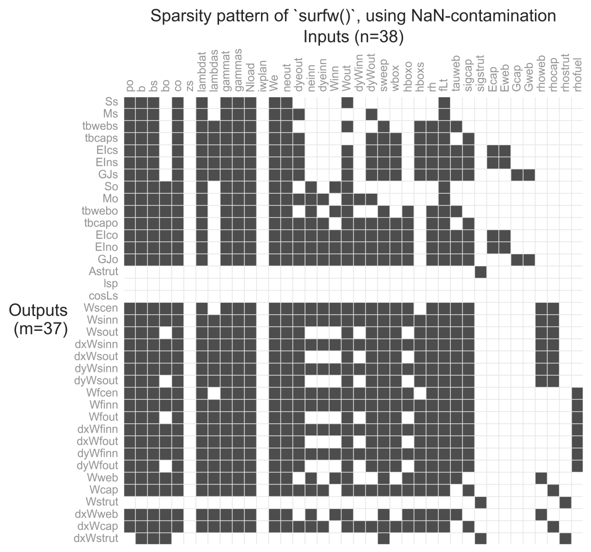
NaN output f_i, while the remaining elements are unaffected. Most programming languages and math libraries (e.g., NumPy, Julia) implement this behavior, allowing an accurate sparsity trace. However, some older libraries may aggressively contaminate and yield a NaN output for the entire output vector, which effectively results in false-positives.

6.4.3 Branching Code Execution

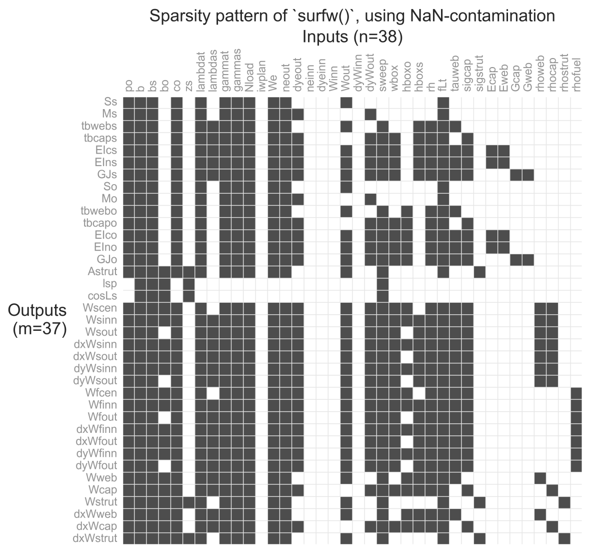
Finally, branching code execution remains a challenge for NaN-contamination-based sparsity tracing, just as it is with existing methods. An example of this can be shown with the wing weight demonstration problem described in Section 6.2. Here, the input variable `iwplan` is essentially an enumerated type that controls the wing configuration. Specifically:

- `iwplan = 0` or `iwplan = 1` corresponds to a cantilever wing without a strut.
- All other values of `iwplan` correspond to a wing with a strut.

Perhaps unsurprisingly, the sparsity pattern of the function becomes materially different if the wing configuration is changed by adding a struct. This is shown in Figure 6-4.



(a) Without strut (`iwplan = 1`)



(b) With strut (`iwplan = 2`)

Figure 6-4: Sparsity pattern of the TASOPT wing weight model, estimated using NaN contamination, as the presence of a strut is varied by changing an input variable.

With respect to branching code execution, NaN-contamination based strategies may have an advantage over existing methods due to the presence of *static conditional* operators. These operators, which are offered in several math libraries, allow for conditional statements to be represented at an *operator* level rather than a *language* level. (In many languages, this is a ternary operator with a name similar to `ifelse`, `cond`, or `where`.) In such operators, both branches of the conditional are evaluated, yet only one is returned. This contrasts with language-level conditionals (e.g., traditional `if` statements), where the actual path of code execution changes. In some math libraries (e.g., CasADi) static conditionals allow NaNs to propagate through regardless of the condition—if either branch evaluates to NaN, the result is a NaN even if the NaN branch would not ordinarily be taken. Because of this, NaN-contamination-based sparsity tracing can potentially handle branching code execution better than existing options, depending on the math library used by the black box.

6.5 Mitigating Branching Code Execution for Black-Box Engineering Analyses

While branching code execution inherently poses problems for black-box sparsity tracing in the general sense, there are heuristic strategies that can be used to improve performance in practical cases. In particular, we look at strategies that tend to work well specifically in the context of engineering analysis code. In these cases, code branching often corresponds to variables that either change the design configuration or the mode of an analysis. The `iwplan` variable discussed in Section 6.4.3 is an example of this. With this in mind, we can make two observations specific to engineering analysis code that can be exploited.

First, inputs that trigger branching code execution – collectively, "flag inputs" – are often represented as discrete data types (e.g., integers, booleans, or strings) rather than continuous types (floating-point values). Because of this, knowledge of the input data types would allow some inferences to be drawn about which inputs are likely to trigger branching execution. In some cases, users may be able to manually mark flag inputs, either by leveraging domain knowledge or by inspecting known type-stable code[4]. Of course, in the general case of a black-box function, this ability to inspect a function's type signature (either manually or dynamically) is not guaranteed. However, one possible option is to perform automatic type inference at runtime based on the types that constitute the initial guess.

Secondly, we can also take advantage of the fact that, in many cases, the value of flag inputs remains fixed during a single optimization study. For example, the design configuration may be kept fixed in a sizing study, or an analysis may only be run under certain assumptions. In such cases, the potential

[4]e.g., if the black-box function is written in a statically-typed language, and the user can view at least the function signature

damage of a false negative is nullified, because the model is never evaluated on the other code branch.

One possible solution that takes advantage of both of these observations is to use a greedy approach. Concretely:

1. Run an initial sparsity trace on the black-box model using the initial guess as the input.
2. Every time a new value is observed on a discrete-typed input, re-run the sparsity trace.
3. Take the result of this sparsity trace, and compute the union with all previously-observed sparsity patterns for this black-box model. Use this new sparsity pattern for Jacobian compression in subsequent evaluations.

A visual illustration of this method is given in Figure 6-5. Here, the sparsity pattern is iteratively refined as new values are observed on the discrete-typed input.

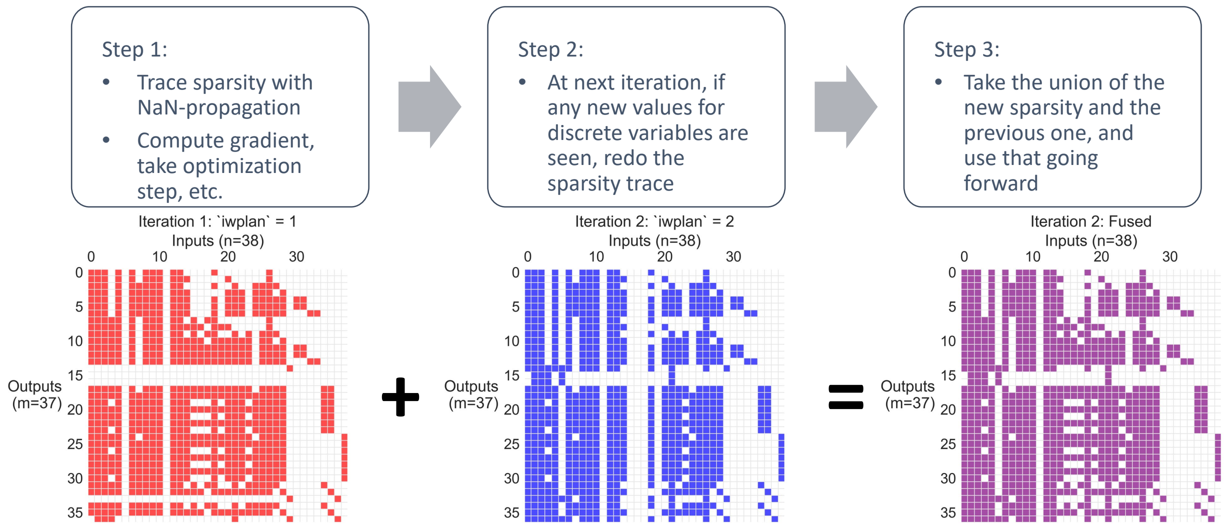
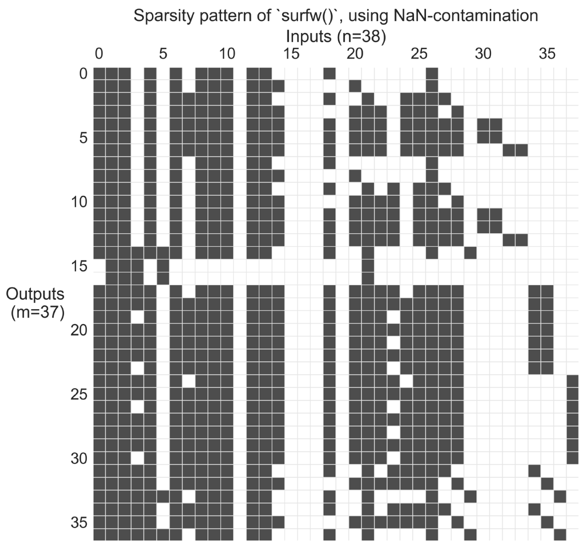

Step 1:
Trace sparsity with NaN-propagation
Compute gradient, take optimization step, etc.
Step 2:
At next iteration, if any new values for discrete variables are seen, redo the sparsity trace
Step 3:
Take the union of the new sparsity and the previous one, and use that going forward
Iteration 1: `iwplan` = 1
Iteration 2: `iwplan` = 2
Iteration 2: Fused
Inputs (n=38)
Outputs (m=37)

Figure 6-5: Illustration of a greedy algorithm for iteratively estimating the sparsity pattern of black-box models, in the presence of branching code execution. This approach is intended for use with engineering analysis models, and the process is applied to the example problem of Section 6.2.

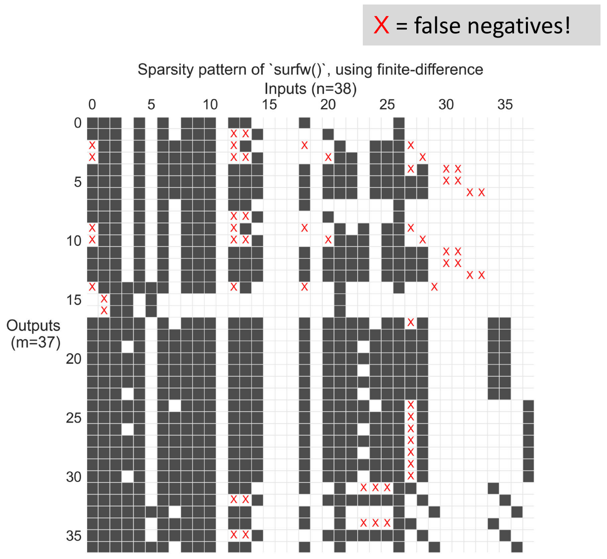
It should be emphasized that this algorithm is only a heuristic, and piecewise branching on continuous variables remains problematic. However, this strategy still offers substantial practical improvements over existing techniques that use a sparsity pattern purely based on Jacobian evaluation at the initial guess.

6.6 Overall Comparison to Existing Methods

In summary, the main advantage of the NaN-propagation technique is that it eliminates a key false-negative failure mode of existing gradient-based sparsity detection methods. To compare this more precisely, we can show a side-by-side comparison of the sparsity patterns obtained by each method for the TASOPT wing weight model. This is given in Figure 6-6. In this example, the initial guess used for Jacobian evaluation is taken from the TASOPT-supplied default for aircraft design optimization. Subfigure 6-6b reveals dozens of false negatives (marked as X) that are not captured by existing methods, but are detected with NaN-propagation. This clearly demonstrates the potential utility of NaN-propagation for sparsity-tracing black-box code functions.



(a) Sparsity pattern obtained by NaN-propagation.



(b) Sparsity pattern obtained using existing finite-difference methods. All red X marks denote false negatives that are not captured by existing methods, but detected with NaN-propagation.

Figure 6-6: Comparison of sparsity patterns obtained by NaN-propagation and existing finite-difference methods for the TASOPT wing weight model. Existing methods lead to false negatives via coincidental zero gradients, which can lead to costly mistakes during Jacobian compression.

6.7 Advanced Strategies

This chapter has introduced the core NaN-propagation technique, along with its benefits over existing methods and potential limitations. In this section, we introduce several advanced strategies that can be used to further improve the performance of NaN-propagation-based sparsity tracing.

6.7.1 NaN Payload Encoding

One possible extension is based on a recognition that not all NaN values are identical, and there are multiple unique binary representations that are interpreted as NaN values. Careful use of these values can essentially allow the encoding of additional information to aid sparsity tracing. To illustrate this, consider the binary representation of a single-precision (32-bit) floating-point NaN value, following the IEEE 754 standard [199]:

```
NaN = x111 1111 1xxx xxxx xxxx xxxx xxxx xxxx
```

where: `0, 1` = bit values

`x` = *any* bit (with the edge-case exception that not all trailing bits can be zero, or the value is interpreted as ∞)

Regardless of the value of the `x` bits, this value will be interpreted as a NaN. Therefore, single-precision NaN representations allow us to encode nearly 22 bits of information (as there are $2^{22} - 2$ unique NaN values) into the these `x` bits, which we call the *payload*. In double-precision (64-bit) floating point arithmetic, which is more common in engineering analysis, this payload is even larger at nearly 51 bits.

Crucially, these unique NaN payloads are propagated through mathematical operations. Therefore, we can "tag" individual inputs with unique NaN payloads, and observe which payloads are present in the outputs. Because NaN values are no longer fungible, *multiple columns of the sparsity pattern can be simultaneously traced within a single function evaluation*. This presents a novel and fundamental theoretical improvement in the time-complexity of black-box sparsity tracing, since the number of function evaluations can now be independent of the number of inputs.

An immediate natural follow-up question to ask is how NaN values are propagated through dyadic operations: in cases where a function receives multiple payload-encoded NaN arguments, it is not obvious what the payload of the output will be. In general, most programming languages and math libraries will simply propagate one of the two arguments unmodified, and ignore the other. However, which argument is preferred varies. For example, in Python, the addition, subtraction, and multiplication operators propagate the first NaN input, while the division and exponentiation operators propagate the second NaN input.

Because of this effect, some care is needed to take advantage of NaN payload encoding. To demonstrate this, consider a dyadic black-box function of the form $f(x_1, x_2)$. Say we perform a sparsity trace by evaluating the function at $x_1 = \texttt{NaN}_1$, $x_2 = \texttt{NaN}_2$, with subscripts representing that these are uniquely identifiable `NaN` values. If the output of function f is $\texttt{NaN}_1$, this indicates that f is surely a function of x_1, but the sparsity with respect to x_2 is unknown. Likewise, the opposite is true if the output is $\texttt{NaN}_2$. In such cases, these simultaneous column evaluations of the sparsity pattern can be thought to "shadow" one another: a sparsity "hit" on one column precludes receiving any information about the other column(s). Because of this *shadowing effect*, the time complexity of NaN-contamination with payload encoding is not quite $\mathcal{O}(1)$, though it can still be better than the $\mathcal{O}(N)$ of existing approaches (discussed later). Regardless, the real benefit associated with payload encoding occurs when the output f is a function of neither input, as this allows exclusion of both inputs from the sparsity pattern with a single function evaluation. This provides theoretical upper- and lower-bound cases on the method's time complexity. To reason more intuitively and loosely, any strategy that allows us to gain information about multiple inputs from a single function evaluation should be as least as fast as standard $\mathcal{O}(N)$ time, and often faster.

This sparsity pattern construction process via NaN-payload-encoding requires an algorithm to select which combination of columns should be simultaneously evaluated next for sparsity tracing. Development of an efficient column selection algorithm is left as an area of future work. While this may at first appear to be a variant of a vertex coloring problem, the shadowing effect creates meaningful differences and a far more fascinating algorithmic challenge. To aid such future efforts, we offer some initial observations that may guide this algorithmic development for the interested reader:

1. During sparsity tracing, the state of the sparsity pattern needs to be stored in a way that captures all currently-known information: one possible representation is a trinary matrix with three possible states: `0` (no dependency), `1` (dependency), and `2` (unknown). (Other state representations that admit this partial observability exist[5].) Because decisions about column selection need to be made while the sparsity pattern is only partially observable, any good algorithm is inherently probabilistic. This unknown factor is also why this algorithm is not simply a degenerate case of a graph coloring algorithm.

2. A good algorithm likely benefits from some tracker of the belief state about the overall density of the sparsity pattern (i.e., the fraction of unknown entries that are likely to be dependent vs. independent). This is because the number of columns that should be evaluated simultaneously is conditional on the estimated density of the sparsity pattern. If we believe that the sparsity pattern is relatively dense, then fewer columns should be evaluated simultaneously, as the shadowing effect will be significant. Conversely, if the sparsity pattern is believed to be sparse, then we should be more aggressive about evaluating more columns simultaneously, since we will gain more information per function evaluation in expectation.

   One possible way to implement this density belief state is to use a Bayesian approach, where the belief state is updated at every step based on the results of the previous function evaluations. The initial belief state could be an uninformative prior (e.g., a uniform distribution), or one could accept a user-supplied estimate. This approach has some flaws (e.g., assuming the sparsity of each

[5]An example would be a continuous belief state based on priors about how engineering code sparsity patterns tend to be arranged, like block-diagonality. This is more complicated, but could allow inference based on structural patterns in the partially-observed sparsity matrix. This may also have benefits in cases where branching code execution is observed, as uncertainty about sparsity can be recorded into the state and used to strategize.

entry is independent), but it presents a reasonable starting point.

3. An algorithm that selects which combination of columns should be evaluated next is inherently a combinatorial optimization problems. These are notoriously slow to solve, in the general case. However, if the black-box function is slow to evaluate (as it often is, in the case of higher-fidelity engineering analysis), it may be well worth it to solve this column-selection problem in order to minimize the number of required evaluations. In addition, there are strategies that can accelerate this combinatorial optimization problem in this context. These could include dynamic programming (since similar versions of this problem are solved sequentially), heuristic methods (e.g., simulated annealing, genetic algorithms), or continuous relaxation.

   As a practical matter, one possible objective function of this optimization formulation is to maximize the expected value of information gain (measured by entropy) in the sparsity pattern, given the current belief state. In principle, this is similar to some forms of Bayesian optimization.

4. We hypothesize that best-case time complexity[6] should approach $\mathcal{O}(\log(N))$, as measured by the number of function evaluations and where N is the number of inputs. This best-case complexity occurs in the limit of low-density sparsity patterns where the shadowing effect becomes insignificant, as column selection becomes that are amenable to recursive halving[7]. A worst-case time complexity is $\mathcal{O}(N)$, which would occur in the case of a fully-dense sparsity pattern—this is identical to both existing methods and NaN-propagation without payload encoding.

[6]ignoring the trivial case where the pattern is all-zeros and this is suspected in advance
[7]by similar logic to scaling in a binary search

6.7.2 Chunking

Another novel proposed acceleration, which we term "chunking", seeds multiple adjacent elements of the input vector with NaN values simultaneously. This is similar to the multiple seeding associated with payload NaN encoding, but its inclusion here is meant to emphasize that some benefits can be achieved even without uniquely identifying NaN values, in cases where this direct bit-manipulation may not be possible or desirable.

As an example: if pairs of adjacent inputs are seeded, this has the benefit of halving the number of function calls, at the detriment of providing overly-conservative sparsity information[8] (i.e., one with false positives). This exploits the fact that a conservative sparsity pattern can still provide some speedup during Jacobian compression, even if it is less than what might be achieved with a more accurate sparsity pattern.

It also exploits the fact that sparsity patterns of human-written code are not uniformly random. In particular, these sparsity patterns tend to have a high degree of locality (e.g., inputs at indices 3 and 4 are much more likely to share a similar sparsity pattern than inputs at indices 3 and 30). This locality is due to the fact that, in an engineering analysis with multiple submodules, humans tend to code submodule A in its entirety before implementing submodule B. Because of this, these chunked sparsity patterns using adjacent inputs can be surprisingly accurate approximations in practice.

Thus, chunking may offer significant speedups in engineering practice by exploiting typical problem structure; a computational analogy would be how an A^* graph traversal algorithm outperforms Dijkstra's algorithm in practice despite identical worst-case complexity. This and other heuristics can significantly reduce the number of black-box function evaluations required to trace the sparsity pattern.

[8]since one cannot determine which NaN input caused a given output to return NaN, so neither input can be ruled out – both inputs must be assumed to be the possible cause

6.8 Computational Reproducibility

All source code used to generate example results in this chapter is publicly available at `https://github.com/peterdsharpe/nan-propagation`.

Chapter 7

Physics-Informed Machine Learning Surrogates for Black-Box Models

This chapter includes content from a manuscript written by the author [202, 203].

This thesis contribution demonstrates a method of interfacing external black-box physics models with a code transformations paradigm. In this context, a "black-box" refers to a computational analysis where only the inputs and outputs are observable. Hence, the internal workings of such a function cannot be automatically traced into a computational graph, and code transformations cannot be applied without special treatment.

Black-box functions can arise when modeling at any level of fidelity. However, as a practical matter, the limitations of black box models become particularly troublesome when dealing with high-fidelity analyses. This is because the most straightforward workaround — rewriting the black-box analysis in a traceable form – becomes less feasible as the complexity of the model increases.

Although low- and mid-fidelity models are often more than adequate for conceptual design, there are some cases where higher-fidelity modeling with black-box codes is critical. For example, after performing initial conceptual design with low-fidelity tools, a designer may find that a particular subsystem has highly sensitive performance and wish to swap in a higher-fidelity model. Even if a user does not have a specific immediate need for black-box modeling, however, the mere fact that a framework does not allow black-box models can be a significant barrier to industry adoption. Design optimization tools can have long life cycles in organizations, sometimes leading to significant lock-in effects. Therefore, some conceptual designers are understandably hesitant to adopt a code-transformations-based design optimization framework without an assurance that mid- and high-fidelity tools have some eventual pathway to inclusion, if later needed. Hence, providing a method to bring these models into a code transformations paradigm is an important step towards boosting practicality.

Over the past decade, physics-informed machine learning tools have become an increasingly popular way to create rapid surrogates of high-fidelity models for complex physical systems. Because these models are typically constructed to be compatible with machine learning frameworks, they are invariably traceable — if not by the design optimization framework, at least by the machine learning framework. Therefore, there is strong theoretical reason to believe that such models could be directly usable within a code-transformations-based design optimization paradigm, with appropriate framework interfaces. However, the practicality of this approach has not been widely demonstrated in the context of aerospace design optimization.

In addition to these questions about framework compatibility, there are also more traditional concerns to address that are relevant for any surrogate modeling approach. These include common questions about speed-accuracy tradeoffs, but also about applied concerns, such as the generalizability and robustness of these machine learning surrogate models in the presence of adversarial optimization.

To demonstrate and examine some of these effects, this chapter describes a case study for a tool called *NeuralFoil*. NeuralFoil is an open-source Python-based tool for rapid aerodynamics analysis of airfoils, similar in purpose to XFoil. Speedups ranging from 8x to 1,000x over XFoil are demonstrated, after controlling for equivalent accuracy. NeuralFoil computes both global and local quantities (lift, drag, velocity distribution, etc.) over a broad input space, including: an 18-dimensional space of airfoil shapes, possibly including control deflections; a 360° range of angles of attack; Reynolds numbers from 10^2 to 10^{10}; subsonic flows up to the transonic drag rise; and with varying turbulence parameters. Results match those of XFoil closely: the mean relative error of drag is 0.37% on simple cases, and remains as low as 2.0% on a test dataset with numerous post-stall and transitional cases. NeuralFoil facilitates gradient-based design optimization, due to its C^∞-continuous solutions, automatic-differentiation-compatibility, and bounded computational cost without non-convergence issues.

NeuralFoil is a hybrid of physics-informed machine learning techniques and analytical models. Here, physics information includes symmetries that are structurally embedded into the model architecture, feature engineering using domain knowledge, and guaranteed extrapolation to known limit cases. This chapter also introduces a new approach to surrogate model uncertainty quantification that supports robust design optimization.

In this chapter, we discuss the methodology and performance of NeuralFoil with several case studies, including a practical airfoil design optimization study including both aerodynamic and non-aerodynamic constraints. Here, we show that NeuralFoil optimization is able to produce airfoils remarkably similar in performance and shape to expert-designed airfoils within seconds; these computationally-optimized airfoils provide a useful starting point for further expert refinement.

7.1 Introduction

In conceptual aircraft design, the problem of shaping a typical wing is usually decomposed into two parts: planform design and airfoil design. The latter, which is the focus of this case study, is a multidisciplinary design problem that requires consideration of a variety of aerodynamic, structural, and manufacturing objectives and constraints. A non-exhaustive list of major considerations could include:

- Profile drag across the expected operating range of the airfoil (spanning lift coefficients, Reynolds numbers, and Mach numbers), including adequate off-design performance [33];
- Pitching moment and aft-camber coefficients, which can drive tail sizing (modifying trim drag), affect divergence speed;
- Hinge moments and control effectiveness of any control surfaces, which drive actuator design and weight;
- Stall behavior, which can affect handling qualities and safety;
- Thickness at various points, in order to accommodate fuel volume and required structural members to resist failure (e.g., by bending, buckling, divergence, flutter, or control reversal);[158]

- Sensitivity to boundary layer performance, freestream turbulence, and trips, all of which impose constraints on surface finish, cleanliness, and manufacturing tolerances [204–206];
- Peak suction pressures, which affect the critical Mach number in transonic applications or cavitation in hydrodynamic applications;
- Shock stability and buffet considerations in transonic applications;
- Manufacturability, which might include flat-bottom airfoil sections, strictly-convex airfoil shapes (e.g., to accommodate shrink-coverings, which are common in ultra-lightweight applications [207]), or restrictions on trailing-edge angle.

These airfoil design drivers often differ considerably at different locations along the span of the wing, which often leads to a family of airfoils being used in the design of a given wing. To fulfill such design requirements, a designer will typically either find an existing airfoil or design a new airfoil. Given the specificity of the requirements illustrated in the list above, designing bespoke airfoils can yield considerable performance improvement.

Currently, three approaches are commonly used to computationally design new airfoils: inverse design methods, direct manual methods, and optimization methods. In the inverse design approach, popularized by Drela's XFoil code [95] and Eppler's Profil code [208, 209], conformal mapping methods are used to reconstruct a new airfoil shape from a user-specified pressure distribution. This has the benefit of allowing the engineer to directly operate on the most relevant aerodynamic quantities, and it produces considerable design insight. However, it can be time-consuming to produce airfoils that satisfy non-aerodynamic constraints (e.g., manufacturability, spar thickness) due to the lack of direct control here. Conformal mapping methods are also only strictly applicable to potential-flow-governed regions, so the specified pressure distribution is often only the inviscid, rather than the viscous (true) pressure distribution[1].

In the direct manual method (or "geometric design" method, using parlance from XFoil), an engineer formulates an airfoil shape directly and modifies this iteratively by hand. Aerodynamic analysis is performed by any code that will perform the forward problem (geometry $\rightarrow$ aerodynamics), such as XFoil, MSES [210], or any RANS-based CFD code [84]. This allows for easier satisfaction of non-aerodynamic constraints. However, it is predictably more difficult to directly target aerodynamic quantities. Significant user expertise is also required to identify the most relevant geometric parameters and to make effective changes to the airfoil shape.

In the optimization approach, a parameterized airfoil shape is optimized to minimize a cost function and satisfy specified constraints (which may be both aerodynamic and non-aerodynamic). At first glance, this appears to be an automation of the direct manual method. However, Drela and others note the surprising difficulty of posing the correct optimization problem [32, 33], so this approach often requires just as much (if not more) human expertise than the direct manual method. As stated by Drela [33], optimization-based airfoil design "is still an iterative cut-and-try undertaking. But compared to [direct] techniques, the cutting-and-trying is not on the geometry, but rather on the precise formulation of the optimization problem." To support this, Drela gives compelling case studies of how airfoil design optimization can go awry in the absence of user review and care[2]. However, despite these cautionary notes, the optimization-based airfoil design approach also offers compelling benefits: resulting airfoil performance can equal that of airfoils sculpted by an expert user, particularly on problems with unique or

[1]This can be alleviated by using nonlinear optimization to target the viscous pressure distribution, albeit with reduced numerical robustness.

[2]Some codes, like LINDOP [210], alleviate this somewhat by using a hybrid of the direct and optimization approaches: update directions are computed by an optimizer, but the actual changes are reviewed and implemented by a human between iterations.

otherwise non-intuitive constraints [33]. This optimization process can also require orders of magnitude less engineering time, and it provides a systematic and disciplined approach that is especially suited to the most challenging design problems [211].

In all these methods, some form of a computational tool for airfoil aerodynamics analysis is required. For subsonic airfoils, the industry gold standard of such tools is XFoil [95]. Morgado et al. find that XFoil actually *exceeds* the accuracy of RANS-CFD-based tools in subsonic cases [212], yet it has a computational cost that is roughly 1,000x lower than RANS approaches – a testament to the power of its modeling approach, which strongly-couples integral boundary layer and potential flow methods. A complete description of this modeling approach is available in Drela's *Aerodynamics of Viscous Fluids* [213], and in recent state-of-the-art work by Zhang [184, 214]. However, despite XFoil's many strengths, it has several attributes that make it less-than-ideal for directly driving numerical optimization studies [84]. Among these:

- XFoil is not guaranteed to produce a solution. When an "ambitious" calculation is attempted, XFoil often fails to provide a converged solution; the unconverged result often has wildly-diverging values and is effectively unusable. In some cases, calculations can lead to infinite loops or process crashes due to unhandled exceptions. While this is acceptable in certain applications (e.g., manual direct analysis), it is generally unacceptable for use in numerical optimization. Instead, optimization strongly benefits from a robust analysis tool that always produces a result, even if that result has degraded accuracy; this allows the analysis to steer the optimizer back towards the design space of reasonable airfoils [211]. A particularly useful attribute is when the model is deliberately made to be slightly pessimistic (e.g., overestimate drag) in regions of the design space with high uncertainty, adding further optimization pressure towards reasonable designs with low performance uncertainty.

    - More generally, design optimization is not the only application that strongly benefits from an aerodynamics analysis tool that always produces an answer. Other examples where a non-answer, infinite loop, or crashed process are unacceptable include real-time control (e.g., as an aerodynamic model for a model-predictive controller onboard an aircraft) and flight simulation.

- XFoil solutions are not necessarily unique, and slightly different solutions can be obtained for the same analysis problem (airfoil shape, angle of attack, and Reynolds and Mach numbers). In practice, this manifests as an effective hysteresis depending on whether the angle of attack is swept up or down. This flow non-uniqueness is in fact a real physical[3] effect [211, 215, 216]. However, this non-uniqueness can be exceptionally problematic for numerical optimization, as an infinitesimal change in an input parameter can result in the solution jumping to a different Newton basis of attraction. Therefore, there is no limit to how sensitive performance can be to input parameters, which hampers techniques like finite-differencing for gradient-based optimization.

- XFoil solutions are non-smooth[4] with respect to input parameters, which makes them fundamentally incompatible with gradient-based optimization. (Any attempt to directly optimize XFoil results with gradient-based methods invariably results in premature stopping at a local minimum.) Interestingly, this is a consequence of how laminar-turbulent transition is handled by XFoil's integral boundary layer solver. This solve requires the use of laminar and turbulent boundary layer *closure models*, which are curve-fitted functions that yield various necessary quantities (H^*, c_f, $c_\mathcal{D}$,

[3]For example, flow over an airfoil may separate as its angle of attack increases past 12°, but it may not fully reattach until the angle of attack descends back to below 11°

[4]precisely, they are C^0 continuous but not C^1 continuous

etc.) of the von Karman integral momentum and kinetic energy equations as a function of the two values that parameterize the boundary layer (H, Re_θ). The laminar and turbulent versions of these functions differ. XFoil implements a cut-cell approach on the transitioning interval, which restores C^0-continuity (i.e., transition won't truly "jump" from one node to another discretely); however, a sharp change in gradient occurs whenever an individual node switches its equation from laminar to turbulent. Adler et al. provide a graphical depiction of this phenomenon [84].

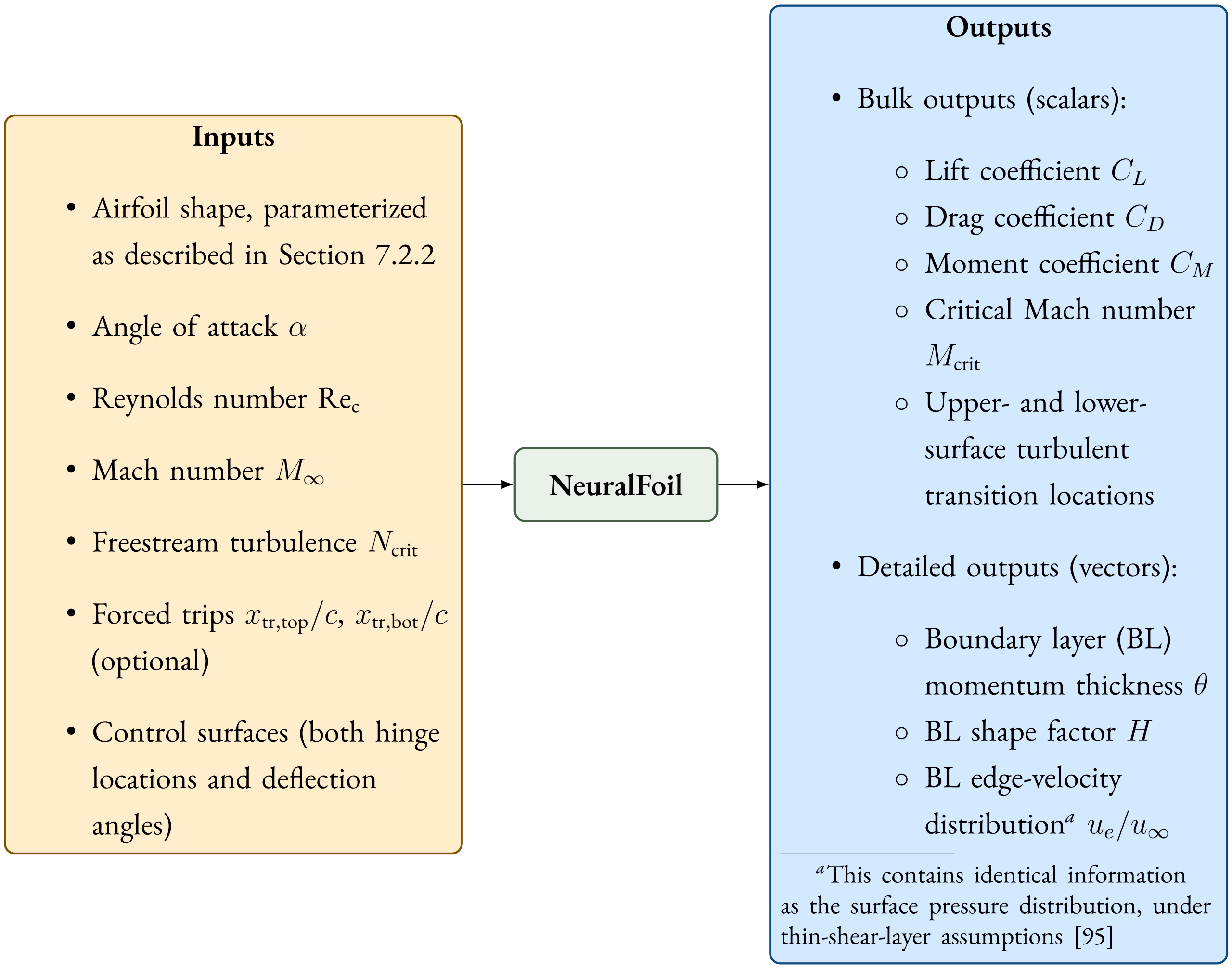
- Most interfaces between an optimizer and XFoil communicate through a series of text files (i.e., hard disk), rather than by sharing data in memory (i.e., RAM). Given the quick speed of an individual XFoil run, this input-output overhead imposes a non-negligible performance penalty. While this programming-language-agnostic interface is arguably one of the reasons for XFoil's long-enduring popularity, it comes at the cost of runtime performance if the tool is to be used in a high-throughput setting (e.g., for design optimization or aerodynamic database construction).

This motivates the development of a new airfoil aerodynamics analysis tool that captures the advantages of XFoil (accuracy, speed) while mitigating these drawbacks (i.e., incompatibility with gradient-based optimization, non-convergence challenges). In recent years, many fields have benefited from a hybrid data-and-theory approach [217], where data-driven models are used to augment traditional physics-based models with learned closures. This NeuralFoil case study presents a similar physics-informed approach as applied to analyzing airfoil aerodynamics.

7.2 NeuralFoil Tool Description and Methodology

7.2.1 Overview

To address these needs, this chapter introduces NeuralFoil, a tool for rapid aerodynamics analysis of airfoils, similar in purpose to XFoil [95]. A precise list of inputs and outputs to NeuralFoil is given in Figure 7-1.



Inputs
• Airfoil shape, parameterized as described in Section 7.2.2
• Angle of attack α
• Reynolds number Re_c
• Mach number M_∞
• Freestream turbulence N_{crit}
• Forced trips $x_{\mathrm{tr,top}}/c$, $x_{\mathrm{tr,bot}}/c$ (optional)
• Control surfaces (both hinge locations and deflection angles)
NeuralFoil
Outputs
• Bulk outputs (scalars):
◦ Lift coefficient C_L
◦ Drag coefficient C_D
◦ Moment coefficient C_M
◦ Critical Mach number M_{crit}
◦ Upper- and lower-surface turbulent transition locations
• Detailed outputs (vectors):
◦ Boundary layer (BL) momentum thickness θ
◦ BL shape factor H
◦ BL edge-velocity distribution[a] u_e/u_∞
[a]This contains identical information as the surface pressure distribution, under thin-shear-layer assumptions [95]

Figure 7-1: User-facing inputs and outputs of the NeuralFoil model.

Although NeuralFoil's results will be most accurate in "well-behaved" flow regimes (e.g., attached flow), reasonable aerodynamics estimates can be expected:

- For nearly all practical single-element airfoil shapes that can be analyzed with XFoil, including modifications for trailing-edge control surface deflections up to substantial deflections (roughly $\pm 40°$)
- Across the $360°$ angle of attack range, by leveraging analytical post-stall models regressed from high-α wind tunnel data by Truong [218]
- Across a large range of Reynolds numbers (roughly 10^2 to 10^{10}; described in Table 7.2), with physically-sensible extrapolation even beyond this range (e.g., Stokes flow limit)
- At Mach numbers from zero to the transonic drag rise

NeuralFoil has a mathematical form that is fully explicit (i.e., no iterative solvers are used; there is no value-dependent code execution). This guarantees that a deterministic result is returned in bounded computational time and without any need for initial guesses. This also makes compatability with automatic differentiation frameworks much more straightforward, as the computational graph is static for any input. At a high level, the mathematical model within NeuralFoil consists of the following main steps:

1. **Pre-solve**: Converts the user-supplied airfoil shape into the required parameterization (including any control surfaces, which are made part of the airfoil geometry as described in Section 7.2.2)
2. **Encoding**: Transforms all inputs (except Mach number, which is treated later) into an appropriate input vector space for the neural network. This is performed using prescribed functions based on domain knowledge.

3. **Learned Model**: A learned neural network maps from this latent vector space to another latent vector space.

4. **Decoding**: Transforms the outputs of this neural network back into the space of user-facing outputs. This is also performed using prescribed functions based on domain knowledge.

5. **Uncertainty quantification, model fusion, and extrapolation**: Based on the neural network's self-reported trustworthiness (via a process described in Section 7.2.4), the network's results are merged with analytical models for airfoil behavior in massively-separated flow conditions.

6. **Compressibility correction**: The solution is corrected for non-zero Mach numbers using analytical methods, as described in Section 7.2.4.

In the following sections, we will describe this model architecture and theoretical basis in more detail. Readers primarily interested in general performance metrics and validation studies are invited to skip ahead to Section 7.3. Readers primarily interested in a practical aerodynamic shape optimization example are directed to Section 7.4.

7.2.2 Pre-Solve

Airfoil Geometry Parameterization

NeuralFoil accepts user-specified airfoil shapes in a variety of common formats – for example, as an array of (x, y) coordinates, as a standard coordinate-array `*.dat` file, as a series of CST parameters, or as an Airfoil-class object within the AeroSandbox aircraft design optimization framework [1].

Underneath this interface layer, NeuralFoil converts this specified airfoil geometry to an 8-parameter-per-side CST (Kulfan) parameterization, including Kulfan's added leading-edge-modification (LEM) and trailing-edge thickness parameter [17, 18]. This gives a total of 18 parameters to describe a given airfoil shape, which are illustrated in Figure 7-2. This parameterization family was chosen due to work by Masters [19] and others, which shows that this is one of the most parameter-efficient representations of airfoil shape. Kulfan's parameterization is strongly related to an orthogonal polynomial decomposition using Bernstein polynomials. Because of this, the format is interpretable: it is a linear combination of mode shapes, each of which is roughly locally-supported[5]. Another benefit of the CST parameterization is interoperability, as it is commonly implemented in existing aerospace tools such as OpenVSP [94].

The CST parameterization allows for varying numbers of degrees of freedom. An 18-parameter representation was chosen based on the work of Masters [19], which shows that error in aerodynamic force prediction decreases substantially near this threshold. Thus, this parameterization strikes an acceptable balance between parameterization error and dimensionality. The 18-parameter representation also corresponds to one of the initially-proposed discretization levels proposed by Kulfan [17] (labeled in this work as "BPO8"), so this parameterization is a natural choice for compatibility with existing airfoil design tools.

Conversion of user-specified airfoils to this format for NeuralFoil to use is automatically and efficiently handled as a least-squares fitting problem.

[5]this contrasts with approaches such as taking an SVD over a standard airfoil database, which creates less-interpretable mode shapes

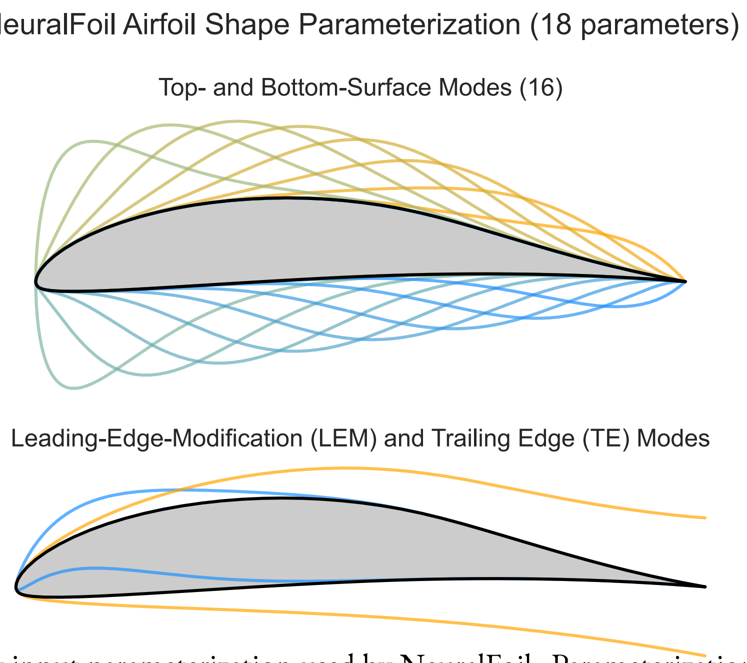

NeuralFoil Airfoil Shape Parameterization (18 parameters)
Top- and Bottom-Surface Modes (16)
Leading-Edge-Modification (LEM) and Trailing Edge (TE) Modes

Figure 7-2: Geometry input parameterization used by NeuralFoil. Parameterization is an 18-parameter CST (Kulfan) parameterization [17–19]. Each colored line in the figure represents a mode shape associated with one of these parameters; modes are linearly combined to form the airfoil shape.

Control Surfaces

Control surfaces are handled as a degenerate problem by re-normalizing the deflected airfoil shape. This is illustrated in Figure 7-3. In step 1, an example airfoil is given. In step 2, a user-specified control surface deflection is applied. In step 3, the airfoil is re-normalized by applying the necessary similarity transformation (rotation, translation, and scaling) such that the leading-edge and trailing-edge locations are placed at their standard $(0,0)$ and $(1,0)$ locations in chord-normalized airfoil coordinates. The geometric rotation required for this re-normalization is later applied as a change in the effective angle of attack, $\Delta\alpha$. For consistency, the scaling factor required for this operation is also later used to scale the input Reynolds number appropriately, though the effect of this Reynolds scaling is typically minor. Finally, the resulting pitching moment must also be adjusted to account for the shifting of the force center due to translation during re-normalization.

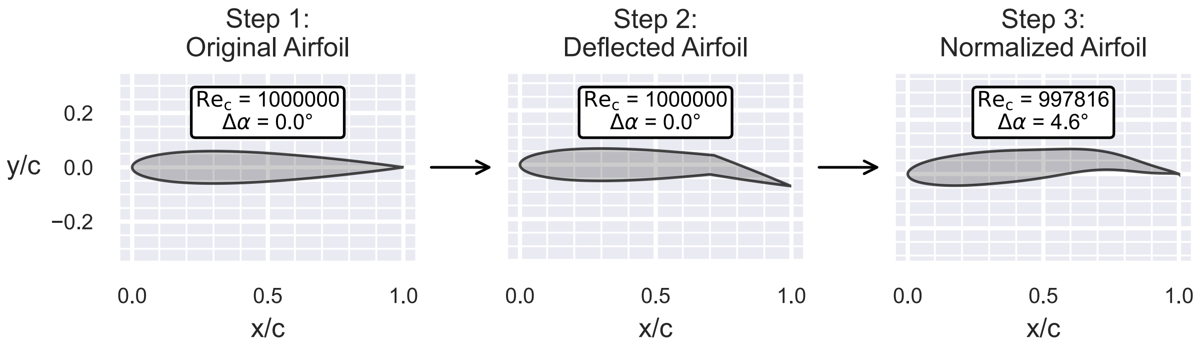


Figure 7-3: Illustration of the automatic procedure for handling control surface deflections in NeuralFoil.

Also visible in Figure 7-3 is the geometric effect of restricting the airfoil shape to the space of CST-parameterized airfoils, which is performed along with the re-normalization step between steps 2 and 3. The effect of this is that the sharp corners associated with the control surface deflection are smoothed out, which is a consequence of the smooth mode shapes associated with the CST parameterization. The rationale behind accepting this loss of fidelity here is that the control surface deflection invariably causes a turbulent transition at the deflection point on the suction side, regardless of whether the sharp or smoothed geometry is used. Because the turbulent boundary layer that follows the hinge is less sensitive to the pressure distribution (and hence, airfoil shape), the loss of geometric accuracy is less significant. Nevertheless, some loss of aerodynamic accuracy is to be expected.

To quantify this loss of accuracy, Figure 7-4 shows airfoil aerodynamics results obtained by both NeuralFoil and XFoil for various control surface deflections. Notably, reasonably close agreement is seen even for control surface deflections as aggressive as $\pm 40^\circ$.

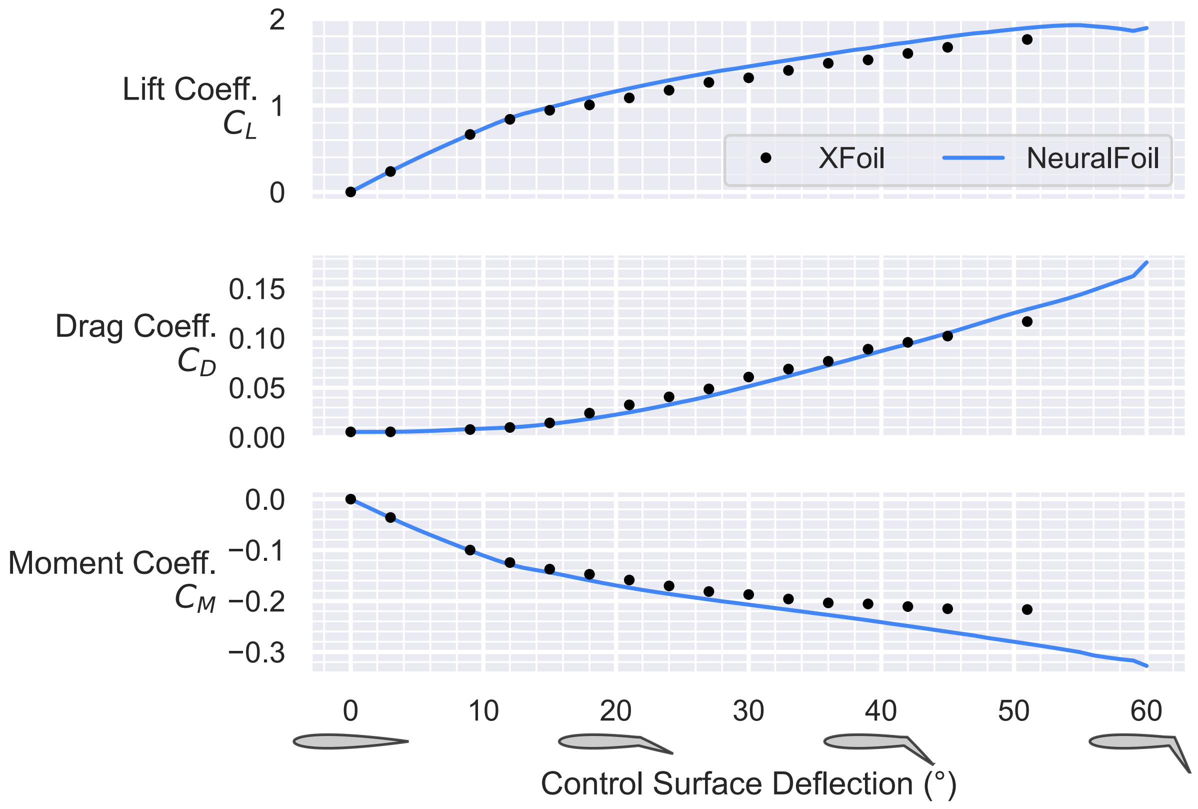


Figure 7-4: Accuracy of NeuralFoil on an airfoil with large control surface deflections. Here, we show aerodynamics results from both NeuralFoil and XFoil. All runs are on a NACA0012 airfoil at $\text{Re}_c = 10^6$, $M_\infty = 0$, and $\alpha = 0°$, with varying control surface deflections on a trailing-edge flap hinged at $x/c = 0.70$.

7.2.3 Encoding, Learned Model Architecture, and Decoding

After geometry parameterization, NeuralFoil transforms the user-facing inputs and outputs shown in Figure 7-1 into intermediate vector spaces (latent spaces) that are more amenable to learning by a neural network. These latent spaces are carefully parameterized such that the learned model has a close-to-affine mapping of inputs to outputs. Effectively, this is feature engineering using domain-specific knowledge.

This encoding/decoding scheme has two major effects. First, it substantially increases the model's parameter efficiency[6], since the training data has fewer nonlinearities in this latent space. Secondly,

[6] i.e., test-set accuracy, relative to the number of parameters (which represent model complexity and computational cost)

the combination of encoding functions, model architecture, and decoding functions can be used to guarantee physically-sound extrapolation beyond the dataset [219]. An example that illustrates both effects is the encoding of both the Re_c input and the C_D output into logspace. This means that an affine model in the latent space corresponds to a power-law $C_D(\text{Re}_c)$ model in user-facing space, which is a relationship supported by physical theory for both laminar and turbulent flows (e.g., Falkner-Skan and Schlichting boundary layer models [213]).

Encoding

The user-facing input space (shown in Figure 7-1) is transformed into the following input latent space, which is effectively what is seen by the neural network:

$$z_{\text{in}} = \text{Affine}\left(\begin{bmatrix} \text{Airfoil shape (18 parameters)} \\ \sin(2\alpha) \\ \sin^2(\alpha) \\ \cos(\alpha) \\ \ln(\text{Re}_c) \\ N_{\text{crit}} \\ x_{\text{tr,top,forced}} \\ x_{\text{tr,bot,forced}} \end{bmatrix} \right) \tag{7.1}$$

The resulting input latent space z_{in} is 25-dimensional. Note that the inputs described in Equation 7.1 also undergo a simple elementwise affine transformation. This transformation is such that the distribution of typical inputs in the latent space has a mean of roughly zero and a standard deviation of roughly one[7]. Exact scaling and shift factors are available in the open-source codebase described in Section 7.5. The purpose of this transformation is to improve the stability of the neural network training process, as well as to improve the performance of the weight-decay-based regularization strategy that is later used during training to improve generalization (described in Section 7.2.6).

In addition to the previously-discussed log-space transformation of the Reynolds number, some nonlinear transformations on the angle of attack α are present and merit discussion as well. Encoding of the angle of attack into a purely-trigonometric representation embeds the periodic nature of the problem into the model architecture. For example, model evaluation at $\alpha = 0^\circ$ and $\alpha = 360^\circ$ are structurally identical, as these are encoded to the same location in the input latent space. The $\sin(2\alpha)$ and $\sin^2 \alpha$ terms are directly proportional to common post-stall analytical models of lift and drag, respectively (such as those by Hoerner and Truong [218, 220]). The goal of this representation is to improve generalization performance in massively-separated flow conditions, where training data is scarce.

The freestream Mach number is a notable omission from the input latent space, and the learned model is both trained and evaluated on the equivalent incompressible flow. A compressibility correction is then performed after the neural network evaluation, as described in Section 7.2.4. This dimensionality reduction was performed to keep the required amount of training data more manageable, and because analytical models can accurately and quickly perform this compressibility correction (up to the critical Mach number).

Learned Model Architecture

After encoding inputs into an input latent space, NeuralFoil processes these inputs through a feedforward neural network (i.e., a multilayer perceptron). NeuralFoil offers eight different deep neural network

[7] This normalization is performed on the basis of the training data, which is described later in Section 7.2.5.

models, which offer a tradeoff between accuracy and computational cost. This tradeoff is implemented as differences in the number and size of hidden layers, which are detailed in Table 7.1 for each model.

Table 7.1: Neural network model sizes offered in NeuralFoil, offering a trade between accuracy and speed.

Model	Hidden Layers	Neurons per Hidden Layer (width)
"xxsmall"	1	48
"xsmall"	2	48
"small"	2	64
"medium"	3	64
"large"	3	128
"xlarge"	4	128
"xxlarge"	4	256
"xxxlarge"	5	512

Each layer is a fully-connected linear layer. The activation function applied between each layer is the Swish function[8], defined as $\sigma(x) = x/(1 + e^{-x})$. Compared to typical machine learning scenarios, the choice of activation function here has surprising importance, due to the desired properties of the resulting model. The Swish activation function is smooth (C^∞-continuous), which makes the resulting neural network a smooth function as well. By preserving this continuity, NeuralFoil is made much more amenable to later gradient-based design optimization.

Likewise, the asymptotic behavior of the Swish function has key implications. Networks with activation functions that asymptote[9] to piecewise-linear functions with differing positive and negative

[8]sometimes written with an optional parameter β, which we set as $\beta = 1$

[9]When we refer to the *asymptotic behavior* of a function here, we're referring to the function's behavior when evaluated at some $\vec{x}$, in the limit $|\vec{x}| \to \infty$ in any direction. Colloquially, this refers to the function's behavior when one "zooms out" enough such that the nonlinearities occupy a negligble portion of the domain.

slopes (e.g., Swish, ReLU, LeakyReLU, Softplus) have fundamentally different extrapolation properties than networks with activation functions that asymptote to a constant (e.g., Sigmoid, Tanh). The former creates networks that asymptote to locally-affine functions, while the latter creates networks that asymptote to locally-constant functions. This observation about network extrapolation properties, discussed in greater detail by Xu et al. [219], forms part of the justification for the latent space encodings described in the previous section where affine extrapolation is physically-consistent. Other activation functions were also considered. For example, the Softplus activation function meets many of the aforementioned criteria, but here it was abandoned in favor of Swish because networks with the latter achieved slightly better generalization performance.

Considerations during training of this network are discussed in Section 7.2.6.

Embedding of Angle of Attack Symmetry

A key step during neural network evaluation (applied both during training and inference) is the embedding of physics symmetry with respect to angle of attack. The rationale behind this symmetry can be intuitively explained with the aid of the illustrations in Figure 7-5. Here, we consider a generic cambered airfoil at some angle of attack, as well as its "image scenario" consisting of the flipped airfoil at a flipped angle of attack. The flow physics in these two scenarios should be exactly equal and opposite, which will not generally be the case unless this is enforced. To enforce this symmetry, NeuralFoil computes the neural network output for both the original and flipped airfoil, and then merges the results. Outputs that are physically-guaranteed to exhibit even symmetry (e.g., C_D) are averaged, while outputs that exhibit odd symmetry (e.g., C_L, C_M) are subtracted. More details about this approach to embedding symmetries and invariants into neural networks are discussed in recent work by Zhang [184].

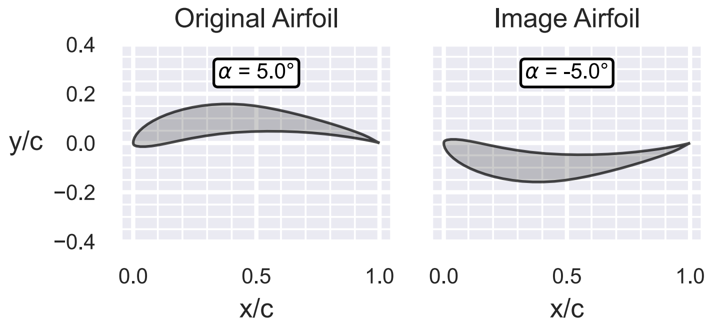


Figure 7-5: Illustration of the "image" approach used to structurally embed symmetry with respect to angle of attack in NeuralFoil.

This symmetry embedding has important impacts for end-use cases of NeuralFoil. For example, when NeuralFoil analyzes a symmetric airfoil at $\alpha = 0^\circ$, it will always yield a lift coefficient and pitching moment of *exactly* zero[10], and upper- and lower-surface transition locations will also be exactly equal. This exactness would not be the case if other common methods, such as data duplication to learn physics symmetries, were used. A practical case where error in this symmetry would be particularly noticeable is on an aircraft's vertical stabilizer. Here, flows around symmetric airfoils at zero local incidence are common, and any error in enforcing this symmetry would lead to a noticeable nonzero net yaw moment on the aircraft.

Decoding

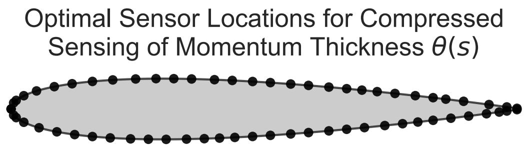
The raw result of the learned model is another latent-space vector, which is then transformed back into the user-facing output space. This decoding process is broadly motivated by the same considerations as the encoding process, where the goal is to embed physical intuition into the architecture. The output latent space consists of the following vector, shown here as a function of the user-facing outputs:

[10] to within machine precision, with differences only due to compensated summation algorithms during matrix multiplication

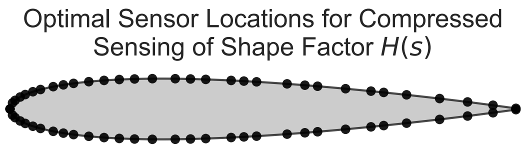
$$z_{\text{out}} = \text{Affine}\left(\begin{bmatrix} \text{Analysis Confidence} \\ C_L \\ \ln(C_D) \\ C_M \\ x_{\text{tr,top}} \\ x_{\text{tr,bot}} \\ \ln(\text{Re}_\theta) \text{ at 64 locations} \\ \ln(H) \text{ at 64 locations} \\ u_e/u_\infty \text{ at 64 locations} \end{bmatrix}\right) \tag{7.2}$$

The output latent space has 195 dimensions, and, similar to the input latent space, undergoes an elementwise affine transformation to normalize the distribution of typical outputs. The "analysis confidence" output described in Equation 7.2 deserves special attention and is discussed in Section 7.2.4.

The boundary layer outputs in Equation 7.2 are computed at 64 locations along the airfoil surface, which are evenly spaced in the normalized x-coordinate on the top and bottom surfaces. These data points are conceptually similar to physical sensors, as they are effectively a discrete projection of the actual boundary layer flow, which itself is a continuous function of the surface coordinate s. The spacing of these discrete "sensors" was chosen on the basis of a compressed sensing study, the results of which are shown in Figure 7-6. In short, this study aimed to determine "optimal sensor placement" for airfoil boundary layer data: where should one place 64 discrete sensors on an airfoil to minimize the error when reconstructing the boundary layer data? Data for this study was generated from XFoil via the same process detailed in Section 7.2.5. Compressed sensor placement was performed using approximate QR-factorization-based methods described by de Silva et al. [221]


Optimal Sensor Locations for Compressed Sensing of Momentum Thickness $\theta(s)$

(a) Optimal sensor placement for minimum-error $\theta(s)$ reconstruction.


Optimal Sensor Locations for Compressed Sensing of Shape Factor $H(s)$

(b) Optimal sensor placement for minimum-error $H(s)$ reconstruction.

Figure 7-6: Results from a compressed sensing study to determine optimal discrete locations to track boundary layer data for minimal reconstruction error.

The results in Figure 7-6 show that accurate reconstruction of the momentum thickness $\theta(s)$ is best achieved by roughly-uniform sensor placement. Interestingly, the optimal placement for the shape factor $H(s)$ is not uniform, but rather is concentrated near the leading edge. Since H closely relates to laminar/turbulent behavior and transition location, this result is physically intuitive: small changes in H near the leading edge have a disproportionate impact on the downstream flow.

7.2.4 Uncertainty Quantification, Model Fusion, and Extrapolation

The "analysis confidence" output detailed in Equation 7.2 can be loosely interpreted as a representation of NeuralFoil's self-reported model uncertainty. Quantitatively, this output is trained on binary features corresponding to whether an XFoil analysis with these input conditions converged. In effect, this is a classification problem based on whether the surrogate's underlying model returns trustworthy answers near a given point in the input space. The raw network output is a logit, which is then transformed into a confidence score in the range $(0, 1)$ using a logistic function. Example results of this "analysis confidence" metric are shown and discussed in Section 7.3.2.

Presenting the user with a measure of analysis confidence has several real-world benefits. First, it gives the user direct feedback on when the result of a requested analysis should be trusted. Surrogate models, especially those created with machine learning, invariably lose accuracy when extrapolating beyond the training data. This can be mitigated somewhat with embedded physics constraints, but ultimately NeuralFoil is no exception here. Furthermore, most neural surrogates give no obvious information about when this spurious extrapolation is occurring. By providing a confidence score, NeuralFoil can flag these scenarios; this may encourage a user to cross-check with a higher-fidelity model or pursue a less-uncertain design. Second, the confidence score can be used as a direct constraint during aerodynamic shape optimization driven by NeuralFoil. This enables a form of robust optimization where results are guaranteed to be within the region of the input space where the surrogate model is trustworthy.

Learning this "analysis confidence" binary classifier with no priors or physics knowledge presents a problem, however. Without such modifications, there is no guarantee that the model will extrapolate towards "untrustworthiness" (i.e., low analysis confidence) when far from the training data distribution. To enforce this behavior, the analysis confidence logit (i.e., logarithm of odds-ratio) is modified during both training and inference. The added term is the negative squared Mahalanobis distance[11] of the query point with respect to the training data distribution:

$$\text{Analysis Confidence} = \sigma\left((\text{Raw Logit}) - (\vec{z}_{\text{in}} - \vec{\mu})^T S^{-1} (\vec{z}_{\text{in}} - \vec{\mu})\right) \tag{7.3}$$

where:Analysis Confidence = The final output of the analysis confidence model, in the range $(0, 1)$

σ = The logistic function, defined as $\sigma(x) = 1/(1 + e^{-x})$

$\vec{\mu}$ = The mean of the training data distribution in the input latent space

Raw Logit = The raw output of the neural network, before adding the Mahalanobis distance term

S = The covariance matrix of the training data distribution in the input latent

[11]essentially, the Euclidian norm of the multidimensional generalization of the z-score

space

$\vec{z}_{\text{in}}$ = The query point in the input latent space

Because the covariance matrix of the training data is positive-definite, the correction term is guaranteed to asymptote to $-\infty$ as the query point moves away from the training data in any direction. By contrast, the raw logit is structurally guaranteed to extrapolate to a locally-affine function, as previous discussed in Section 7.2.3. Therefore, this modification guarantees that the analysis confidence tends to zero far from the training data distribution. Because this modification is included at both train and inference time, this modification has minimal effect on the model's performance *within* the training data distribution, and only serves to embed desirable extrapolation properties when *outside* the distribution.

Another benefit of this modification is that it tends to make level sets of the analysis confidence more convex, since the modification is a quadratic form with a positive-definite Hessian[12]. By regularizing the analysis confidence to be "more convex", this modification ultimately leads to better convergence in downstream gradient-based optimization problems that use the analysis confidence as a constraint.

The weighting of this Mahalanobis distance term could be varied, depending on how willing the developer of a neural surrogate is to allow the surrogate to extrapolate beyond the training distribution. In the case of NeuralFoil, a unit weighting (as shown in Equation 7.3) was chosen and appears to strike a good balance between extrapolation correctness and the amount of added nonlinearity. However, more broadly, this tradeoff forms an interesting area of further research in neural surrogates with self-reported trust metrics.

Based on the analysis confidence of this learned model, NeuralFoil fuses its attached-flow results with empirical models for massively-separated (post-stall) flow conditions. Specifically, NeuralFoil uses the analytical post-stall models regressed by Truong [218] to provide a more accurate prediction in these

[12] directly proportional to the inverse of the covariance matrix

conditions. Example results showing this post-stall model fusion are given in Figure 7-7.

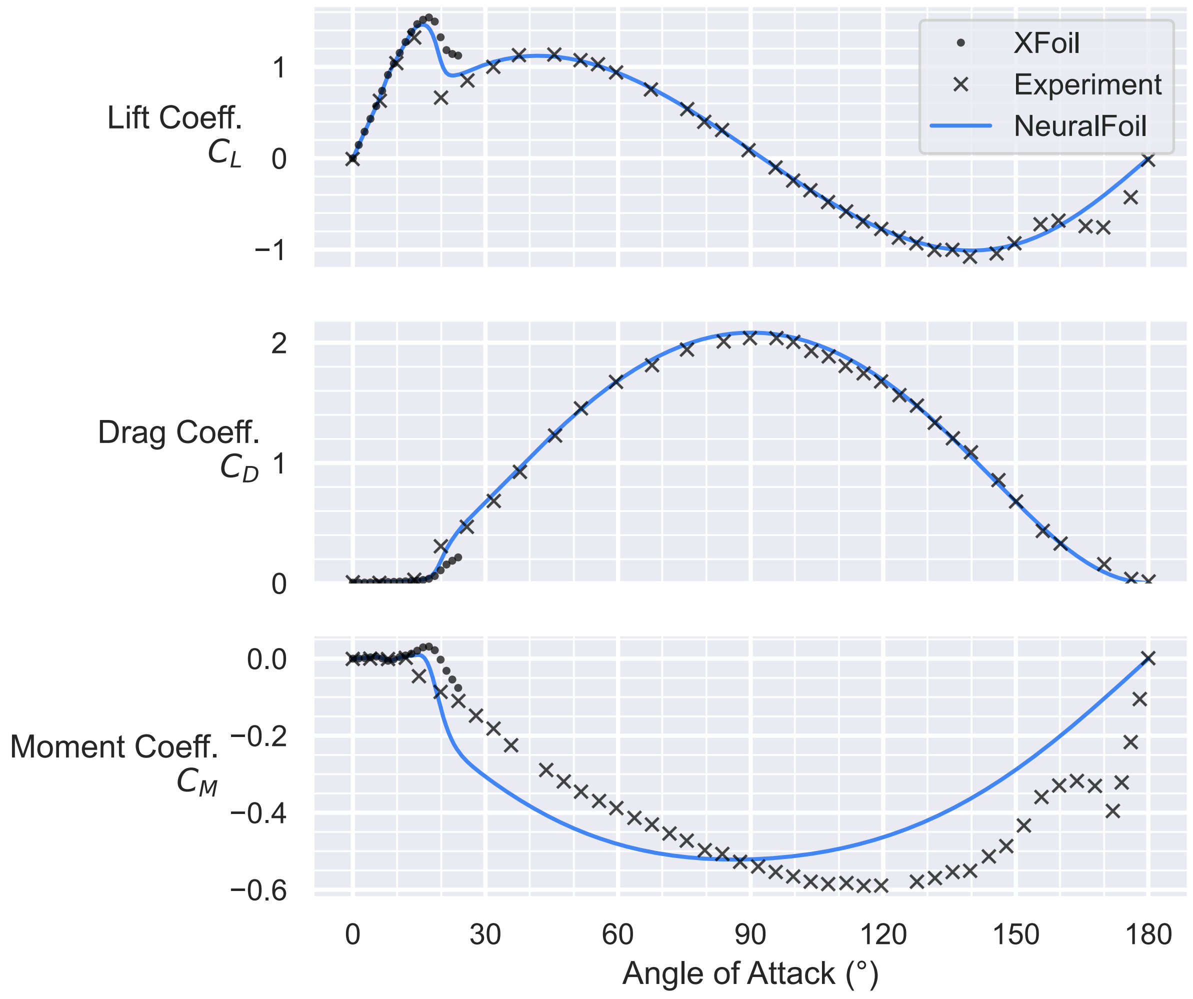


Figure 7-7: Illustration of the performance of NeuralFoil in massively-separated flow conditions. NACA0012 airfoil at $\mathrm{Re}_c = 1.8 \times 10^6$. Experimental data reproduced from Langley wind tunnel data in NACA TN 3361 [20]. NeuralFoil computes post-stall lift and drag quite accurately; moment computation is somewhat less accurate.

The NeuralFoil analysis results have some notable deviations from experiment. First, moment computation is somewhat less accurate than lift and drag computation. This is not particularly surprising, as aerodynamic moment computation is inherently more challenging than force computation: moment computation requires one to know not only the general *magnitude* of forces, but the *distribution* of such forces. Secondly, NeuralFoil does not capture reversed-flow reattachment near $\alpha = 180°$.

Nevertheless, lift and drag results are relatively accurate, and results follow physically-sensible trends. In cases where post-stall data would be used (e.g., flight simulation, real-time control, helicopter blade stall, or recovering from a bad initial guess during design optimization), a less-accurate-but-still-reasonable answer may be useful. The model fusion shown in Figure 7-7 is also smooth, facilitating later gradient-based optimization.

Handling of Compressibility Effects

At a high level, compressibility is handled by applying a correction factor to the incompressible results. This correction factor is computed using Laitone's rule [222], which is a higher-order variant of the well-known Prandtl-Glauert and Karman-Tsien compressibility corrections. All of these compressibility corrections take two inputs: the pressure coefficient in the equivalent incompressible flow C_{p_0} and the Mach number M; they return the pressure coefficient in the actual compressible flow C_p. For comparison, all three corrections are given here, which conveniently show the higher-order terms retained in each successive relation:

$$\text{Prandtl-Glauert:} \qquad C_p = \frac{C_{p_0}}{\beta} \tag{7.4}$$

$$\text{Karman-Tsien:} \qquad C_p = \frac{C_{p_0}}{\beta + M_\infty^2/(1+\beta) \cdot C_{p_0}/2} \tag{7.5}$$

$$\text{Laitone [222]:} \qquad C_p = \frac{C_{p_0}}{\beta + M_\infty^2/\beta \cdot C_{p_0}/2 \cdot \left(1 + \frac{\gamma-1}{2} M_\infty^2\right)} \tag{7.6}$$

where $\beta = \sqrt{1 - M_\infty^2}$, and γ is the ratio of specific heats (1.4 for air near standard conditions). In theory, these compressibility corrections should apply only to the pressure-derived forces on the airfoil, while the shear forces are relatively unaffected. To approximate this, NeuralFoil applies the compressibility correction to the lift force and pitching moment (which are pressure-dominated) but does not apply the correction to the drag force (which is often shear-dominated). This assumption, while relatively simple, proves to match compressible airfoil drag data computed with other methods quite closely, as shown in Section 7.3.5.

Another useful definition is that of the sonic pressure coefficient $C_{p_{\text{crit}}}$, which is the pressure coefficient below which flow goes supersonic. This is derived from the isentropic relations, yielding:

$$C_{p_{\text{crit}}} = \frac{2}{\gamma M_\infty^2} \left(\left(\frac{1 + \frac{\gamma - 1}{2} M_\infty^2}{1 + \frac{\gamma - 1}{2}} \right)^{\frac{\gamma}{\gamma - 1}} - 1 \right) \tag{7.7}$$

At the location where supersonic flow first begins at M_{crit}, Equations 7.6 and 7.7 can be set equal. This yields a relation that maps the minimum value of the incompressible pressure coefficient $C_{p_{\text{min}}}$ to the critical Mach number M_{crit}. This relation does not admit a closed-form solution, but it can be solved numerically; this is shown in Figure 7-8.

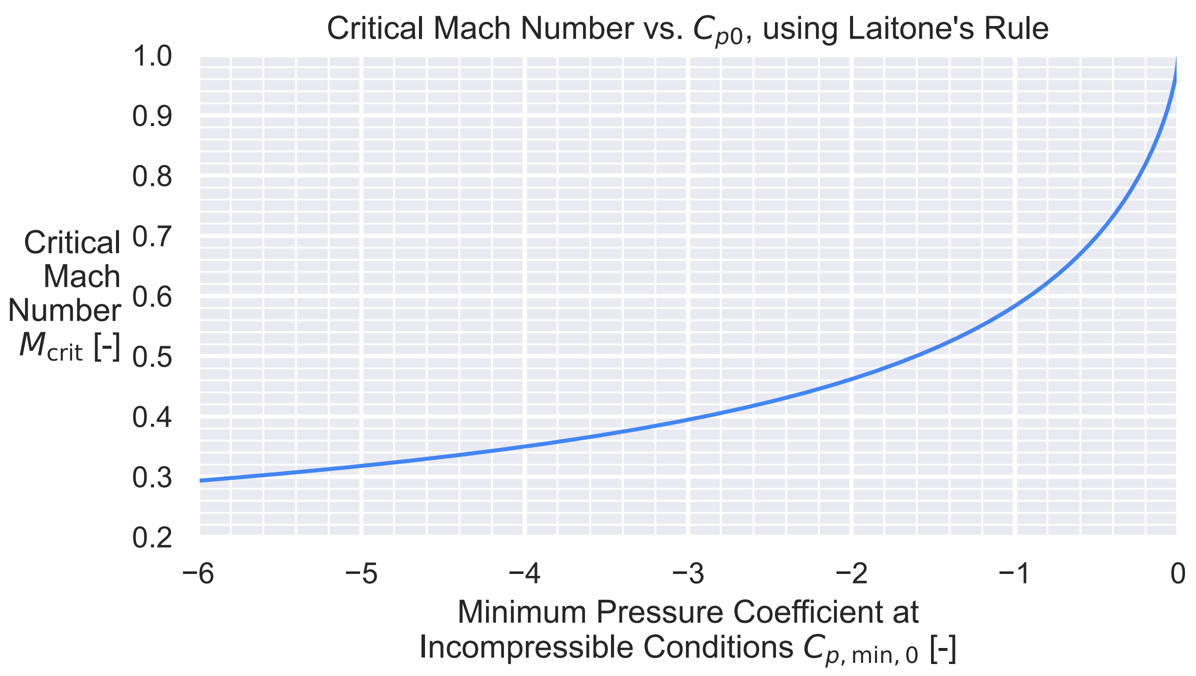


Figure 7-8: Laitone's rule allows a mapping from the minimum incompressible pressure coefficient $C_{p_{0,\min}}$ to the critical Mach number M_{crit}.

This implicit mapping via a nonlinear solve is less desirable at runtime, for two main reasons. First, this method is iterative and does not have a static computational graph, complicating compatibility with automatic differentiation tools. Secondly, it is not guaranteed to converge, depending on the initial guess. Instead, this relation can be replaced with an explicit surrogate model, which was obtained using symbolic regression (via PySR [223]):

$$M_{\text{crit}} = \left(1.0083619 - C_{p_{0,\min}} + \left(-0.51058894 \cdot C_{p_{0,\min}}\right)^{0.6553655}\right)^{-0.5536965} \tag{7.8}$$

Replacing the implicit mapping of Figure 7-8 with the surrogate model of Equation 7.8 introduces negligible error, with a corresponding mean RMS error in M_{crit} of 0.0014 over a representative range of $C_{p_{0,\min}}$ values[13]. This relation allows NeuralFoil to estimate the critical Mach number M_{crit} relatively

[13]More precisely: finding the mean error with respect to Mach in the interval $M \in [0.001, 0.999]$.

accurately using only incompressible quantities, as shown later in Section 7.3.5 and Table 7.4 relative to full-potential and RANS solutions. For the purposes of Equation 7.8, the incompressible $C_{p_{0,\text{min}}}$ is computed definitionally from surface values of velocity magnitude, as reported by the learned model:

$$C_{p_{0,\text{min}}} = 1 - \left(\frac{u_{\text{max}}}{u_\infty} \right)^2 \qquad \text{at } M_\infty = 0 \tag{7.9}$$

Following a derivation from Mason [224], an empirical relation for the shape of the drag rise beyond M_{crit} is included. Drag in this transonic regime, especially beyond the drag-divergent Mach number M_{dd}, is relatively simple and typically errs on the side of over-estimating wave drag. Thus, NeuralFoil's predictions of *when* transonic flow will occur are reasonably trustworthy, but wave drag results deep within this transonic regime are not. However, given that a primary goal of the NeuralFoil tool is to drive design optimization, this simple empirical model achieves its purpose of steering an optimizer away from thick transonic airfoils with strong shocks.

7.2.5 Training Data Generation

Synthetic training data was generated by running XFoil on randomly-generated airfoil shapes and flow conditions. All training data is analyzed without compressibility in XFoil (i.e., $M_\infty = 0$); compressible effects were handled outside of the neural network training process, as described in Section 7.2.4. The stochastic procedure used to generate the airfoil shapes used in training can be described as follows:

1. First, three parent airfoils are randomly selected from an airfoil database[14], and converted to the Kulfan geometry parameterization. Three random weights are drawn, and the airfoils are merged

[14]This database consists of 2,174 airfoils, which are drawn from a variety of sources (most notably, the UIUC airfoil database [225] and TraCFoil database [226]) and manually cleaned. The database is publicly available via AeroSandbox [1].

into one based on these weights. This procedure effectively ensures that each training airfoil is derived from a unique combination of three parent airfoils.

2. To increase the diversity of training data, several further modifications are applied:
    - The thickness is randomly scaled by a factor drawn from Lognormal$(\mu = 0,\ \sigma = 0.15)$.
    - The airfoil's Kulfan parameters are perturbed by a random vector drawn from a multivariate normal distribution. The mean and covariance of this distribution are taken from the sample statistics of the airfoil database.
3. Flow conditions are randomly selected. Angle of attack α is drawn from the sum of a uniform and normal distribution, and Reynolds number Re_c is drawn from a log-normal distribution. Distributional statistics for these variables and transition criteria are given in Table 7.2.

Table 7.2: Summary statistics of flow conditions in the overall dataset, which is later partitioned into separate training and test datasets. Percentages refer to percentiles of the distribution. Strictly speaking, this table gives summary statistics for the subset of generated cases that were later associated with a successfully-converged XFoil run, so the true distribution of the generator is slightly wider. Suffixes 'k', 'M', and 'T' denote 10^3, 10^6, and 10^{12}, respectively.

	Minimum	**2.5%**	**25%**	**50% (Median)**	**75%**	**97.5%**	**Maximum**
Angle of attack α	$-27.9°$	$-17.3°$	$-7.5°$	$+0.9°$	$+8.7°$	$+17.6°$	$+28.6°$
Reynolds number Re_c	0.916	1.87k	31.1k	296k	3.47M	262M	2.92T
Critical amplification factor N_{crit}	Uniformly distributed in $[0, 18]$.						
Transition locations $x_{\text{tr,forced}}/c$	With 80% probability, natural transition. Otherwise, uniform in $[0, 1]$.						

In total, 7,913,292 data points (i.e., airfoils and flow conditions) were generated with this procedure and analyzed with XFoil. Cases that did not result in converged XFoil solutions were not used to train physics outputs; however, this non-convergence was still recorded as a binary feature to train the analysis confidence output described in Section 7.2.4. Overall, 56% of generated cases resulted in converged XFoil solutions, which suggests that this data-generating process does a reasonable job of spanning the space of inputs where XFoil convergence is likely. XFoil results that were purportedly converged but violated physical constraints (e.g., $\theta < 0$, $H < 1$, non-physical u_e values) were also automatically filtered and treated as unconverged, though this was rare and affected only 0.03% of cases. The distribution of training data points was sufficiently diverse that converged XFoil results were obtained with lift coefficients ranging from -2.67 to $+3.44$.

It is not currently known whether the number of points in this dataset is too little, too much, or appropriate. As a point of comparison, the number of data points used to train NeuralFoil is roughly two orders of magnitude larger than that of similar studies (see Table 7.5), although NeuralFoil's data distribution is substantially wider, and hence this may be warranted. In total, data generation required roughly 24 hours on MIT Supercloud, a computing cluster operated by the Lincoln Laboratory Supercomputing Center. It is possible either that similar performance could be achievable with much less data, or that more data could further improve performance. This is left as an area of future research.

7.2.6 Training Process

The neural networks at the heart of this approach were trained with MIT Supercloud's GPU computing capabilities. The training process was implemented using the PyTorch [117] framework. The RAdam optimizer [227] was used with a decaying learning rate scheduled based on the plateau of the training loss.

Synthetic data generated using the procedure described in Section 7.2.5 was split into a training dataset (95%) and a test dataset (5%), with the latter used only to evaluate generalization accuracy.

Critically, a small amount of weight decay (effectively, a L^2-norm penalty on all weight and bias parameters) was added to the loss function, which improves generalization performance and causes the network training to asymptote without over-fitting. Other techniques to improve generalization, such as batch normalization and dropout, were tested; however, it was found that weight decay alone consistently produced the models with the most accurate generalization to the test dataset. Various hyperparameters associated with training the model were optimized via a parallelized grid search. Because the weight decay regularization effectively limits the amount of overfitting that is possible, all models were trained until the test-set loss reached an asymptote.

The loss function used when training the network is a Huber loss metric on each of the latent-space outputs shown in Equation 7.2. For this Huber loss, the L^1-to-L^2 transition point is at $\delta = 0.05$ (recalling that one unit in the output latent space corresponds roughly to the training data's output standard deviation). The exception to this Huber loss metric is the analysis confidence output, which is instead converted to a binary cross-entropy loss. Exact training details are publicly-available as described in Section 7.5.

Individual components of the loss function are weighted differently, to reflect differing importance during typical analysis and optimization. The highest weight is applied to drag, followed in descending order by lift, moment, and transition locations. The relative weightings of such parameters were optimized as a hyperparameter, with the goal of minimizing generalization error (i.e., test-set performance) of drag estimation. Interestingly, this error appears minimized not by tilting the weighting entirely towards drag, but by also adding small but significant weights to other outputs, such as the detailed boundary layer. Stated more informally, this suggests that the best way to compute drag is to learn a general model that estimates all physical outputs, rather than a model focused purely on drag. This suggests that the model learns deeper physics relationships than simple memorization, as the presence of intermediate representations results in improved generalization performance on bulk quantities.

7.3 Results and Discussion

7.3.1 Point Validation of NeuralFoil Accuracy with respect to XFoil

Qualitatively, NeuralFoil tracks XFoil very closely across a wide range of angles of attack and Reynolds numbers. In Figure 7-9, we compare the performance of NeuralFoil to XFoil on aerodynamic polar prediction. This study aims to evaluate the generalization performance: to what extent is the model learning physics vs. merely memorizing its training data? This generalization performance forms an important metric of usefulness for real-world design problems.

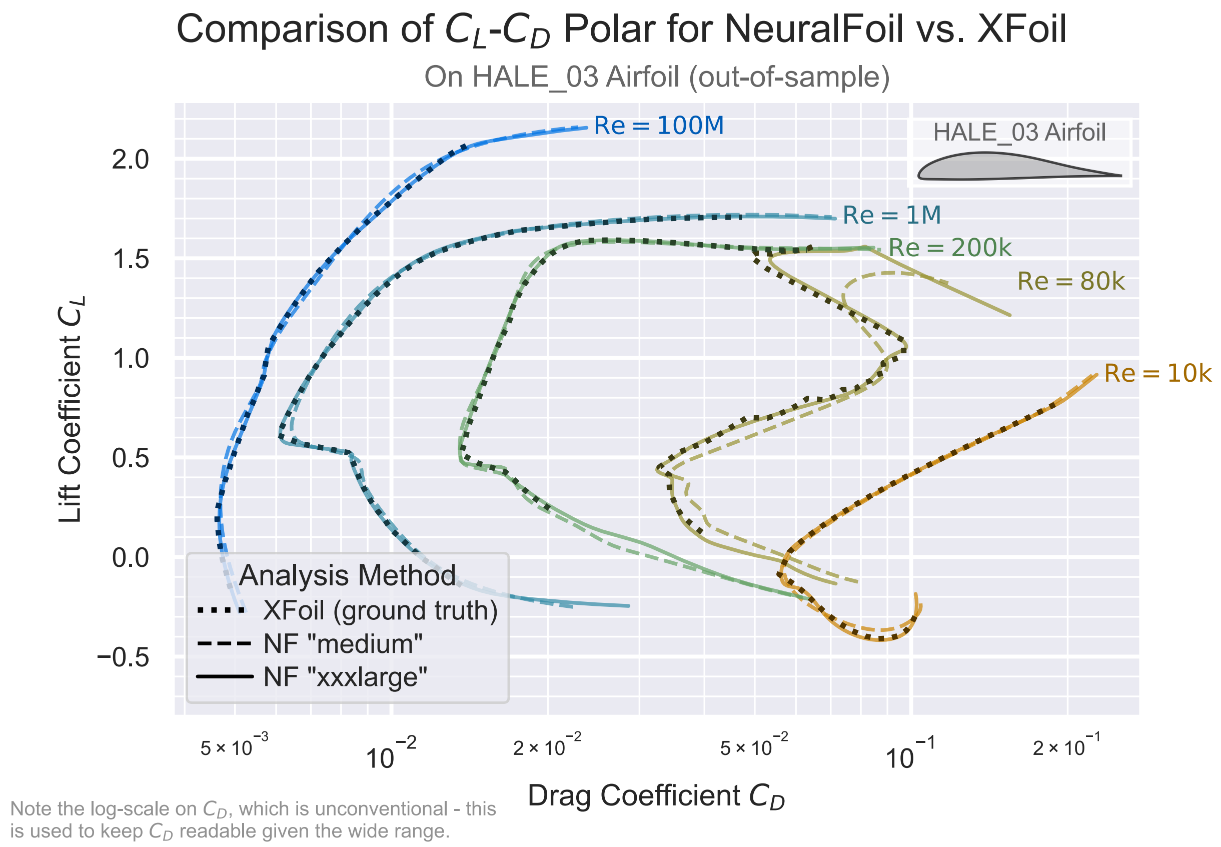


Figure 7-9: Generalization performance of NeuralFoil, assessed by sample validation with repsect to XFoil on an out-of-distribution airfoil. Each colored line represents an analysis at a different Reynolds number. Results are for incompressible, viscous flow with $N_{\text{crit}} = 9$ and natural transition. Solid lines represent NeuralFoil ("NF") analyses, which are given for two models with different accuracy-speed tradeoffs. The dotted line represents XFoil results at the respective Reynolds number. Agreement with XFoil is almost exact for the "xxxlarge" model, across a broad range of angles of attack and Reynolds numbers.

The airfoil analyzed here was developed separately for a real-world aircraft development program (*Dawn One*, a high-altitude long-endurance aircraft [13, 158]), using an inverse design process. This airfoil was not included in the training data, either directly or within the "parent airfoil" database described in Section 7.2.5. Because of this, this represents a previously-unseen airfoil for NeuralFoil, so no unfair advantage is gained by memorization. The airfoil geometry itself is also shown in Figure 7-9.

In this study, aerodynamic performance is assessed across a range of Reynolds numbers spanning four orders of magnitude. The amplification factor is set as $N_{\text{crit}} = 9$ and natural transition is used. All cases are run in the incompressible limit ($M_\infty = 0$).

Figure 7-9 shows that excellent agreement is achieved between NeuralFoil and XFoil. The results are nearly exact for the "xxxlarge" model, which is the most accurate and computationally expensive model included with NeuralFoil. The "medium" model achieves slightly reduced accuracy while offering considerable speed improvements, as shown in Table 7.3.

The Reynolds numbers shown in Figure 7-9 were chosen to illustrate accuracy in a challenging flow condition that occurs near $\text{Re} = 80 \times 10^3$ for this airfoil. Here, the upper-surface boundary layer becomes extremely delicate, as illustrated by the sudden, discontinuous jump near $C_L = 1.0$. At angles of attack below this jump, the boundary layer simply undergoes laminar separation and never reattaches. At angles of attack above this jump, a laminar separation bubble (LSB) is formed, which undergoes turbulent reattachment before eventually separating again farther downstream. This specific effect only occurs in a narrow window of airfoil shapes, Re_c, and N_{crit} near the values shown in Figure 7-9, which forces the network to rely on learned physics. Nevertheless, this phenomenon is well-captured by the "xxxlarge" NeuralFoil model, demonstrating its generalization performance. Another benefit of NeuralFoil shown in Figure 7-9 is the inherent C^∞-continuity of its solutions, which smooths out XFoil's "jagged" predictions in regions such as the aforementioned discontinuity.

7.3.2 Self-Reported Uncertainty Quantification

As discussed in Section 7.2.4, NeuralFoil self-reports an analysis confidence metric that is intended to aid the user in assessing surrogate trustworthiness. This metric is shown in Figure 7-10 for the same airfoil and flow conditions as in Figure 7-9. Notably, regions with uncertain flow features are immediately flagged. This includes obvious regions of uncertainty, such as post-stall, but also more subtle ones, such as the region near $\text{Re}_c = 80 \times 10^3$ and $C_L = 1.0$. Another subtle example is a small region in the $\text{Re}_c = 100 \times 10^6$ case near $C_L = 1.0$, where movement of the stagnation point causes sudden changes in the bottom-surface transition location, and thus greater uncertainty.

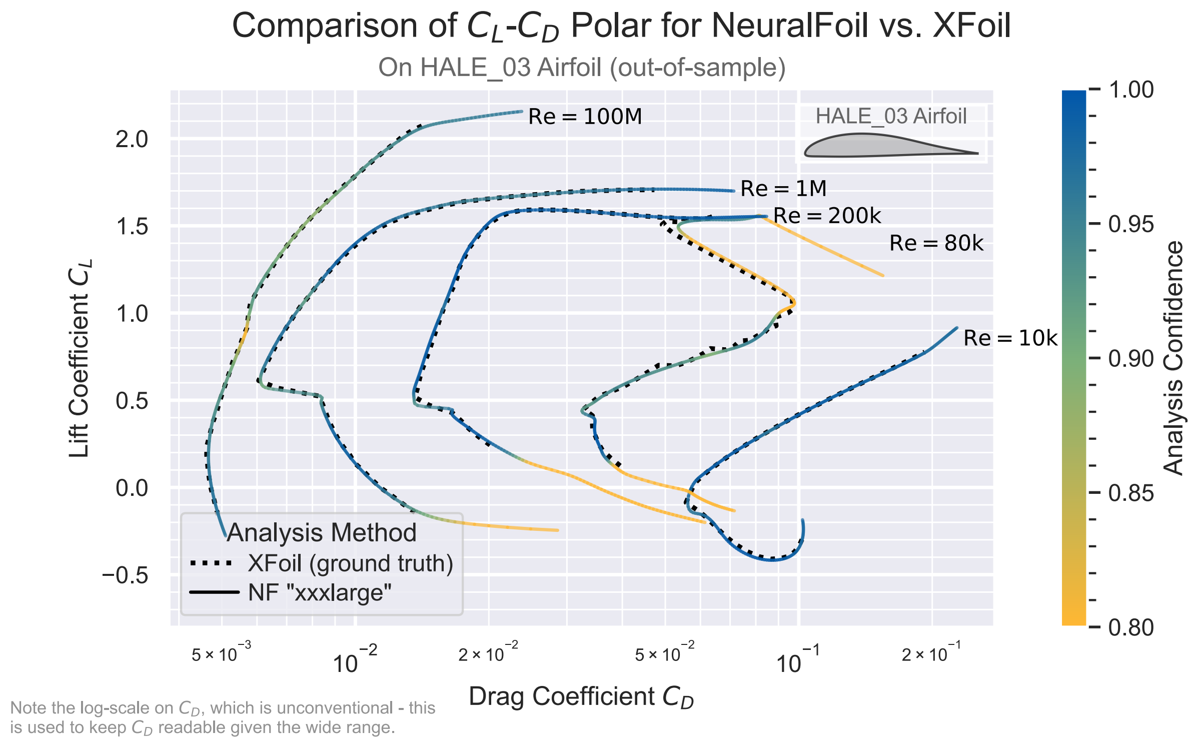


Figure 7-10: Demonstration of NeuralFoil's self-reported "analysis confidence" metric, shown in an identical analysis setup as Figure 7-9. In this figure, color represents the value of the analysis confidence metric, which varies throughout the input space.

7.3.3 NeuralFoil Accuracy on the Test Dataset

To report the accuracy of NeuralFoil on a broader range of airfoil shapes and flow conditions, we can measure performance on the test dataset. This dataset was generated using the same methods described in Section 7.2.5, but was not used during training. In total, this test dataset consists of 395,665 cases.

The analysis accuracy on this test dataset is shown in Table 7.3. At a basic level, the figures of merit are accuracy (here, treating XFoil as a "ground truth") and computational speed. This table details both considerations. The first set of columns shows the error with respect to XFoil on the test dataset. The second set of columns gives the runtime speed of the models, both for a single analysis and for a large batch analysis. Here, the benefit of NeuralFoil's vectorization is shown: for large batch analyses, runtimes many orders of magnitude faster than XFoil are possible, with minimal loss in accuracy.

Table 7.3: Performance comparison of NeuralFoil ("NF") physics-informed machine learning models versus XFoil in terms of accuracy (treating XFoil as a ground truth) and speed, as measured on the test dataset of Section 7.2.5. Runtime speeds are measured on an AMD Ryzen 7 5800H laptop CPU.

Aerodynamics Model	**Mean Absolute Error (L_1-norm) of Given Metric, on the Test Dataset, with respect to XFoil**				**Computational Cost to Run**	
	Lift Coeff. C_L	Fractional Drag Coeff. $\ln(C_D)$ †	Moment Coeff. C_M	Transition Locations x_{tr}/c	Runtime (1 run)	Total Runtime (100,000 runs)
NF "xxsmall"	0.041	0.079	0.0072	0.038	4 ms	0.94 sec
NF "xsmall"	0.030	0.057	0.0053	0.027	5 ms	0.96 sec
NF "small"	0.027	0.049	0.0046	0.023	5 ms	1.14 sec
NF "medium"	0.020	0.039	0.0034	0.017	6 ms	1.34 sec
NF "large"	0.016	0.028	0.0025	0.011	9 ms	2.35 sec
NF "xlarge"	0.014	0.025	0.0022	0.009	11 ms	2.77 sec
NF "xxlarge"	0.012	0.021	0.0018	0.007	18 ms	4.81 sec
NF "xxxlarge"	0.011	0.020	0.0016	0.005	61 ms	12.42 sec
XFoil	0	0	0	0	73 ms	2553 sec

† The deviation of $\ln(C_D)$ can be thought of as "the typical relative error in C_D". (E.g., $0.020 \rightarrow 2.0\%$ error.)

7.3.4 Accuracy-Speed Tradeoff vs. XFoil

When developing machine learning surrogate models based on conventional physics solvers, far too often in the literature a speedup is claimed without controlling for accuracy. After all, it is typically trivial to make the conventional physics solver faster by sacrificing accuracy (e.g., by changing the level of discretization). Therefore, any fair and meaningful speed comparison between a machine learning surrogate and a conventional solver must control for accuracy.

Figure 7-11 gives this speed-accuracy tradeoff for NeuralFoil, compared to the most common conventional alternative, XFoil. For NeuralFoil, this tradeoff is controlled by the model size (as described in Table 7.1). For XFoil, this tradeoff is controlled by the number of points used to discretize the airfoil, which was varied from 20 to 260 in Figure 7-11.

In this study, each model is evaluated on a range of airfoils and flow conditions. To minimize issues with non-convergence in XFoil, a new range of aerodynamic cases is used. The airfoil shapes are NACA 4-series airfoils with the location of maximum camber fixed at $x/c = 0.4$. Thickness is varied in the range $[0.08, 0.16]$ and maximum camber is varied in $[0.00, 0.06]$; both are independently varied in increments of 0.01. This creates a total of $9 \times 7 = 63$ airfoil shapes. Each airfoil is evaluated at three Reynolds numbers: 500×10^3, 2×10^6, and 8×10^6. This gives a total of 189 aerodynamic cases. In all cases, $\alpha = 5°$, $M_\infty = 0$, and $N_{\text{crit}} = 9$. This selection of aerodynamic cases is relatively "easy" and is deliberately chosen to give a runtime speed advantage to XFoil. This is because XFoil, as an iterative algorithm, tends to have an increasing computational cost as flow cases become more difficult. By contrast, NeuralFoil has a fixed computational cost per analysis. Overall, 95% of cases converged with XFoil, averaged over all discretization levels.

The "ground truth" reference solutions for this study are taken from a high-resolution XFoil simulation with 289 points, which is the highest resolution that XFoil 6.98 allows before it begins silently truncating wake points[15].

In Figure 7-11, the x-axis shows the accuracy of each model, measured as the mean relative error of the drag coefficient[16] across all 189 cases. The y-axis shows the runtime speed, which is the median time to run each of the 189 cases. By using the median runtime (as opposed to the mean), XFoil again gains a runtime speed advantage, as slow convergence creates a strong positive skew in the distribution of XFoil's runtimes.

As shown in Figure 7-11, all of NeuralFoil's model sizes give a speedup over XFoil, even when controlling for accuracy. With naïve looping over all 189 cases, NeuralFoil achieves the same accuracy level as XFoil at speeds more than 8x faster. However, NeuralFoil really shines when taking advantage of its vectorization and analyzing all 189 cases simultaneously. Here, speedups can be nearly 1,000x at the same accuracy level.

[15] This is due to a fixed-size array that XFoil allocates named `IDX`, which is truncated to avoid overflowing.
[16] Or, equivalently, the mean absolute error of $\ln(C_D)$

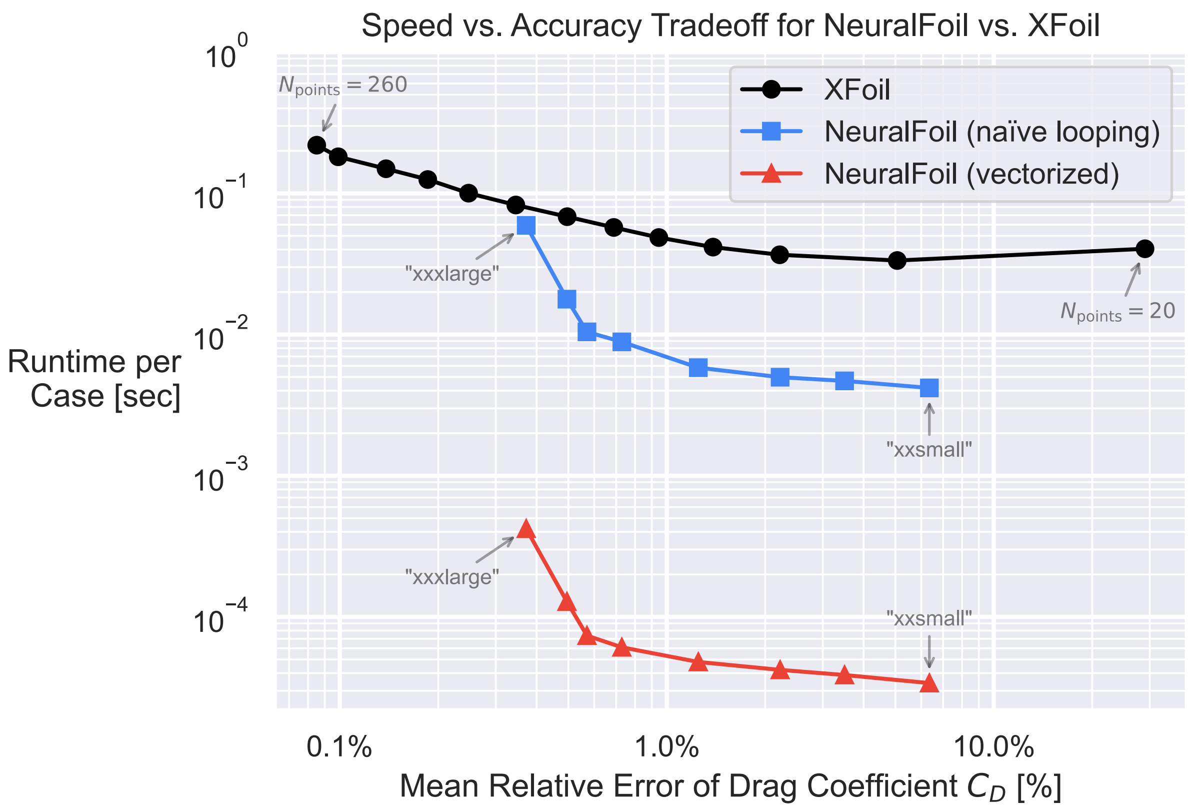


Figure 7-11: Comparison of runtime speed for NeuralFoil and XFoil, while controlling for accuracy. Evaluated on a varied database of NACA airfoils, with ground truth from XFoil at its highest allowable resolution (289 points). XFoil runs are varying resolution; NeuralFoil runs are varying model sizes. All speeds are measured on an AMD Ryzen 7 5800H laptop CPU.

Several other notable observations can be made from Figure 7-11. First, NeuralFoil's accuracy on "easy" cases appears far better than Table 7.3 suggests. This is because the test dataset in Table 7.3 includes many post-stall and transition-sensitive cases, as described in Section 7.2.5. For example, the mean relative error of drag for the "xxxlarge" model is 2.0% on the test dataset, while the same metric on this easier dataset is 0.37%. This reinforces that, when comparing accuracy across studies, the difficulty of the evaluation cases must be considered.

Secondly, NeuralFoil achieves a "knee" in the speed-accuracy tradeoff curve roughly near the "xlarge" model size. This is therefore taken as the default model size for NeuralFoil, though different use cases may encourage other choices.

Finally, the NeuralFoil models all achieve strong-Pareto-dominance over XFoil in this study, as measured by a speed-accuracy tradeoff. This is true even before factoring in NeuralFoil's ability to vectorize computation; when this is taken into account, speedups of nearly 1,000x over XFoil are possible without compromising accuracy.

7.3.5 Sample Validation on Transonic Case

To demonstrate the effect of compressibility on NeuralFoil's aerodynamics estimates, we can compare NeuralFoil to various other approaches in a transonic case study. This case study analyzes a RAE2822 supercritical airfoil at flow conditions of $\text{Re}_\text{c} = 6.5 \times 10^6$ and $\alpha = 1^\circ$, with natural transition and $N_\text{crit} = 9$. The freestream Mach number is varied from subsonic to transonic conditions. The results of this study are shown in Figure 7-12. Here, NeuralFoil's results are compared to those from four existing approaches:

- XFoil 6.98 [95]: This uses a boundary-element potential flow solver with a Karman-Tsien compressibility correction for the outer flow, and an integral boundary layer (IBL) model for the boundary layer. Transition is handled with an e^N method.

- XFoil 7.02: This uses a grid-based full-potential solver for the outer flow; compared to a linearized potential flow solver, this has stronger theoretical grounding for transonic analysis and the ability to directly capture shock waves. The boundary layer and transition models are the same as those in XFoil 6.98.

- MSES [228]: This is a hybrid Euler / Full-Potential solver that has yet-stronger theoretical grounding as shocks become stronger. The boundary layer and transition models are similar to those in XFoil 6.98, though viscous-inviscid coupling is achieved via a displacement-body approach rather than a wall-transpiration approach. This model is believed to be the most accurate of those listed here for transonic flows with weak shocks.

- SU2 [149]: This is a RANS solver with a Spalart-Allmaras turbulence model [229]. Transition is handled using a Langtry-Menter (LM) correlation-based model [230], using the Malan correlation [231]. Freestream turbulence intensity is set to 0.07%, which approximately corresponds to $N_{\text{crit}} = 9$ following equations from Drela [154]. However, this is not a direct equivalence, and the LM model predicts slightly earlier transition than e^N-based methods in this case, causing higher viscous drag.

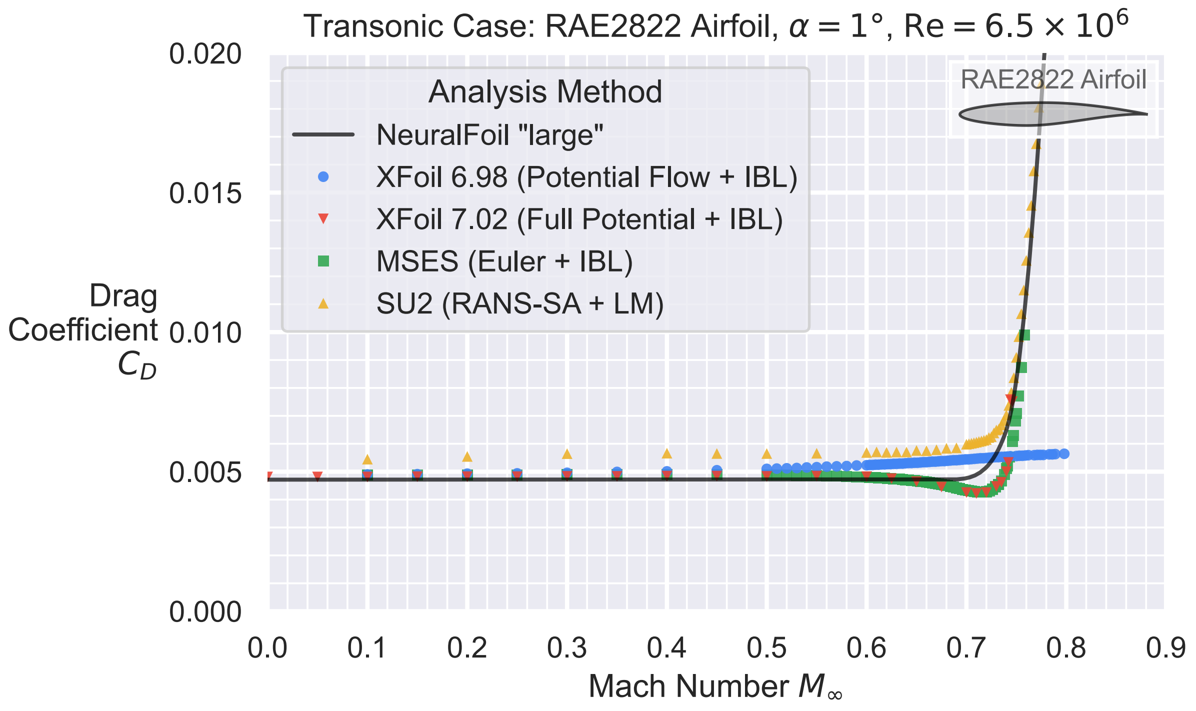


Figure 7-12: Validation of NeuralFoil at predicting the emergence of transonic flow, using the RAE2822 airfoil at $\text{Re}_c = 6.5 \times 10^6$ and $\alpha = 1°$. NeuralFoil's results are compared to those from XFoil 6.98, XFoil 7.02, MSES, and SU2. NeuralFoil's results are shown for the "large" model described in Table 7.1.

NeuralFoil's results in Figure 7-12 are qualitatively similar to those obtained with other methods, particularly with respect to the location and steepness of the transonic drag rise. Some differences remain, however. Notably, models that use both an integral-boundary-layer approach and a higher-fidelity transonic treatment (i.e., XFoil 7.02, MSES) capture a slight decrease in drag in a window between the critical and drag-divergent Mach numbers. This effect is believed to be real, and it occurs because the locally-supersonic flow in this regime causes a favorable pressure gradient that delays the top-surface boundary layer transition. Such an effect is not captured by NeuralFoil, SU2, or XFoil 6.98, as capturing this effect requires the combination of both shock-resolution and advanced transition modeling.

The results of these analyses can be quantitatively compared in Table 7.4, which shows the critical and drag-divergent Mach numbers predicted by each method. Corroborating the values in this table, Drela cites the critical Mach number for this case as $M_{\text{crit}} = 0.71$ [154]. NeuralFoil appears to underestimate the critical Mach number slightly in this case, although the drag-divergent Mach number agrees very closely with other methods.

Table 7.4: Predictions of critical and drag-divergent Mach numbers for the RAE2822 airfoil at $\mathrm{Re}_c = 6.5 \times 10^6$ and $\alpha = 1^\circ$, using various methods.

	Critical Mach Number M_{crit}	Drag-Divergent Mach Number M_{dd} †
NeuralFoil "large"	0.671	0.739
NeuralFoil "xxxlarge"	0.673	0.741
XFoil 6.98 (Potential Flow + IBL)	0.688	N/A
XFoil 7.02 (Full Potential + IBL)	0.705	0.739
MSES (Euler + IBL)	0.714	0.738
SU2 (RANS-SA + LM)	-	0.738

† Defined as $\partial(C_D)/\partial M_\infty \geq 0.1$, following Mason [224].

7.3.6 Comparison to other Airfoil Machine Learning Models

Here, we compare and contrast NeuralFoil with several prior attempts to apply machine learning techniques to airfoil aerodynamics analysis. Peng et al. [24] apply a convolutional neural network to this task, where the input is an array of airfoil coordinates and the output is a scalar lift coefficient. Bouhlel et al. [23] use feedforward neural networks to predict airfoil lift and drag coefficients; this is augmented by Sobolev regularization that adds gradient information into the loss function. Du et al. [22] use multilayer perceptrons, recurrent neural networks, and mixture-of-experts approaches to predict both scalar (lift and drag) and vector (pressure distribution) quantities.

Compared to prior work, NeuralFoil is trained on a significantly broader input space, particularly with respect to angle of attack and transition assumptions. This comparison is illustrated in Table 7.5. The accuracy values listed in Table 7.5 are not directly comparable, as they are computed on different datasets and with different metrics. Nevertheless, NeuralFoil achieves similar accuracy to prior work, despite including many extreme cases in its evaluation sample (e.g., post-stall flows, transitional Reynolds numbers, and high-curvature airfoils associated with 18 shape variables). This is possible partially because of the much larger volume of training data, but also due to the structural embedding of physics knowledge into the model described throughout Section 7.2.

Accurate and rapid aerodynamics analysis across this broader input space has the potential to enable airfoil design optimization without overly restricting the design space. Furthermore, certain types of robust optimization become possible. For example, performing a multipoint optimization study with respect to N_{crit} becomes possible with NeuralFoil, allowing the designer to mitigate the tendency of optimizers to "over-optimize" to one specific transition location[17].

[17]For more details on this, see Drela [33]. In general, airfoil shape optimization algorithms tend to geometrically "fill in" locations where laminar separation bubbles would otherwise go, which dramatically worsens off-design performance.

Table 7.5: Comparison of NeuralFoil to prior literature in airfoil aerodynamics machine learning. NeuralFoil achieves similar accuracy, despite including a much broader range of flow conditions in its training and evaluation distributions. Models from Du et al. [22], Bouhlel et al. [23], and Peng et al. [24] use separate subsonic and transonic models.

		NeuralFoil	**Du et al. [22]**	**Bouhlel et al. [23]**	**Peng et al. [24]**
Diversity of "airfoil design space" used for both training and evaluation	**Airfoil shape diversity**	18 shape variables	16 movable control points	14 shape vars. (subsonic) 8 shape vars. (transonic)	UIUC database ($\sim$1500 fixed shapes)
	Angle of attack range	Uniform + normal distrib.; central 95% in range $[-17^\circ, +18^\circ]$	0° to 3°	-0.5° to 6° (subsonic) -1.5° to 4.5° (transonic)	-2° to 10°
	Reynolds number range	Log-normal distrib.; central 95% in range $[1.9 \times 10^3, 262 \times 10^6]$	10^4 to 10^{10} (training)	None specified	13×10^6 (fixed)
	Mach number range	0 for learned core, although compressibility correction allows prediction up to M_{crit}	0.3 to 0.6 (subsonic) 0.6 to 0.7 (transonic)	0.3 to 0.6 (subsonic) 0.7 to 0.75 (transonic)	0.3 (fixed)
	Transition, turbulence, and forced trips	N_{crit} varied from 0 to 18; Trips varied from LE to TE	None specified ($\tilde{\nu}_\infty = 0$ assumed)	None specified ($\tilde{\nu}_\infty = 0$ assumed)	None specified ($N_{\text{crit}} = 9$ assumed)
	Training data	XFoil (7.5M samples)	RANS-SA via ADflow (86k samples)	RANS-SA via ADflow (42k samples)	XFoil (16k samples)
Accuracy, as **evaluated on the corresponding distribution above**	**C_L mean absolute error**[†]	0.012	0.010	Unspecified	1.0%
	C_D mean relative error[†]	2.0%	1.7%	0.3%	N/A (not attempted)

[†] As further discussed in Section 7.3.6, accuracy results should not be directly compared across models, as they are evaluated on different distributions. In particular, NeuralFoil's evaluation set spans a far larger analysis space, including post-stall flows and transitional Reynolds numbers.Are

7.3.7 Definitional Notes on Physics-Informed Machine Learning

In this work, we have described NeuralFoil as an instance of *physics-informed machine learning* – this is a notoriously overloaded term that academics will vociferously debate the definition of, so it becomes necessary to clarify the definition used in this work[18]. Here, we follow the definition by Brunton [217, 232], where physics-informed machine learning refers to any machine learning model that leverages some form of knowledge about the underlying problem physics to improve its performance (as measured by accuracy, generalization, computational efficiency, data efficiency, etc.)[19]. This knowledge can be embedded at *any* of five stages, reproduced from Brunton [232]:

1. Formulating a problem to model
2. Collecting and curating training data to inform the model
3. Choosing an architecture with which to represent the model
4. Designing a loss function to assess the performance of the model
5. Selecting and implementing an optimization algorithm to train the model

In the case of NeuralFoil, physical knowledge is embedded at the first four of these steps, by a) carefully parameterizing input and output latent spaces to have close-to-affine mappings, b) curating training data that is deliberately diverse in ways likely to excite various physical phenomena, c) embedding in physics symmetries into the model architecture, and d) constructing the loss function based on knowledge about the relative importance of different aerodynamic quantities. Arguably, NeuralFoil's self-estimation of analysis confidence also constitutes a physics-informed feature, as the model asymptotes to empirical

[18]Ultimately, the classification of whether a model is *physics-informed* or not is a marketing distraction. The disproportionate scrutiny on nomenclature would be better-placed on assessing de facto practicality - whether a model is accurate, generalizable, fast, and trustworthy as an engineering tool.

[19]An alternative definition is that physics-informed machine learning encompasses any method that does something *other* than blindly learn a functional mapping from one semantic-free latent space to another.

physics models when it discerns that the learned core can no longer be trusted. These considerations demonstrably lead to a more generalizable model than prior efforts, as shown in Sections 7.3.1 and 7.3.6.

7.4 Airfoil Design Optimization with NeuralFoil

Although the accuracy of NeuralFoil with respect to XFoil has been demonstrated in Section 7.3, this alone is not sufficient to demonstrate that NeuralFoil is useful for design optimization. A major reason for this is that engineering design optimization is an adversarial process, where the optimizer aims to exploit not only the underlying physics, but also any errors in the model. Exploitation of the physics is desirable and ultimately the goal of engineering design optimization, but exploitation of model errors leads to designs that are ostensibly promising but lose their luster when analyzed with other computational tools or actually built and flown.

To assess whether a model such as NeuralFoil is prone to model error exploitation, we set up an aerodynamic shape optimization problem with a known answer from the literature, and evaluate how close the optimized result matches this. In aerodynamic shape optimization (and aircraft design more broadly), there are relatively few problems that are rigorously specified, and even fewer where the correct optimized result is provided in exacting detail. However, one such case study that provides a useful design problem is Drela's 1998 *Pros & Cons of Airfoil Optimization* [33], which discusses airfoil design for the *MIT Daedalus* human-powered aircraft [160, 161, 233]. Basic figures for this aircraft, along with its centerline airfoil, the DAE-11, are shown in Figure 7-13.

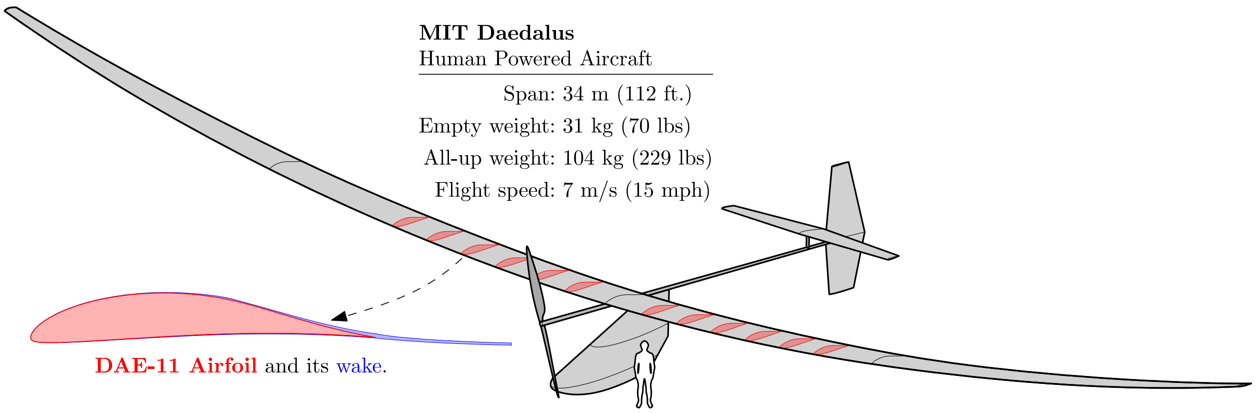

MIT Daedalus
Human Powered Aircraft
Span: 34 m (112 ft.)
Empty weight: 31 kg (70 lbs)
All-up weight: 104 kg (229 lbs)
Flight speed: 7 m/s (15 mph)
DAE-11 Airfoil and its wake.

Figure 7-13: Drawing of the *MIT Daedalus* human-powered aircraft, along with its centerline airfoil, the DAE-11. This airfoil is the subject of the airfoil design optimization problem posed in this section.

The design of the DAE-11 airfoil is effectively defined by the following airfoil design optimization problem; this formulation is taken directly from Drela [33] with minor modifications[20]:

- **Objective**: Minimize $C_{D,\text{mean}}$ at $\text{Re}_c = 500\text{k} \cdot \left(\frac{C_L}{1.25}\right)^{-0.5}$ and $\text{M}_\infty = 0.03$
  - Where $C_{D,\text{mean}}$ is the weighted average of C_D values at $C_L = [0.8, 1.0, 1.2, 1.4, 1.5, 1.6]$, with relative weights of $[5, 6, 7, 8, 9, 10]$ at each respective C_L
- **Variables**: airfoil shape (18 variables; described in Section 7.2.2) and angle of attack α
- **Constraints**:
  - $C_M \geq -0.133$ (at all C_L operating points)
  - Trailing-edge angle $\theta_{\text{TE}} \geq 6.03°$ (allows manufacturability)
  - Leading-edge angle $\theta_{\text{LE}} = 180°$ (prevents the formation of a sharp leading edge)
  - At $x/c = 0.33$, $t/c \geq 0.128$ (to accommodate the main spar)
  - At $x/c = 0.90$, $t/c \geq 0.014$ (to accommodate the rear spar)

[20]Specifically, a constant-$\text{Re}_c\sqrt{C_L}$ (fixed-lift) polar is used instead of a constant-Re_c polar; the C_M constraint is clarified to apply to all C_L operating points; and the θ_{TE} constraint is matched to the DAE-11 airfoil

When posing airfoil design optimization problems (using NeuralFoil or with any other tool), it is often helpful to add basic regularization constraints; these tend to make airfoil optimization better-behaved by ruling out non-physical and unrealistic airfoils. For the simple airfoil design problem described above, these regularization strategies are not necessary for convergence. However, they are discussed here as “best practices” for airfoil optimization on more complicated problems:

The first such constraint is that the airfoil thickness must be positive everywhere. This prevents the creation of self-intersecting airfoils, which can be analyzed in both XFoil and NeuralFoil but are clearly not physically-valid shapes.

The second such constraint aims to guide the optimizer away from airfoils that have excessively high local surface curvature, as these have boundary layer behavior that is inherently difficult to characterize. A useful global metric for this curvature, which we call the wiggliness w, is mathematically similar to a metric by Wahba [234] that has long proven successful in univariate spline regularization:

$$w(\text{airfoil}) = \int_0^1 \left(\frac{d^2}{dx^2} y_{\text{lower}}(x) \right)^2 + \left(\frac{d^2}{dx^2} y_{\text{upper}}(x) \right)^2 dx$$

We can loosely approximate the spirit of this wiggliness measure from the discrete Kulfan parameterization as follows:

$$w(\text{airfoil}) \approx \sum_{i=2}^{N-1} \left(c_{\text{lower},i+1} - 2 \cdot c_{\text{lower},i} + c_{\text{lower},i-1} \right)^2 + \left(c_{\text{upper},i+1} - 2 \cdot c_{\text{upper},i} + c_{\text{upper},i-1} \right)^2$$

where $c_{\text{lower},i}$ and $c_{\text{upper},i}$ are the ith Kulfan (CST) coefficients of the lower and upper surfaces, respectively. A reasonable heuristic for this regularization constraint is to restrict this metric to no more than four times that of the original initial guess airfoil (which is typically some generic airfoil, such as a NACA0012).

Using the problem formulation described above, we can solve this airfoil design optimization problem by coupling NeuralFoil with the AeroSandbox aircraft design optimization framework [1]. This adds automatic differentiation capabilities to NeuralFoil, allowing efficient optimization using gradient-based methods. This optimization problem is solved in approximately 7 seconds on a standard laptop; the speed of this solution provides rapid feedback to the designer on how to improve the optimization problem formulation to capture design intent.

Figure 7-14 compares three airfoil designs produced with different methodologies, each aimed at solving the *MIT Daedalus* airfoil design problem described previously:

- **NeuralFoil-optimized**, which is the result of optimizing while using NeuralFoil as the aerodynamics analysis tool. Notably, this uses NeuralFoil's 18-parameter CST shape parameterization.

- **XFoil-optimized**, which is the result of optimizing while using XFoil as the aerodynamics analysis tool. Here, we directly wrap XFoil calls in a gradient-based optimizer, with gradients computed by finite-differencing. Notably, here the airfoil shape is parameterized using 60 geometric degrees of freedom, to reproduce results from a study by Drela [33].

- **Expert-designed**, which is the original DAE-11 airfoil designed by Drela [207] and used on the as-flown *MIT Daedalus* aircraft.

For comparison, Figure 7-14 also includes the initial guess airfoil that was provided to the optimization algorithms (a simple NACA0012). This showcases that the optimization process is robust to initial guesses that are relatively far from the optimal solution.

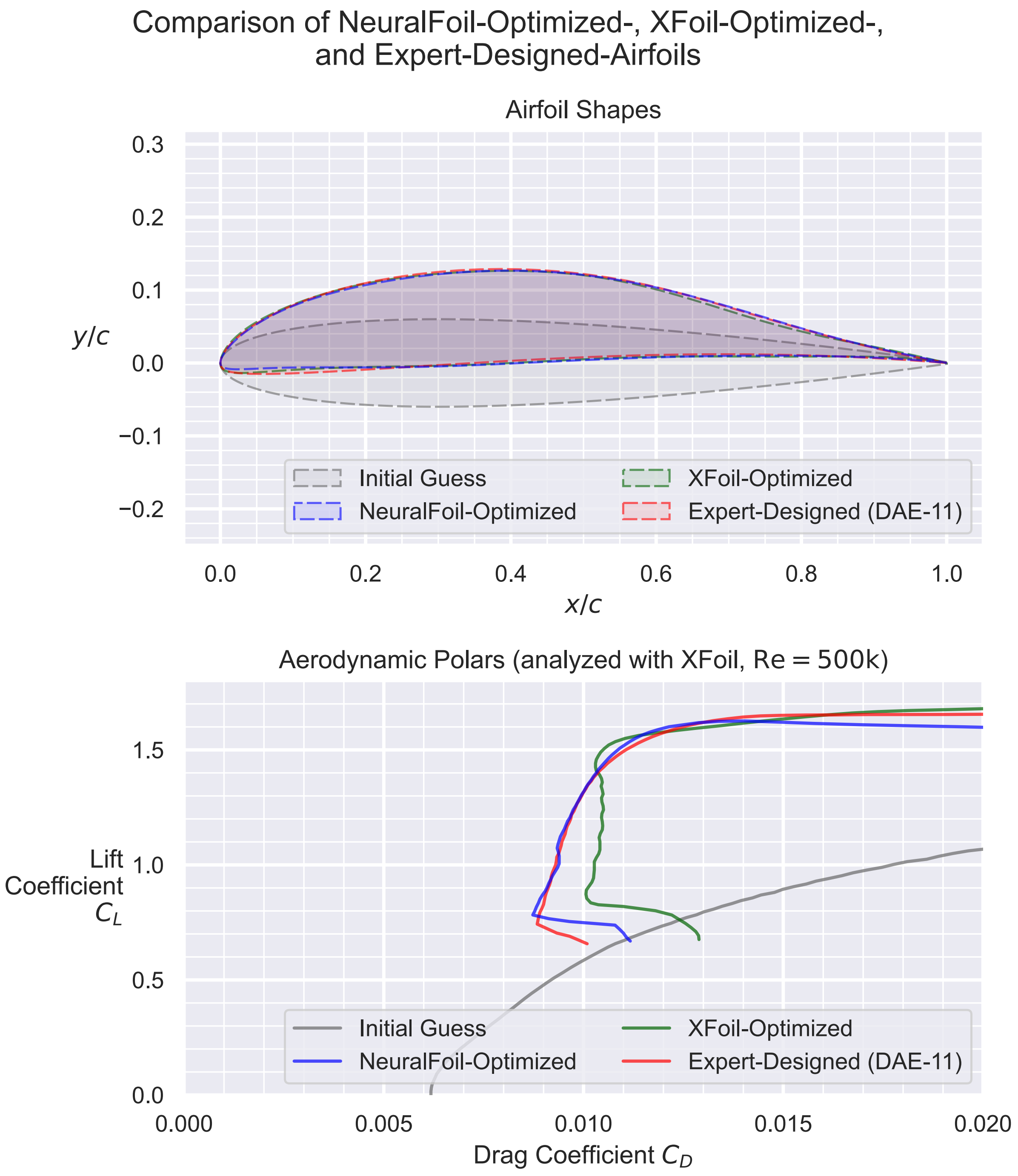


Figure 7-14: NeuralFoil-optimized airfoils have similar performance and shape to expert-designed airfoils. Adversarial optimization is not observed, as the NeuralFoil-optimized and XFoil-optimized airfoils yield similar performance when analyzed using XFoil. The gray "Initial Guess" airfoil is used to initialize NeuralFoil optimization.

The airfoil produced using NeuralFoil optimization is quite similar in shape to the those produced using XFoil optimization and expert-designed airfoils, when given the same design objectives and constraints. Likewise, aerodynamic performance is quite similar across all three airfoils. Notably, the aerodynamic polars depicted in Figure 7-14 were produced by post-optimality analysis using XFoil. Given the minimal discrepancy between the NeuralFoil-optimized and XFoil-optimized airfoils, this suggests that NeuralFoil is reasonably resistant to model error exploitation on problems of practical interest.

Notably, the "XFoil-optimized" airfoil (shown in green in Figure 7-14) actually performs worse than the NeuralFoil-optimized airfoil. This may initially appear surprising, as NeuralFoil is a surrogate model for XFoil – how can optimizing through a surrogate model yield better performance than optimizing through the original model? The answer lies in the fact that NeuralFoil yields a C^∞-continuous solution, and hence a gradient-based optimizer using NeuralFoil is much less likely to get stuck in a local minimum than an optimizer using XFoil. On top of this, NeuralFoil yields exact gradients through automatic differentiation, while XFoil's gradients are computed via finite differencing and hence less accurate.

Furthermore, NeuralFoil does not exhibit the point-optimization problem seen in a prior study by Drela [33], where the optimizer would tend to create small separation-bubble-sized bumps. This is an inherent feature of the lower-dimensional shape parameterization used by NeuralFoil, and is in fact a nice feature rather than a bug – practical airfoils should not have these bumps, as this is an artifact of the discrete optimization problem formulation.

Interestingly, the geometric differences between the expert-designed and optimized airfoils (both from XFoil and NeuralFoil) in Figure 7-14 are attributable to design goals that were not factored into the quantitative problem formulation. For example, the optimized airfoils both exhibit a small amount of lower-surface concavity in the vicinity of $x/c \approx 0.15$, while the expert-designed DAE-11 has a flatter lower surface. Drela discusses reasons for this discrepancy in [33], noting that the concavity that the optimizer prefers would cause the wing covering (in the case of *MIT Daedalus*, a thin shrunk covering of Mylar) to lift off of the surface of the rib, creating a bubble. This undesirable effect is not captured in the quantitative problem formulation, providing a cautionary tale that an optimized airfoil is only as good as the problem formulation that led to it.

Nevertheless, the strength of NeuralFoil-powered optimization is that it allows one to generate optimized airfoils that are remarkably similar to expert-designed airfoils in mere seconds. This optimization capability is more-than-adequate for conceptual aircraft design, and it provides an excellent starting point for expert-guided airfoil design refinement.

7.5 Computational Reproducibility

NeuralFoil is implemented as an open-source Python package with minimal dependencies (only NumPy [125] for the actual aerodynamic modeling), allowing easy installation across a variety of platforms.

All source code used to generate results in this chapter is publicly available at `https://github.com/peterdsharpe/neuralfoil`. Model architecture, weights, training scripts, and usage examples are included in this repository. For end-users, NeuralFoil is best accessed through the open-source AeroSandbox aircraft design optimization framework [1]; this provides some of the advanced features (e.g., 360° angle of attack, compressibility corrections, and control surface deflections) described in this chapter that are not yet available in the standalone NeuralFoil package.

7.6 Summary and Future Work

This chapter has introduced NeuralFoil, a physics-informed machine learning tool for airfoil aerodynamics analysis. Relative to existing popular tools such as XFoil, NeuralFoil offers improved computational efficiency, even after controlling for equivalent accuracy. Its ability to handle a wide range of practical airfoil shapes and flow conditions makes it a versatile tool in aerodynamic analysis.

The accuracy of NeuralFoil with respect to XFoil is demonstrated across several test cases, with a particular focus on airfoils that NeuralFoil was not trained on (i.e., out-of-distribution). Much of the accuracy achieved here is due to the embedding of domain-specific physics knowledge into the model architecture, which increases the parameter-efficiency and generalizability of the resulting model.

Also presented in this chapter is an application of NeuralFoil to airfoil design optimization for the *MIT Daedalus* human-powered aircraft, where it quickly produces designs close in performance and shape to expert-crafted airfoils. Incorporation of geometric constraints into this study shows that NeuralFoil-based optimization is able to capture the non-aerodynamic considerations that are typical of practical design problems. This case study also demonstrates the real-world value of many optimization-friendly features that are embedded into the model architecture (e.g., continuity, differentiability, and static code execution paths). Finally, these results also suggest that NeuralFoil is not prone to adversarial exploitation of model errors.

Another contribution of this chapter is a new technique that enables machine learning surrogate models to estimate their own trustworthiness, a critical capability for robust engineering design. This "analysis confidence" metric can be used directly to guide human-in-the-loop analysis, or it can be directly constrained during a formal optimization process to improve the robustness of the resulting designs.

Future work could include extending NeuralFoil to handle multi-element airfoils, which would require revisiting the geometry parameterization among other considerations. Another useful research direction would be to include compressibility corrections directly within NeuralFoil's learned model, as opposed to using an analytical correction to incompressible results. Currently, the outer flow receives a compressibility correction, but the effects of this modified pressure distribution are not then fed back into the boundary layer model. Therefore, some transonic boundary layer physics (such as an extended laminar run within a supersonic zone that causes a favorable pressure gradient) are fundamentally lost. This does, however, increase the dimensionality of the input space, and thus may affect training data requirements. Related to that, another possible next study could investigate how model accuracy changes with the amount of training data used. NeuralFoil was trained with roughly two orders of magnitude more aerodynamic cases than similar studies, although it is not yet known whether this is strictly necessary to achieve the model's performance.

As a Python-based, open-source package, NeuralFoil offers a practical and versatile solution that is accessible to both academia and industry. Its accurate and rapid performance offer value throughout the aircraft development process, though this capability is especially interesting in the early stages of aircraft development where quick feedback on design changes is crucial.

Chapter 8

Aircraft System Identification from Minimal Sensor Data

This chapter includes content from the author's prior publication [235].

This chapter presents a system-identification-like process for estimating an aircraft's aerodynamic and propulsive performance characteristics from minimal flight data. The method is based on a physics-informed regression approach, which uses an unsteady physics model of the aircraft's dynamics to constrain the regression problem. The method is demonstrated on a flight test dataset collected from a small electric aircraft. The results show that the method is able to estimate the aircraft's power curve, aerodynamic polar, and propulsive efficiency curve with surprisingly high accuracy given limited data. In addition, we show that bootstrapped uncertainty estimates can be obtained for these performance curves using this method.

A related additional contribution is a new method for estimating the statistical properties of the noise component of a noisy signal, only using the noisy observations themselves. This relies on certain common assumptions about the noise-generating process, such as spectral separation between the true signal and the noise. Noise is isolated by repeated numerical differentiation, which is shown to be stable in numerical experiment and on application to actual flight data. An example implementation of this noise variance estimator is given and shown to be convergent to the true noise variance in numerical experiment. Higher-order implementations of this estimator are also presented and shown to isolate the noise accurately even when the underlying true signal is quite close to the Nyquist frequency.

8.1 Introduction

A primary goal of a new aircraft's initial flight test campaign is to experimentally determine the aircraft's aerodynamic and propulsive performance characteristics. The desired outputs of this process typically include:

1. The aircraft's power curve, which gives the required power to maintain level flight as a function of airspeed.

2. The aircraft's C_L-C_D aerodynamic polar. This gives the relationship between the aircraft's lift

and drag coefficients as angle of attack is varied, representing the locus of possible aerodynamic operating points.

3. The aircraft's propulsive efficiency curve, which yields the overall propulsive efficiency, typically as a function of throttle setting and/or propeller advance ratio (if relevant).

All three of these results represent sweeps through the aircraft's flight envelope, varying the aircraft's airspeed, pitch trim setting, and throttle setting, and measuring the resulting lift, drag, and power required.

8.1.1 Traditional Flight Measurement Methods and Limitations

Typically, these performance outputs are obtained by performing an extensive campaign of careful, controlled flight experiments at quasi-steady flight conditions.

For example, the aerodynamic polar is commonly measured by performing a series of long, steady power-off glides at different airspeeds, thus isolating the effect of propulsive power. (If the aircraft is propeller-driven, propeller windmilling drag can also be estimated and calibrated out for improved accuracy, although this introduces significant additional uncertainty.) An implicit assumption here is that pitch trim is adjusted to maintain these airspeeds. The glide ratio (or equivalently, if local airmass motion and airspeed changes are calibrated out, the L/D) is then measured at a given airspeed by using a simple two-point finite-difference between the beginning and end of the glide:

$$(L/D) = \frac{h(t_1) - h(t_2)}{U \cdot (t_2 - t_1)}$$

where $h(t)$ represents the altitude at time t, U represents the airspeed, and t_1 and t_2 are the beginning- and end-times of the glide. The drag is then computed as $D = W/(L/D)$, and C_L and C_D can be nondimensionalized from here.

For aerodynamically-efficient aircraft with high L/D ratios (such as sailplanes), using this strategy without calibration leads to substantial error due to airmass motion (e.g., thermals, ridge lift) and airspeed changes. Instead, another common strategy for estimating aerodynamic polars during flight testing is to fly these power-off glides along with a wingman aircraft. This wingman aircraft flies the same power-off glide as the test aircraft, at the same airspeed, with the same starting altitude, and in the same vicinity. The wingman aircraft is assumed to have a known aerodynamic polar, and hence the difference in altitude between the two aircraft at the end of a glide segment can be used to estimate the test aircraft's aerodynamic polar. This approach calibrates out the effect of local airmass motion on the measured glide ratio.

More extreme strategies have also been implemented to measure the aerodynamic performance curve. For example, the glide ratio of the P-51 fighter aircraft was allegedly measured by removing the propeller, towing it aloft, and gliding it down. This removes the error associated with estimating propeller windmilling drag, but this is a more involved and cumbersome procedure to include in a flight test campaign.

Full-scale wind tunnel testing is another possible method for estimating the aerodynamic polar of the as-built aircraft, although this is similarly expensive and time-consuming, rendering it infeasible for most aircraft development programs. Scale models in wind tunnels as well as CFD simulation can also provide useful aerodynamic estimates, but these results can differ significantly from the aerodynamics of real as-built aircraft. For example, Hoerner [236] gives a pathological example of the Messerschmitt Bf 109 fighter aircraft, where roughly *half of the drag of the as-built aircraft* is attributable to components that are typically de-featured in modern computational analysis[1]. This Bf 109 case likely has an unusually-high number of such protuberances, but even for modern aircraft, the drag of the as-built aircraft can differ significantly from computational estimates or small-scale wind tunnel testing. This motivates flight-test-based aerodynamic polar characterization.

Like the aerodynamic polar, the power curve is also often measured by flying in steady flight segments at various airspeeds. For each airspeed, level flight is held with power and pitch trim adjusted as needed for stabilized cruise. The required input power to the propulsion system P_{in} is then measured. In a conventionally-fueled airplane, this can be computed using the fuel flow rate at a given throttle setting (read off the panel via a fuel flow meter) and the specific energy of the fuel, or indirectly from the engine RPM depending on the powerplant data available. For an electric airplane, it can be computed as the product of battery current and voltage, which are readily available. This procedure is then repeated for a variety of airspeeds, and the resulting power curve is plotted.

The propulsive efficiency can be roughly estimated based on these experiments as well. One possible procedure is as follows:

1. The input power to the propulsion system P_{in} is measured, as above.

2. We observe the steady-state climb/sink rate of the aircraft at this throttle setting and we compare this to the power-off sink rate. The difference between these two sink rates represents the thrust power done by the propulsion system. The thrust power (or "air power") can be computed as:

$$P_{\text{thrust}} = T \cdot U = mg \left(\left.\frac{dh}{dt}\right|_{\text{power on}} - \left.\frac{dh}{dt}\right|_{\text{power off}} \right)$$

[1]e.g., rivets, sheet metal gaps, control surface gaps & hinges, engine and radiator thermal effects, inlet flow distortion, antennae, sensors, lights, guns, holes for ventilation and cooling, etc.

3. Then, we can compute the propulsive efficiency as the ratio of the input power to the air power:

$$\eta_{\text{propulsive}} = \frac{P_{\text{thrust}}}{P_{\text{in}}}$$

These traditional methods for measuring aerodynamic and propulsive performance have several limitations. The biggest limitation by far is that they require vast amounts of flight time to obtain a sufficient number of data points to characterize the aircraft's performance.

Furthermore, the aircraft must be flown at precisely-controlled conditions. Traditional methods are not well-suited to estimating the performance of an aircraft that is not in steady flight – in the methods described here, only steady data (e.g., a stabilized glide or cruise) can be used. For example, data recorded during an aircraft's climb or descent to flight test altitude is discarded, which is wasteful; ideally, every single second of data should contribute to refining our estimate of the aircraft's performance. This requirement to maintain steady conditions can be frustrating and tedious for the pilot to maintain, and it can also be dangerous if the aircraft is flown for an extended duration at conditions that are close to the edge of the flight envelope (e.g., behind the power curve on a not-yet-characterized experimental airplane).

Traditional methods also make no direct estimate of sensor noise (and hence, uncertainty) – data is collected and averaged until the experimenter is satisfied that the data is "good enough". This is a subjective process that leaves it difficult to quantify the uncertainty in the resulting performance estimates.

Nevertheless, the methodology described here provides a useful existence proof that airplane performance curves are fundamentally observable (and hence, able to be recovered) given airspeed, altitude, and power data of sufficiently varied flight. A primary goal of this chapter is to develop a methodology that can estimate these performance curves from a much smaller amount of flight data, which would decrease the cost and intentionality required to obtain these performance estimates.

8.1.2 Inference-Based Flight Data Reconstruction Methods

We propose a method of estimating these performance characteristics that views the aircraft flight test as a data-generating process, from which we can infer performance relationships. Figure 8-1 broadly illustrates the toolchain for performance reconstruction that represents this chapter's central contribution. This approach contrasts with traditional methods, which view flight testing as a process of pure measurement, based only on analyzing the data itself. This distinction manifests in a few ways, each of which recognizes that we have process- or domain-specific information that collapses the possible solution space of possible performance models:

Firstly, with certain reasonable assumptions about the sensor noise characteristics, the data itself can reveal a surprisingly large amount of information about the sensor noise. Essentially, this involves using the data to estimate the "trustworthiness" *of the data itself*, a process described more fully in Section 8.3.1. One contribution of this chapter is a new cross-validation-like approach for estimating the sensor noise using only the properties of the data itself. This allows cleaner raw data inputs to the subsequent performance analysis toolchain that we develop, ultimately resulting in more accurate performance estimates and aiding uncertainty quantification. In Figure 8-1, this is represented by the separation of sensor data into its signal and noise components.

We also know certain characteristics of the aircraft's performance curve based on first-principles physics, which can further increase the effective information density and our resulting performance estimates. Another way to think about this is that the function space of performance curves is restricted to functions that generally follow first-principles physics, allowing us to reduce uncertainty on these resulting inferred relations. For example, we know that the true state of the aircraft must follow invariants from Newtonian physics (e.g., conservation of energy). Likewise, propulsive efficiency is a bounded function that must not exceed unity to be physically plausible. Embedding these constraints and invariants from physics reduces the number of unknowns, enabling more robust correction for unsteady effects. This is described in Section 8.4.

Finally, there are characteristics about performance relations that we can be embedded from domain-specific expertise. Even for models that do not have simple, closed-form invariants, we almost always have some priors (i.e., informed guesses) about how performance parameters should be related - model structure. For example, we know that the aircraft's drag coefficient will primarily be a function of lift coefficient, and the general shape of this function can be assumed. Similarly, we know that the propulsive efficiency will largely be a function of advance ratio, and the advance ratio can be inferred from the airspeed, input power, and first-order physics. Incorporation of these physics-based performance models embeds our physical understanding of the aircraft system into the estimation process, improving accuracy. For this reason, we term the performance reconstruction methodology described in this work a "physics-informed" approach.

The example given in the subsequent sections is for a small electric-driven propeller aircraft, although the methodology is applicable to any aircraft type. Some considerations for applying this methodology to other aircraft types are discussed in Section 8.6.

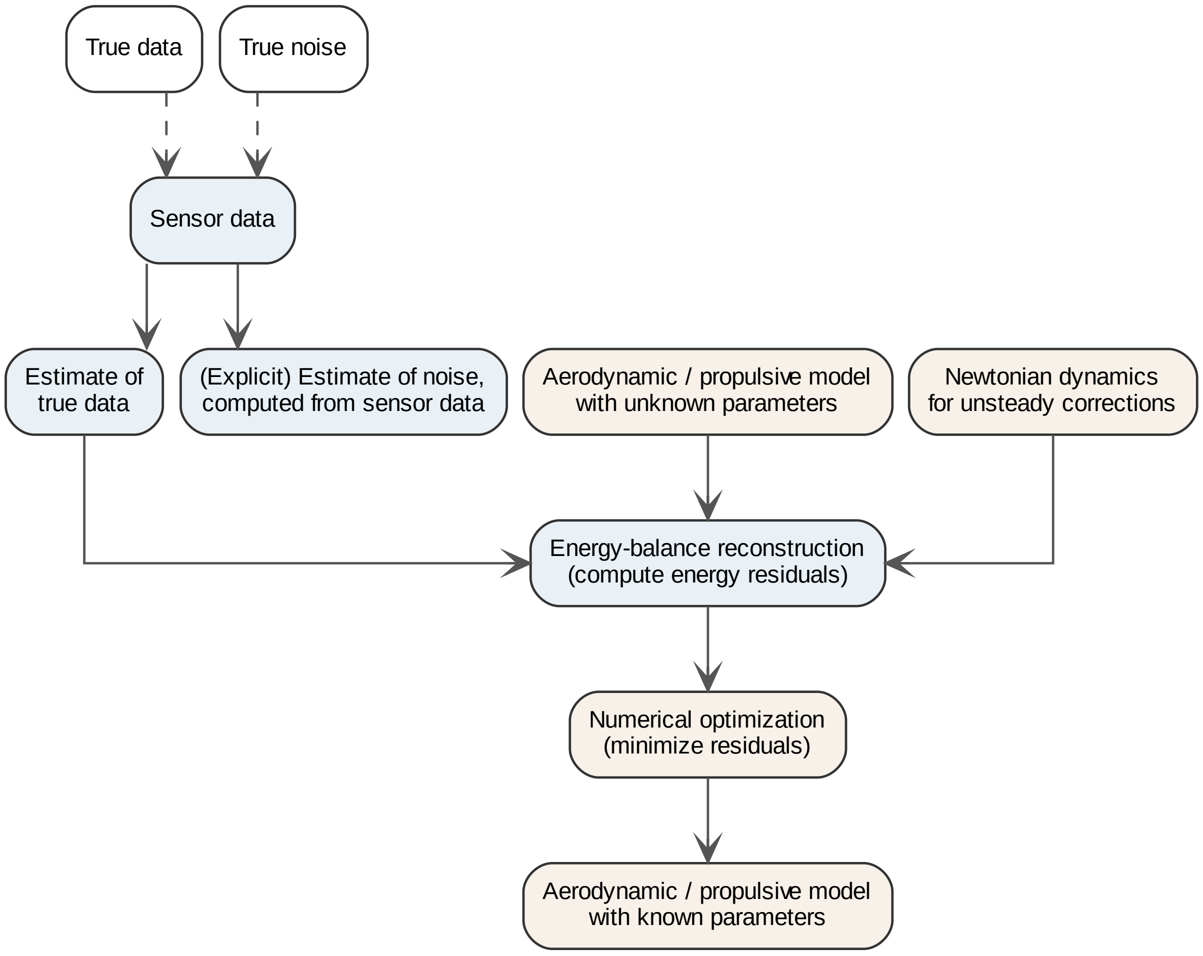

True data
True noise
Sensor data
Estimate of true data
(Explicit) Estimate of noise, computed from sensor data
Aerodynamic / propulsive model with unknown parameters
Newtonian dynamics for unsteady corrections
Energy-balance reconstruction (compute energy residuals)
Numerical optimization (minimize residuals)
Aerodynamic / propulsive model with known parameters

Figure 8-1: Process for inference-based performance reconstruction from flight data.

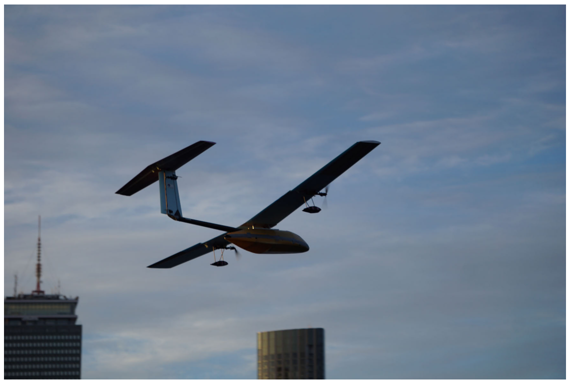
8.2 A Minimal Flight Test Dataset

To illustrate our proposed flight test data analysis procedure, this manuscript will use example data from a flight test conducted during MIT 16.821: Flight Vehicle Development, a senior-level aircraft design/build/fly capstone course at MIT AeroAstro. The flown aircraft, named *Solar Surfer*, is a remote-controlled solar-electric seaplane design with a 14-foot wingspan. The as-flown all-up mass is 9.4 kg.

General specifications of the aircraft can be found in the early design drawing that is reproduced in Section 4.5.1. While this drawing does not reflect as-built weights or various planform adjustments that were made during detailed design and construction, the overall configuration and performance numbers are roughly representative of the aircraft that was flown.

8.2.1 Flight Test Campaign

The aircraft was flown in the vicinity of the Charles River basin near Cambridge, MA. Five tests were conducted on the morning of May 3, 2023, of which two were airborne flight tests. The final airborne flight test lasted approximately 260 seconds (4.3 minutes), beginning and ending with a successful water takeoff and landing. Figure 8-2a depicts a still image of the aircraft in flight. A simple racetrack-like pattern was flown, as shown in Figure 8-2b. Winds were calm at roughly 1.5 m/s from the south.


(a) *Solar Surfer* in flight.

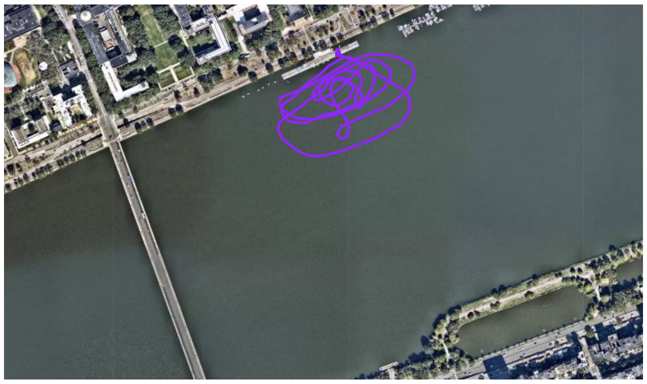

(b) Recorded GPS ground track during test flight.

Figure 8-2: Media from the *Solar Surfer* aircraft test flight.

Figure 8-3 illustrates the subset of the raw sensor data during flight that will be used for performance reconstruction. Overall, the data is reasonable, and we can clearly identify takeoff and landing by the strong increase in barometric altimeter noise near $t = 0$ sec and $t = 263$ sec. We can also identify several distinct flight phases, such as a high-speed pass near $t = 150$ sec and gliding periods near $t = 40$ sec, $t = 210$ sec, and $t = 250$ sec.

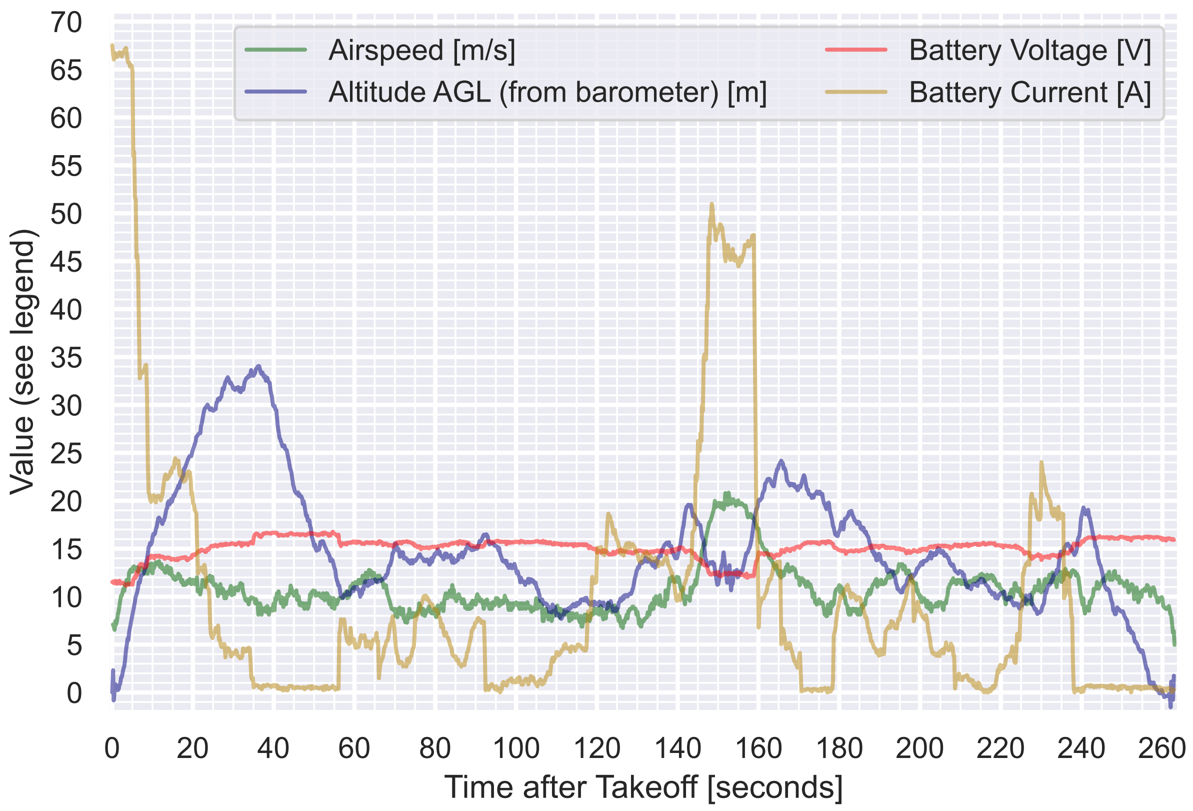


Figure 8-3: Raw sensor data from *Solar Surfer* aircraft test flight

Summary statistics of this data are given in Table 8.1.

Table 8.1: Summary statistics of raw sensor data from *Solar Surfer* aircraft test flight. Based on 1,315 samples per measured quantity.

Description	Airspeed [m/s]	Altitude AGL [m]	Voltage [V]	Current [A]
Mean	10.8	16.4	15.05	10.31
Standard Dev.	2.52	6.81	1.08	13.73
Min	5.0	0	11.13	0.00
25%	9.1	12.3	14.69	0.94
50%	10.3	15.4	15.28	5.25
75%	11.7	19.3	15.67	13.28
Max	20.9	35.6	16.77	67.53

However, there are two primary problems with the data we have. Firstly, all the data sources, and in particular the airspeed sensor, yield noisy measurements. The unsteady corrections that we will later implement require taking the derivative of this data; this is a problem because taking the derivative of noisy data tends to amplify the noise further.

Secondly, the total amount of data is extremely limited, both in duration and sample rate. For example, there are only 1,315 data points *in total* for each of the plotted sensor traces. After eliminating noise (which effectively acts as a low-pass filter on our data, as noise tends to be relatively high-frequency), the effective amount of information we have from the sensors is quite limited. While traditional methods might be able to extract general aircraft flight performance based on this quantity of data (i.e., average required power over the flight), a detailed airspeed-based power curve is not typically achievable with such limited data.

Consider that even the most sophisticated techniques for system identification via statistical inference require the system to be observable in some form - no amount of math can recover a signal that is simply not present in the data. This is difficult, because after noise removal, the data really no more than perhaps a half-dozen "system excitations" from which we can infer aircraft performance. For these reasons, this dataset makes a good example case for what kinds of flight data reconstruction are possible with very limited, noisy data using statistical inference techniques and corrections from unsteady flight physics.

8.2.2 Naive Estimation of Power Curve

The most straightforward approach to estimating the aircraft's power curve is to simply cross-plot the airspeed and electrical power throughout the flight, and then to fit a curve to it.

Figure 8-4 quickly illustrates the infeasibility of this approach. While a general trend is somewhat apparent (higher airspeeds yield higher power draw), the uncorrected data has far too much unexplained variation and noise to make any believable conclusions about the shape of the power curve.

Figure 8-4 also demonstrates the pitfalls of naive curve-fitting in the absence of any kind of physics-based model structure. Perhaps the most severe of these is the extrapolation to physically-impossible conclusions. For example, the naive linear fit shown in blue would imply that the power draw at zero airspeed is negative, which is clearly impossible. The naive cubic fit shown in red initially appears more reasonable, although it too makes erroneous conclusions: it implies that beyond 20 m/s, the power draw decreases with increasing airspeed. Clearly, our intuitions about what the "correct" model would look are not being incorporated into the curve-fitting process, and hence we are leaving information on the table.

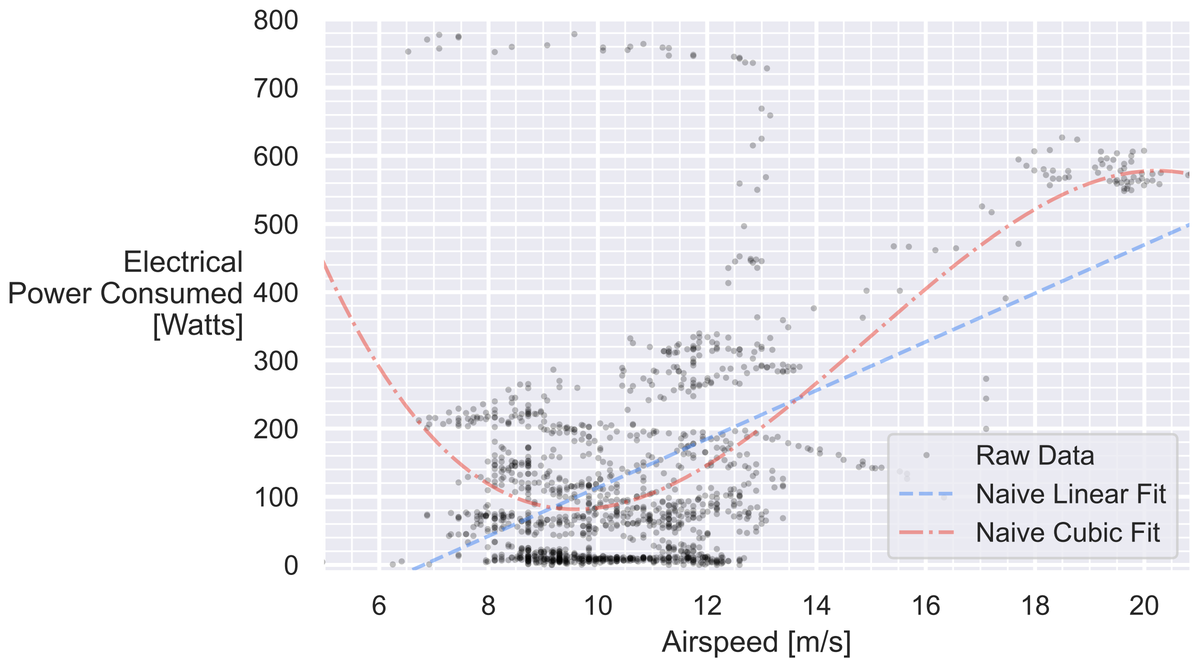


Figure 8-4: Raw uncorrected data for the power curve, with naive curve fitting to demonstrate the infeasibility of this approach. Uncorrected data exhibits high amounts of unexplained variance. Naive curve fits extrapolate to physically-impossible conclusions and do not capture uncertainty.

8.3 Sensor Data Pre-Processing

The first contribution here is to develop a means of pre-processing the sensor data to isolate noise; readers who are more interested in in physics-informed regression than uncertainty quantification may skip forward to Section 8.4. Traditionally, noise is removed by simple averaging; after this process is complete, any uncertainty in the data is typically not considered, and the average is regarded as the ground truth. This works fine for traditional steady analysis where there is a wealth of data that we can average over.

However cases where data is limited provide a strongly incentive to find a way to extract some amount of information from unsteady data. This is difficult because it forces us to directly deal with the sensor noise (and embed priors about the bias-variance tradeoff of data), rather than hand-waving it away by averaging.

As an illustrative example, imagine that we wish to reconstruct a *truth* estimate of the airspeed from the example dataset. This truth value is not directly observable, although data from an associated airspeed sensor is. However, this sensor data has noise; hence, any attempt to reconstruct the truth value must adopt a strategy to separate the noise from the data.

However, there are (literally) an infinite number of possible strategies that one can use to do this noise removal, depending on our embedded assumptions about how sensor noise is entering our data-generating process. For example, consider three different possible state estimations of the vehicle's airspeed, as illustrated in Figure 8-5.

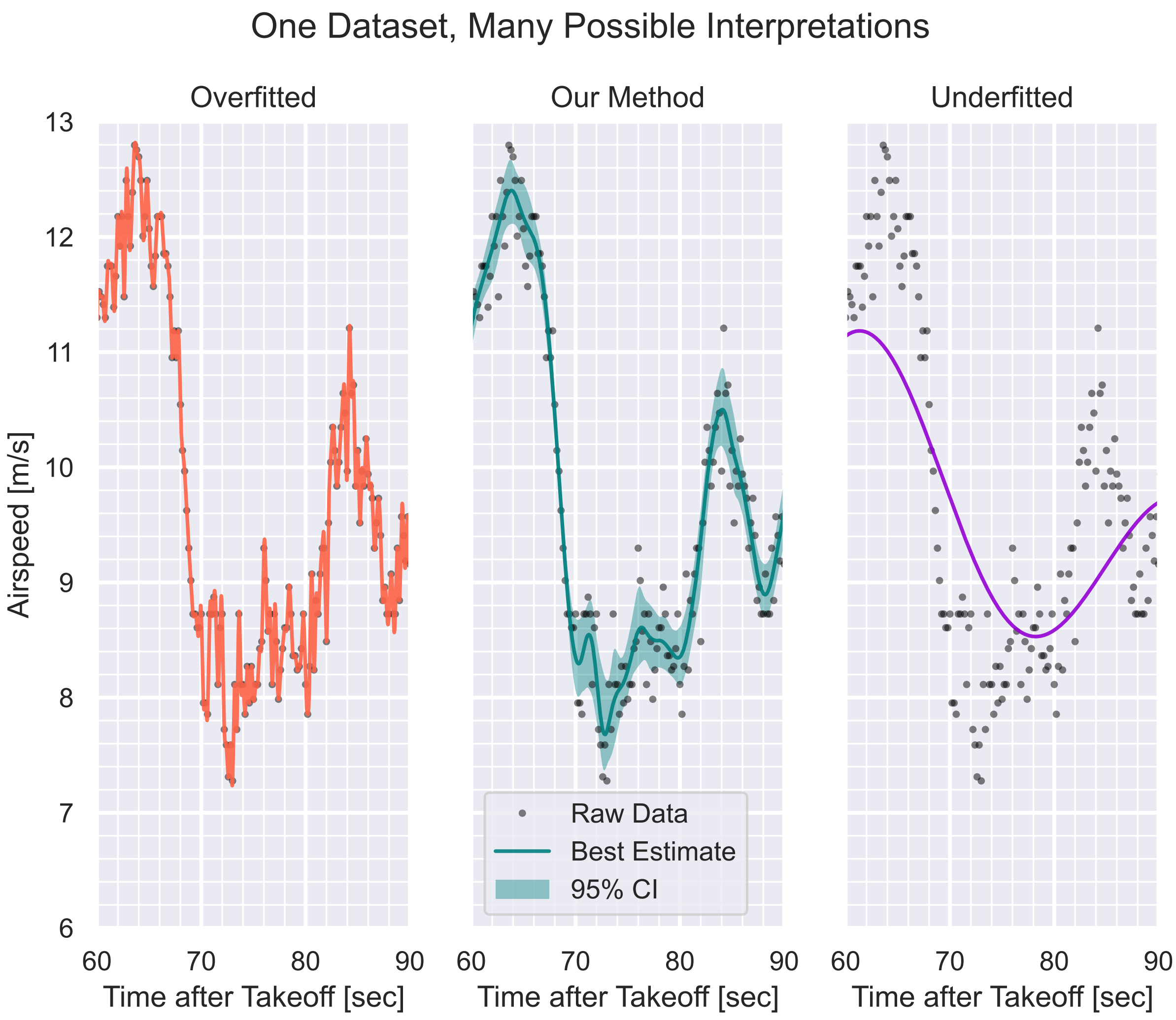


Figure 8-5: Three possible curves that could be used to estimate the true underlying state of the vehicle. Airspeed data, zoomed to $t \in (60, 90)$; a subset of Figure 8-3. 95% confidence interval (CI) of reconstruction shaded in middle plot.

Ultimately, all three charts in Figure 8-5 represent different interpretations of the ground truth based on the data. If we didn't have any more information about the problem, all three could be equally justifiable given the appropriate context—but, in fact, we do have more information, since we know this data originates from a physical system.

The chart on the left of Figure 8-5 is clearly overfitted (i.e., tracks the data too closely) based on engineering intuition. Essentially, we are assuming that each sample is the true data, and that each sample has no noise; hence, the "truth" curve exhibits extremely high variance (strong wiggles). There is no *mathematical* reason why this interpretation can't be correct (it's theoretically possible that the sensor is giving perfect information), but of course, it is is *physically* implausible for our system. If this interpretation were true, it would imply that the vehicle is achieving velocity-vector-aligned accelerations on the order of 2.4 G, which is not physically plausible given the onboard propulsion system.

The chart on the right of Figure 8-5 is clearly underfitted (i.e., tracks the data too slowly), as there are large regions where the estimator has a consistent bias (e.g., consistently too high or low) with respect to the underlying data. It might seem like there are no contexts where such an interpretation would be reasonable, but that is not necessarily the case. For example, if the sensor noise at each sample was not independent, but rather correlated with noise from the previous samples, we might not be able to rule out this interpretation of reality. Of course, a quick glance at the three charts above would tell us "the middle one looks about right", although it's not immediately clear why.

8.3.1 Optimal Sensor Data Reconstruction using Data-Driven Noise Estimates

The key difference is that the middle chart embeds certain assumptions that we intuitively know about our data into its reconstruction. Specifically, the middle chart is an "optimal" reconstruction of the data assuming that:

- The noise in each sample is independent (i.e., uncorrelated with previous samples) and normally-distributed
- The noise is unbiased (i.e., there are no systematic errors in the data, only random ones)
- The noise is homoscedastic (i.e., the standard deviation of the noise is constant across the entire dataset)

- The sample rate is significantly higher than the underlying dynamics of the system that we aim to recover (i.e., the system is "slow" relative to the sample rate), with this assumption quantified later in Section 8.3.1.

Here, an "optimal" reconstruction is defined based on minimization of an ordinary cross-validation function, which is sometimes known as the "leaving-one-out" lemma. An exhaustive mathematical treatment of this metric is given by Wahba [234].

Under these assumptions above, the probabilistic properties of the noise reduce to a single parameter: the variance of the noise σ_n^2. Thus, by constructing an estimator for this statistic, we can recover the probability distribution of the noise and hence an optimal estimator of the data.

Motivation for Finding the Variance of the Noise

The variance of the noise is a useful quantity to estimate for several reasons. First, a probabilistic model of the sensor noise allows us to make optimal choices about the bias-variance tradeoff, armed with a rigorous definition of what optimal means here.

There is a second factor that motivates us to estimate the variance of the noise. Computing an optimal smoothing spline for a time-series dataset is a well-studied problem, but this process is greatly aided if we know the variance of the noise [234]. So, if we can obtain an estimate of noise variance, we've done most of the work required to compute an optimal data reconstructor.

In an ideal world, we would estimate this variance of the noise by taking a large number of samples while the vehicle is at some fixed, steady, known condition. For example, to estimate the airspeed data noise, we might place the aircraft in a wind tunnel at some constant speed. If we ran that experiment for an extended duration, we would know that the true underlying data was constant, and hence any observed variance in the samples would be due to the variance of the sensor noise.

If we want to reconstruct data from unsteady measurement, however, we need to find a way to estimate the variance of the noise using the unsteady data itself, which is much more difficult.

Initial Approach and a First-Order Data-Based Noise Estimator

One way to do this is by assuming that the data is "slow" relative to the sample rate, while the noise remains white. If this is true, then subsequent samples will effectively have the same underlying truth value, but with noise drawn independently from the same underlying distribution.

Quantitatively, consider a scenario where the measured data $x(t)$ consists of a true signal $s(t)$ and noise $n(t)$:

$$x(t) = s(t) + n(t)$$

Assume the noise of the sample $n(t)$ is drawn from $\mathcal{N}(0, \sigma_n^2)$, with initially-unknown σ_n. Now, consider two discrete measurements taken at adjacent points in time t_1 and t_2:

$$x(t_1) = s(t_1) + n(t_1) \qquad\qquad x(t_2) = s(t_2) + n(t_2)$$

If $t_1 \approx t_2$ (as would be the case for subsequent samples), then $s(t_1) \approx s(t_2)$, contingent on the sample rate greatly exceeding the effective fastest frequency of the true signal. Hence:

$$x(t_2) - x(t_1) \approx n(t_2) - n(t_1)$$

In other words, the difference of two subsequent observed samples will be approximately equal to the difference of two independent draws from the same noise distribution. This means that we can estimate the variance of the noise by looking at the variance of the differences between subsequent samples. From the properties of the difference of two independent random normal variables, we can then say:

$$n(t_2) - n(t_1) \sim \mathcal{N}(0,\ 2\,\sigma_n^2)$$

where σ_n^2 is the variance of the sensor noise. So, the following is then approximately true:

$$x(t_2) - x(t_1) \sim \mathcal{N}(0,\ 2\,\sigma_n^2)$$

Therefore, taking the mean difference across all adjacent pairs of measured data $x(t)$ provides an unbiased estimator of σ_n, the variance of the noise:

$$\sigma_n^2 = \frac{1}{2 \cdot (N-1)} \sum_{i=1}^{N-1} \Big(x(t_{i+1}) - x(t_i) \Big)^2 \tag{8.1}$$

where N is the number of samples in the dataset. We call this a *first-order* noise estimator as it takes a first-order numerical derivative of the measured data.

Numerical Demonstration of the First-Order Noise Estimator

We can demonstrate that this works in practice and is convergent to the correct answer by constructing a synthetic dataset with known noise properties, and then reconstructing the noise variance using the method above. The synthetic dataset we use is a simple sinusoid at some frequency f_{signal} with added independent, normally-distributed noise. We then sample this sinusoid at some frequency f_{sample} (ideally with $f_{\text{sample}} \gg f_{\text{signal}}$). We attempt to recover the noise variance using the method above and compare it to the true noise variance. This process is described in Figure 8-6, with results for the first-order estimator in the respective column of Table 8.2. The higher-order estimators are described in the following sections.

Table 8.2: A numerical demonstration of noise variance estimator performance.

			Estimated σ_n		
f_{signal}	f_{sample}	True σ_n	1st-order Estimator	2nd-order Estimator	4th-order Estimator
1	1000	0.1	0.0999	0.1006	0.1008
10	1000	0.1	0.1048	0.1006	0.1008
100	1000	0.1	0.3236	0.1491	0.1016

As shown in Table 8.2, the first-order noise estimator yields a σ_n value that is very close to the true value. We will also show later that this estimator is convergent to the true value as the number of samples increases.

However, this estimator is only first-order convergent with respect to the ratio between the sampling frequency f_{sample} and the underlying dynamics of the system f_{signal}. In other words, if the system is "fast" relative to the sample rate, then the estimator will be inaccurate. This is demonstrated in the third row of Table 8.2, where the estimator produces an incorrect result by a factor of 3.

Second-Order Noise Estimator

To do a better job of estimating the noise variance even in cases where $f_{\text{sample}} \gg f_{\text{signal}}$ doesn't strictly hold, we can use a second-order estimator. With this extension, we use a three-point numerical stencil.

Derivation of this estimator follows similar principles to the first-order estimator above, with the key result as follows:

$$\sigma_n^2 = \frac{1}{6 \cdot (N-2)} \sum_{i=1}^{N-2} \Big(x(t_{i+2}) - 2 \cdot x(t_{i+1}) + x(t_i) \Big)^2 \tag{8.2}$$

where N is the number of samples in the dataset. As shown in Table 8.2, the second-order estimator

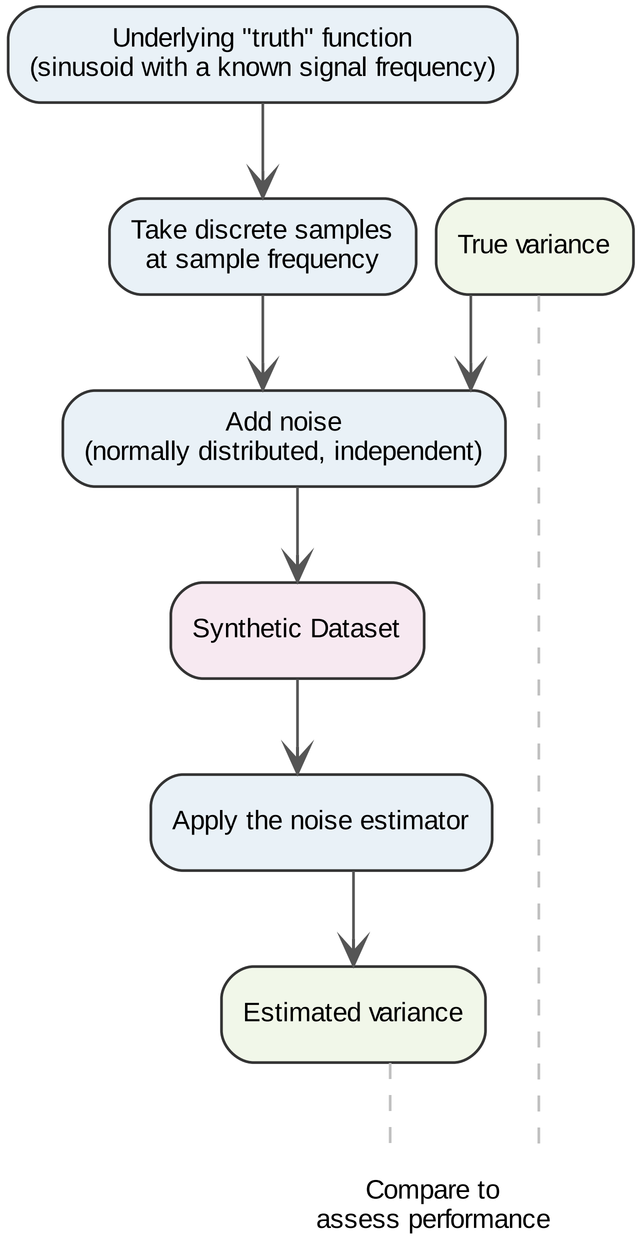

Underlying "truth" function
(sinusoid with a known signal frequency)
Take discrete samples at sample frequency
True variance
Add noise
(normally distributed, independent)
Synthetic Dataset
Apply the noise estimator
Estimated variance
Compare to assess performance

Figure 8-6: Procedure for numerical demonstration of estimator performance.

has superior performance to the first-order estimator when the sample and signal frequencies are not well-separated.

Notably, this effectively implements a first-order, second-degree finite-difference of the underlying data—a discrete derivative. It's instructive to consider the theoretical underpinning of using a discrete derivative operator here. If we assume that the underlying true signal is relatively low-frequency (i.e., smooth), then as we take more discrete derivatives of the observed data, the signal component will asymptote to zero (assuming $f_{\text{sample}} \geq f_{\text{signal}}$). Stated equivalently, the frequency spectrum of the true signal is assumed to have some cutoff frequency (analogous to f_{signal}), above which the power spectral density gradually goes to zero.

In contrast, the noise is assumed to be independent, which means that the noise component of the observed data will not go to zero as we take successive derivatives. Stated equivalently, the frequency spectrum of the noise is white, with uniform spectral power across all frequencies. Therefore, repeated application of the discrete derivative operator acts as a way to spectrally-separate the noise from the signal - a key insight into this method.

Arbitrary-Order Estimators

Clearly, the second-order estimator improves performance compared to the first-order one. Interestingly, we can generalize the logic to an d-th order estimator by observing that the denominator is the sum of the squares of the first-order, d-th-degree, uniform-grid finite difference coefficients[2]. Then, we use the combinatorial trick that:

$$\binom{d}{0}^2 + \binom{d}{1}^2 + \cdots + \binom{d}{d}^2 = \binom{2d}{d} = \frac{(2d)!}{(d!)^2} \tag{8.3}$$

to derive the following d-th order estimator of the noise variance:

[2]Or, perhaps more intuitively, the elements of the d-th row of Pascal's triangle

$$\sigma_n^2 = \frac{1}{\binom{2d}{d} \cdot (N-d)} \cdot \sum_{i=1}^{N-d} \left[\binom{d}{0} x(t_i) - \binom{d}{1} x(t_{i+1}) + \binom{d}{2} x(t_{i+2}) - \binom{d}{3} x(t_{i+3}) \pm \cdots \mp \binom{d}{d} x(t_{i+d}) \right]^2 \tag{8.4}$$

Though equation 8.4 gives the correct expression for this estimator, numerically implementing this requires some care. Appendix C.3 outlines some considerations here and gives an example stable numerical implementation of this estimator.

While we have shown that the second-order estimator has improved robustness to low sample rates, it is not immediately clear that this will hold for arbitrary orders - as the order increases, we might expect to see some form of numerical instability as we take successively higher-order derivatives of noisy data. To test this, we can plot the performance of noise estimators of various orders, as shown in Figure 8-7.

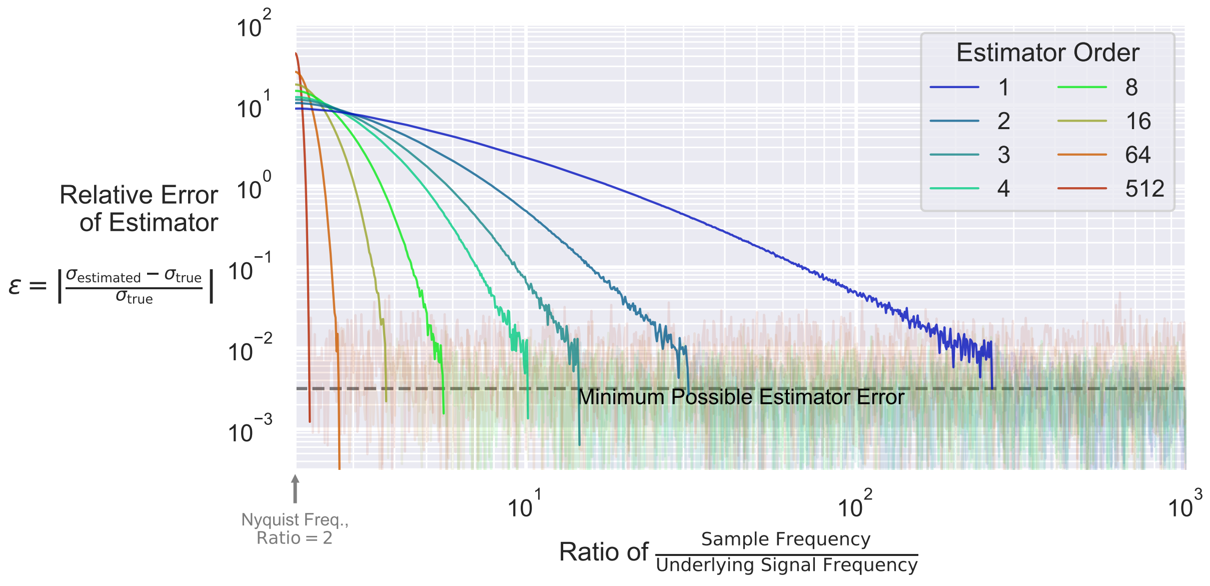


Figure 8-7: Performance of higher-order data-driven noise estimators.

The figure above shows something quite remarkable: we can use extremely-high-order estimators (even up to the 512th order, i.e., taking the 512th-derivative of raw, noisy data via finite differences), and we see not only that it's numerically stable, but also that it accurately estimates the true noise variance even when the underlying signal is almost at the Nyquist frequency.

Interestingly, except for a miniscule region near the Nyquist frequency, estimator performance seems to monotonically improve with increasing estimator order. Since practical data reconstruction would likely have the $f_{\text{sample}}/f_{\text{signal}}$ ratio be much larger than 2, this suggests that one can use extremely-high-order estimators to estimate the noise variance with very high accuracy.

8.3.2 Uncertainty Quantification Using Noise Variance Estimators

With the ability to estimate the amount of noise in the data, we can also estimate the uncertainty of our resulting models and curve fits. To do this, we combine a resampling bootstrap approach (described in [237, 238]) with a noise-variance-aware spline interpolator (described in [234, 237]). Together, these yield not only a best-estimate of the underlying model, but also a measure of the uncertainty in that estimate. An example is depicted in Figure 8-8, which uses the same power-curve dataset as in Figure 8-4.

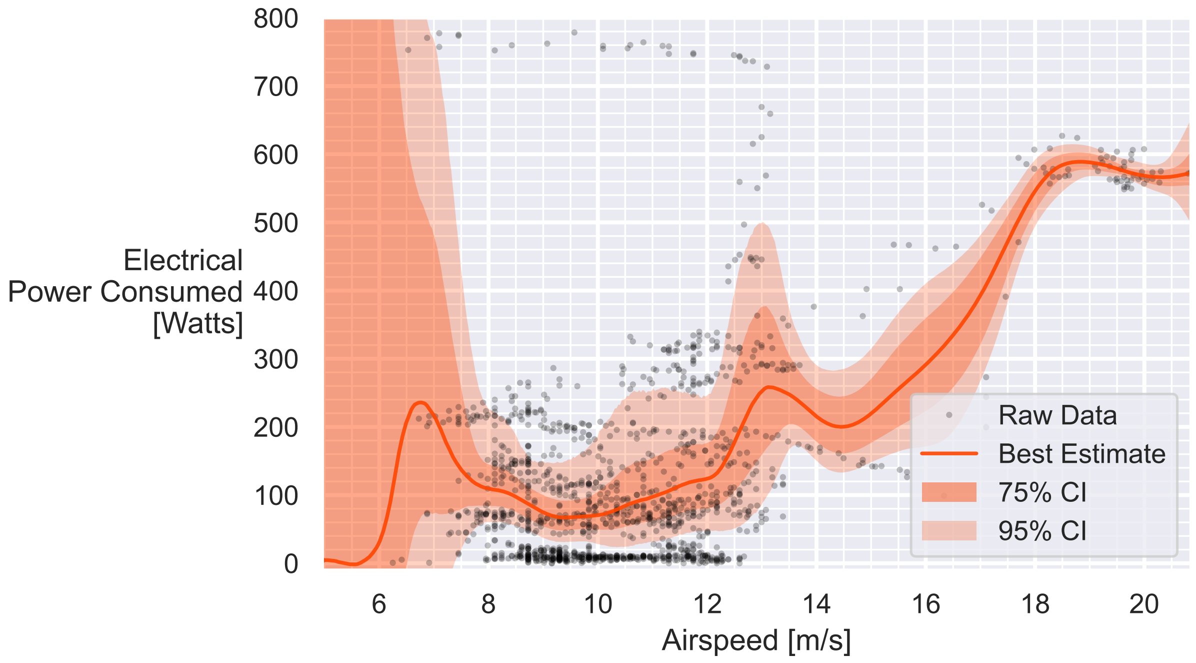


Figure 8-8: Raw uncorrected data for the power curve, with a noise-variance-aware spline model. A resampling bootstrap allows uncertainty quantification. This improves over the naive approach of Figure 8-4, although, the lack of physics-based model structure still leads to the recovery of a physically-implausible result. In Section 8.4, we demonstrate how this reconstruction can be further improved by adding physics-informed model constraints.

Figure 8-8 certainly represents a mild improvement over the naive curve fits shown in Figure 8-4 in that it leverages uncertainty. For example, the uncertainty narrows near 19 m/s where data is abundant, and it widens considerably below 7 m/s where data is sparse. This awareness of model uncertainty reduces our propensity to make statistical conclusions that go beyond what the data can truly support.

Furthermore, this uncertainty quantification allows us to determine where in the flight envelope more data should be collected (i.e., this can serve as an acquisition function). For example, to refine the power curve of Figure 8-8, we might aim to fly future test flights at speeds between 13 and 17 m/s.

With this said, two major problems remain with the approach of Figure 8-8. First, the resulting model uncertainty is extremely high, as shown by the shaded confidence intervals in Figure 8-8. For example, the tightest that we can bound the required power at 10 m/s with reasonably-strong confidence (95% confidence interval) is between 30 and 190 Watts. This is a huge range and not a particularly useful insight. Near the low end of the airspeed range, the model uncertainty becomes so large that even the most basic claims are impossible to assert.

Second, the model still allows physically-implausible relationships, such as the unexplained power curve increase near 13 m/s and the flattening power curve near 20 m/s. Clearly, the uncertainty bounds here are wider than what they would be if our physical insights were incorporated into the model. In the next section, we will show how to improve these results by incorporating physics-based corrections and domain-specific knowledge.

8.4 Incorporating Physics-Based Data Corrections and Domain-Specific Knowledge

When inferring the performance characteristics of a physical system, the laws governing that system can be used to significantly narrow the solution space and reduce the unexplainable variance in the data.

8.4.1 The Aircraft as an Energy System

In the example of the power curve regression, one way to do this is to consider the aircraft as an energy system and exploit the fact that energy should be conserved. Consider the following basic accounting equation for the total energy of an aircraft at some time t:

$$E_{\text{total}}(t) = \frac{1}{2} m\, U(t)^2 + m\, g\, h(t) + \int_0^t \left[P_{\text{loss}}(\tau) - P_{\text{thrust}}(\tau) \right] \, d\tau = \text{constant} \tag{8.5}$$

where: E_{total} = Total energy of the aircraft system. System boundary includes the aircraft but not the onboard stored energy (e.g., battery) or power-to-air (e.g., drag); hence, the representation of these as flow terms. Sign convention is that energy added to the airplane is positive.

m = Mass of the aircraft

$U(t)$ = Airspeed of the aircraft at time t

g = Local gravitational acceleration

$h(t)$ = Altitude of the aircraft at time t

$P_{\text{thrust}}(t)$ = Thrust power $(T \cdot U)$ to the aircraft at time t

$P_{\text{loss}}(t)$ = Power lost from the aircraft at time t; primarily drag, but also other system losses

Note that the above equation is "fully-accounted" in that it includes all energy flows into and out of the airplane, including from onboard sources. Hence, E_{total} should remain constant throughout the flight.

To exploit this knowledge that E_{total} is constant, we differentiate Equation 8.5 with respect to time to obtain an expression for the total power. The total power must equal zero for energy conservation to hold. In writing the following expression, we can also expand the power gains and losses into constituent terms as appropriate for an electric aircraft:

$$P_{\text{total}}(t) = m\,U(t)\,\dot{U}(t) + m\,g\,\dot{h}(t) + \big[U(t)\,D(\theta) + P_{\text{avionics}} - V(t)\,i(t)\,\eta_{\text{propulsive}}(\theta)\big] = 0 \quad (8.6)$$

where: $\dot{(\,)}$ = Time derivative

$P_{\text{total}}(t)$ = Total power at time t. Just as with E_{total}, this is a "fully-accounted" quantity, and hence P_{total} should equal zero.

$V(t)$ = Battery voltage at time t

$i(t)$ = Battery current at time t

$\eta_{\text{propulsive}}(\theta)$ = Propulsive efficiency ($P_{\text{thrust}}/P_{\text{in}}$) as a function of both flight data and unknown parameters θ

P_{avionics} = Power consumed by avionics. In the case of the example dataset, this is assumed to be constant and equal to the average measured power consumption during motor-off periods.

$D(\theta)$ = Drag as a function of both flight data and unknown parameters θ

Ideally, this equation would be used to simply correct the data for the unsteady terms (i.e., acceleration and climb terms) in Equation 8.6, at which point we would directly use that data and apply traditional curve-fitting techniques. However, the observant reader will notice that Equation 8.6 cannot, on its own, be used to correct the data. Both the propulsive efficiency and drag terms are unknown functions of the flight data and unknown parameters θ, and hence these need to be solved for.

These parameters can be defined as the solution of a residual minimization problem (i.e., optimization) between our data and model—effectively, system identification. Here, the residual in question is the instantaneous P_{total} given in Equation 8.6, and we aim to minimize some norm[3] of the residual vector, as resampled from the data reconstruction shown in Section 8.3.1. However, the interesting insight here is that changes to our model parameters affect *both* the model and the corrected data, and hence the model and data must be optimized simultaneously. Stated equivalently: we are not only correcting the data using our models; we are also simultaneously fitting the models to the data.

One other consideration to note with this energy-corrected formulation is that it relies on the time derivative of sensor data. Because our data reconstructors from Section 8.3.1 are based on splines, these can be conveniently differentiated by conducting simple polynomial differentiation piecewise between

[3]Typically the L_1 or L_2 norm

knot points. However, the derivative of the best estimator is not necessarily the bias-variance-optimal estimator of the derivative, due to the amplification of high-frequency variation during the differentiation process. To combat this, we add a Gaussian low-pass-filter with time constant $\sigma = 4$ sec to the derivative estimator. Developing a rigorous method for determining the optimal filter time constant is left as future work, but is theoretically possible using a cross-validation approach.

8.4.2 Parameterizing the Unknown Performance Functions

There are two unknown models in Equation 8.6: the propulsive efficiency and the drag. We replace these with parametric models with forms taken from domain-specific knowledge. In this example, we use simple mathematical relations to demonstrate that even basic restrictions on model form can significantly improve the quality of the resulting model. In practice, more sophisticated models based on parameterized low-fidelity analyses could be used, although this may come at the expense of reduced interpretability.

Drag Model

The aerodynamic drag can be modeled as a function of airspeed, as required to close Equation 8.6. To do this, we use the following procedure:

1. Compute the instantaneous lift coefficient $C_L(t)$ from the airspeed $U(t)$, assuming a load factor of 1 and known aircraft properties (wing area, mass, etc.).

2. Map the C_L to a drag coefficient C_D using a parametric model with unknown parameters θ_1 to θ_6. The form of this model is a piecewise-quadratic $C_D(C_L)$ function, as shown in Figure 8-9. This model form is identical to the assumed form of the profile drag function in the AVL [21]

aerodynamics code. In this model form:

- (θ_1, θ_2) represents the point of positive stall onset
- (θ_3, θ_4) represents the minimum drag point
- (θ_5, θ_6) represents the point of negative stall onset

The piecewise-quadratic function is constructed to be C^1-continuous between the four regions delineated by these points. This requirement fully-constrains the non-stalled regions of the curve. The regions above positive stall and below negative stall use an assumed curvature (identical to the values used in AVL [21]) that generally models post-stall drag rise.

3. Dimensionalize the computed C_D to obtain the drag.

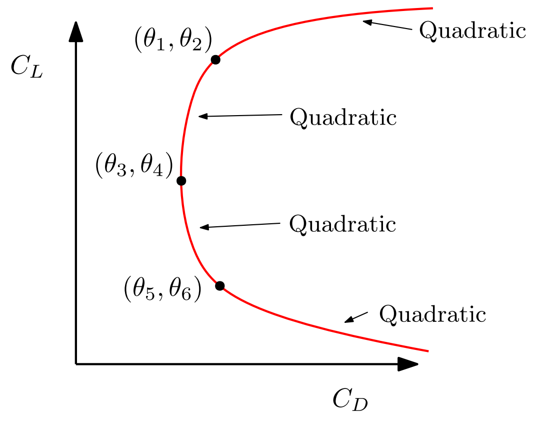


Figure 8-9: Assumed parameterized form of the aerodynamic polar, where drag coefficient is a piecewise-quadratic function of the lift coefficient. Same mathematical form as the profile drag function used by AVL [21].

Propulsive Efficiency Model

Similarly, a characteristic model for the propulsive efficiency is defined based on domain-specific knowledge. In an ideal case, this model would take the propeller advance ratio J as an input and yield the propulsive efficiency $\eta_{\text{propulsive}}$ as an output. However, the advance ratio is not directly measurable in this *Solar Surfer* case study, as this requires propeller RPM data which is not available. However, we do have sufficient data to infer a quantity known to differ from the advance ratio J by a constant factor; we denote this proportional quantity c_J.

Therefore, the propulsive efficiency model is a mapping from c_J to $\eta_{\text{propulsive}}$, rather than from J to $\eta_{\text{propulsive}}$. Although this means that the maximum-efficiency advance ratio J cannot be estimated, the peak propulsive efficiency itself can be estimated; ultimately, it is the peak efficiency that is the primary performance metric of interest.

To construct this model, we use the following procedure:

1. Use the measured current $i(t)$ to estimate c_n, a quantity proportional to the propeller rotational rate n, using the $P_{\text{in}} \propto n^3$ relationship found using the Buckingham π theorem [162]:

$$c_n = i(t)^{1/3} \propto n$$

2. Propagate this to estimate c_J, a quantity proportional to the propeller advance ratio J:

$$c_J = \frac{U(t)}{c_n} \propto J$$

3. Map c_J to a propulsive efficiency $\eta_{\text{propulsive}}$ using the following model form and unknown param-

eters θ:

$$\eta_{\text{propulsive}} = \theta_7 \cdot \text{softmin}\left(c_J/\theta_8, \quad \frac{c_J - \theta_9}{\theta_8 - \theta_9} \right) \tag{8.7}$$

where softmin is a smooth approximation of the minimum function, analogous to the logsumexp definition of softmax by Cook and others [239]:

$$\text{softmin}(x, y) = -\theta_{10} \cdot \ln\left(e^{-x/\theta_{10}} + e^{-y/\theta_{10}}\right)$$

Note the physical interpretation of several parameters here:

- θ_7 provides an upper bound on the peak propulsive efficiency
- θ_8 represents the c_J value corresponding to peak efficiency
- θ_9 represents the c_J value corresponding to the "pitch speed" of the propeller, where the thrust (and hence efficiency) goes to zero.
- θ_{10} is a smoothing parameter that controls the sharpness of the transition between the two regions.

This model form captures expected behavior of the propulsive efficiency curve, including the fact that the propulsive efficiency is zero at zero airspeed and zero at the pitch speed of the propeller. The θ_7 parameter is constrained to be less than one to bound efficiency to physically-plausible values. The resulting model form matches well with expected propeller efficiency trends, such as the curves seen in [162].

8.4.3 Solving the Optimization Problem

With these models in place, a residual minimization problem of the following form can be solved to yield optimal θ values, where P_{total} (the net power into the system, after all inflows and outflows should be accounted for) from Equation 8.6 is used as the residual:

$$\begin{aligned} \underset{\theta}{\text{minimize}} \quad & \sum (P_{\text{total}})^2 \\ \text{subject to} \quad & \theta \text{ constraints described in Section 8.4.2} \end{aligned} \tag{8.8}$$

Optionally, other norms of the P_{total} residual vector can be minimized instead. The L_1 norm makes a particularly compelling choice here due to its robustness to outliers [217]. In the example problem here, we solve this optimization problem numerically using IPOPT [128] via the AeroSandbox [1] framework for engineering design optimization.

8.4.4 Results

With the optimization problem solved, we can now use the resulting θ values to compute the corrected data and model. This is shown in Figure 8-10. The corrected data and model are now in excellent agreement, and the corrected data has significantly less unexplained variance than the raw data. The remaining unexplained variance is attributable to a variety of factors, such as wind updrafts and downdrafts and non-unity load factors during banked flight. In theory, both these effects could be corrected out using a similar inference procedure that is shown in this work (which could be augmented by, for example, IMU data). This is left as future work, which, if successful, would reduce the data spread in Figure 8-10 further.

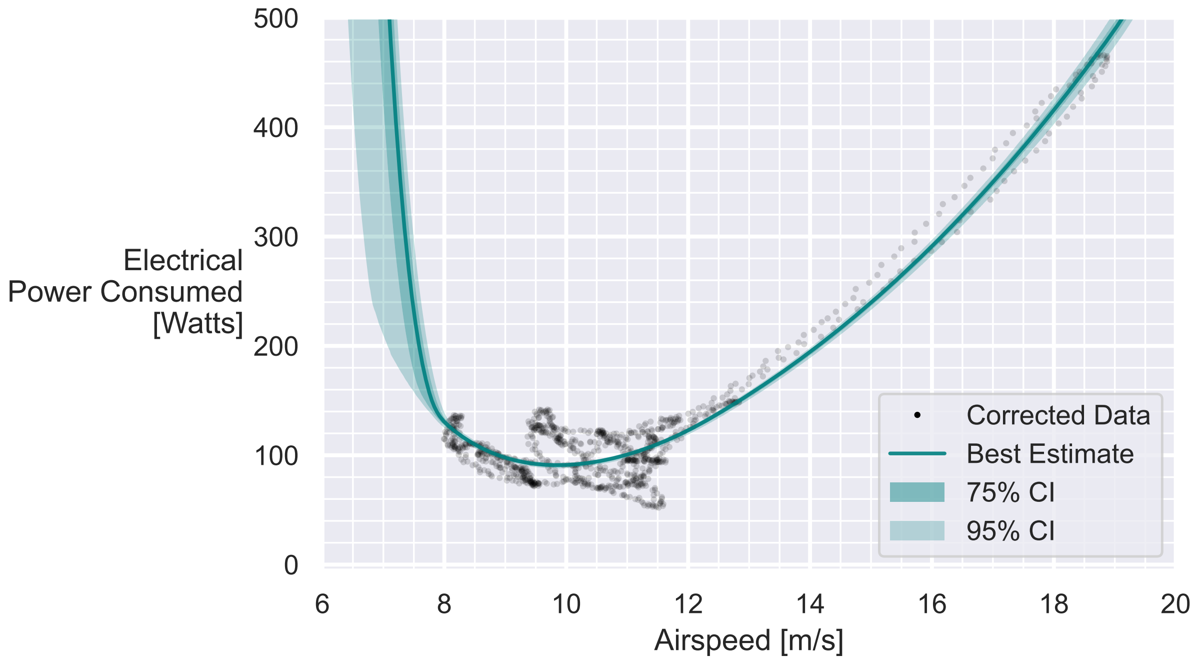


Figure 8-10: Inferred power curve with physics-informed corrections on data and model, and using noise estimator with a resampling bootstrap for uncertainty quantification. Power curve agrees closely with the data and shows physically-plausible behavior. Uncertainty quantification is significantly tighter than in Figure 8-8, as the function space has been restricted using embedded physics knowledge.

Conveniently, this process yields other critical performance information, like the aerodynamic polar and propulsive efficiency curve, "for free" as part of the fitting process. These are shown in Figures 8-11 and 8-12. Furthermore, we also obtain uncertainty estimates for all these quantities, which are critical for understanding the confidence we can have in the results.

A notable observation is that the inferred aerodynamic polar of Figure 8-11 and the propulsive efficiency curve of Figure 8-12 show relatively wide uncertainty bands, while the power curve of Figure 8-10 shows a relatively narrow uncertainty band. This is interesting, since in typical analysis the power curve is computed as a function of these two performance relations; hence, one would intuitively expect that the power curve would inherit the uncertainty of both relations. The reason this does not occur here is that the uncertainties in the aerodynamic polar and the propulsive efficiency curves are not independent. In other words, the power required can be inferred relatively precisely, but there is not quite enough data to rigorously determine whether power losses are more due to aerodynamic drag or propulsive inefficiency.

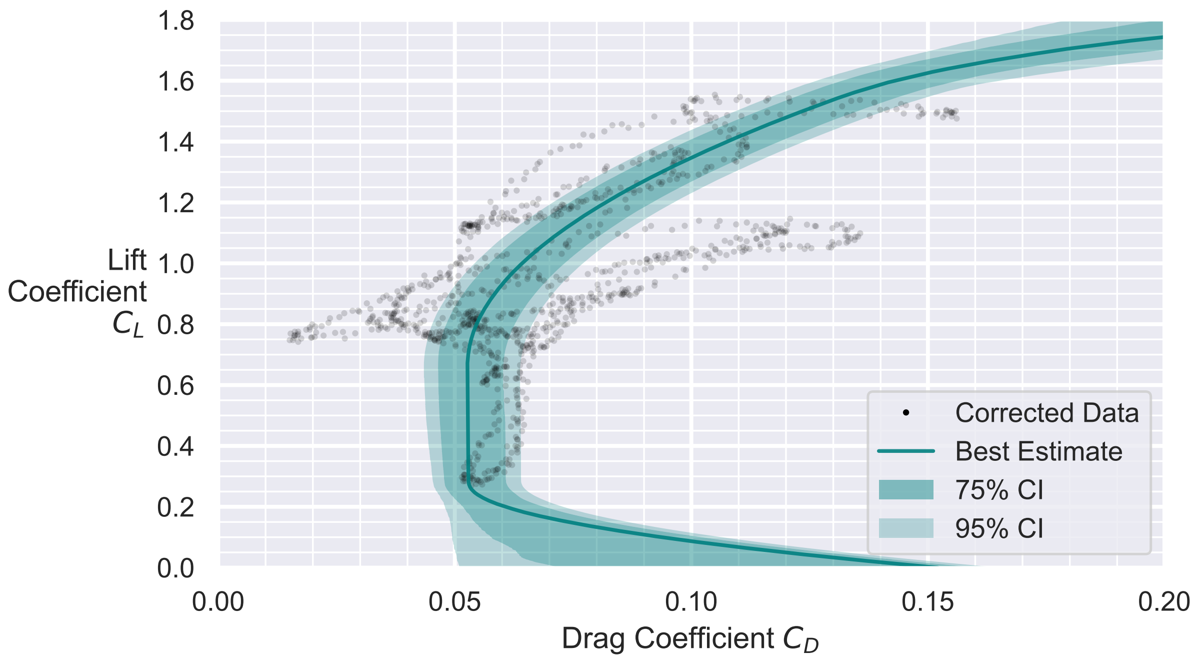


Figure 8-11: Inferred aerodynamic polar with physics-informed corrections on data and model, and using noise estimator with a resampling bootstrap for uncertainty quantification.

Compared to the naive approach in Figure 8-4, Figure 8-10 enables us to make very accurate, physically-grounded, and uncertainty-quantified assertions about the performance of the airplane, based on minimal data (here, only four minutes of flight time).

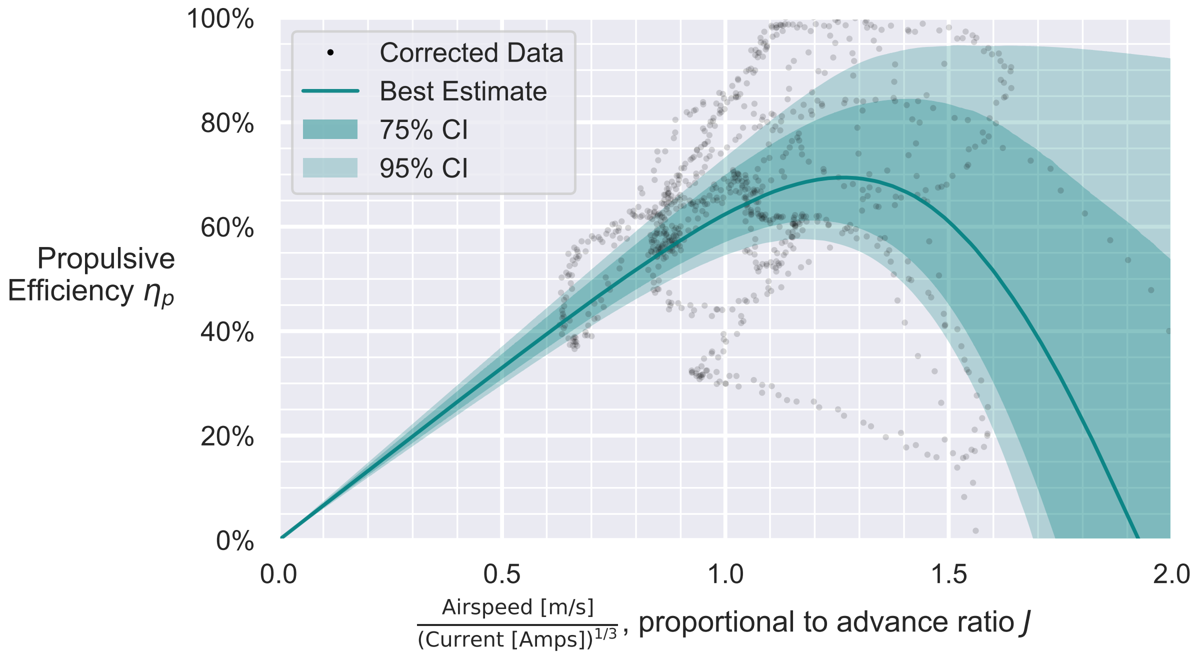


Figure 8-12: Inferred propulsive efficiency curve with physics-informed corrections on data and model, and using noise estimator with a resampling bootstrap for uncertainty quantification.

8.5 Computational Reproducibility

All code and data used in this chapter is publicly available at `https://github.com/peterdsharpe/aircraft-polar-reconstruction-from-flight-test`.

8.6 Summary and Future Work

In this chapter, we have demonstrated a rigorous method for reconstructing the performance characteristics of an aircraft from flight test data. This method is based on a physics-informed correction of the data and model, and is shown to be effective at recovering the true power curve and aerodynamic polar of an example aircraft from noisy, limited, unsteady data. This method is also shown to be robust to the presence of outliers in the data, and to be able to provide uncertainty estimates for the resulting aerodynamic polar. This method is also shown to be able to recover the propulsive efficiency curve of the aircraft, which is a critical quantity for understanding the performance of the aircraft. Finally, this method is shown to be able to recover the aerodynamic polar of the aircraft from only four minutes of flight test data, which is a significant reduction in the amount of data required compared to the state of the art.

This work has focused on the ability of these inference-based reconstruction procedures to significantly reduce the required flight hours to characterize an airplane's performance. However, an alternative way to "spend" this increased ability to extract information is on massively reducing instrumentation costs. For example, this approach makes it conceivable to "estimate out" the ambient wind field[4] from groundspeed data and power measurement alone. If this is possible, it would enable flight test engineers to perform sufficient early-stage characterization of a new crewed airplane using nothing other than the sensors available on a modern smartphone. Here, GPS, IMU, and barometer sensors provide position and groundspeed information, and the phone's microphone can be used to infer engine RPM and hence power and advance ratio from an audio spectrum. If successful, this would be an enormous reduction in instrumentation cost and time compared to the state of the art.

A fair point of hesitancy here might be the fact that applying these corrections to recover useful insights from noisy, limited, unsteady data takes a significant amount of algorithmic development and mathematical complexity. However, algorithms are cheap, and more importantly, scalable - so, once a computational routine is established, it can easily be applied to a large number of flight test campaigns. By contrast, flight-test hours and improved instrumentation represent a recurring high cost. Therefore,

[4]An intuitive explanation for this is that if an airplane loiters in a circle at constant power setting, the upwind and downwind groundspeeds yield information about the wind field. In practice, some additional assumptions about wind steadiness and uniformity would likely be required to collapse the (x, y, z, t) space; otherwise, it is likely too large to realistically span.

it is worth investing in the development of algorithms that can extract the maximum amount of useful insight from the data that we do collect, even if this analysis is somewhat less simple.

Chapter Acknowledgments

The authors would like to thank the MIT AeroAstro 16.821 course staff for their support of the work in this chapter, and the MIT AeroAstro department for their support of the course. The aircraft used to collect the example data in this chapter was constructed by the MIT AeroAstro 16.821 undergraduate student team.

Chapter 9

Conclusions

This thesis introduces five new contributions to aircraft design, multidisciplinary design optimization, and scientific machine learning. These include:

1. A code transformations paradigm for engineering design optimization (Chapter 3), and a reference implementation in the open-source Python library AeroSandbox [38]; also includes a series of

aircraft design case studies to demonstrate this framework (Chapter 4)

2. Traceable implementations of aerospace physics models (Chapter 5)
3. Sparsity tracing via NaN-propagation (Chapter 6)
4. Integration of physics-informed machine learning surrogates into a code transformations framework (Chapter 7)
5. A novel approach to aircraft system identification from minimal flight data (Chapter 8)

These contributions are synthesized into a unified framework for engineering design optimization, AeroSandbox, which is demonstrated throughout on a variety of aerospace design problems. This framework is designed to be high-performance, mathematically-flexible, and user-friendly, in order to address various frictions that industry users currently experience with design optimization workflows (Chapter 2). If this thesis is to be distilled into a series of key takeaways, they would be as follows:

1. Code transformations have enormous potential to improve not only the runtime speed of engineering design optimization problems, but also the ease of model development and implementation. This reduction in the barrier to entry for design optimization can help academic advances translate more readily into industry practice.

2. When developing a new multidisciplinary design optimization (MDO) framework, the decision to use a code transformations paradigm is best considered from the outset of the framework's development. Code ultimately must be written with traceability in mind, and this burden will inevitably either be put on the end-user (undesirable) or on the framework developer (preferred). If the latter is to be achieved, the framework itself must be built on top of a unified numerics stack

that allows computational graphs to be traced[1].

We live in a time of great bounty in scientific computing, thanks to abundant computational power, fundamental algorithmic advances, and widespread accessibility through the vibrant open-source community. These factors *should* supercharge design engineers, but these theoretical improvements have not been reflected in reality – largely due to the difficulty of wielding these tools. Thus, the onus upon tool developers to focus on advancing not just traditional performance metrics, but also usability and interoperability. The contributions of this thesis are intended to address—and kick off further research regarding – the intersection of scientific computing and the practical realities of engineering design, so that as a community we may work toward bridging the math-to-metal gap.

[1]As a result, an MDO framework that only adds in automatic differentiation support to individual models after the fact will have more user frictions than a framework that supports this end-to-end. The discussion of limitations around code syntax and style in Section 3.4 provides examples of why this occurs.

Appendix A

Extended Comparison of MDO Framework Paradigms

This appendix aims to expand on the subjective comparisons made in Table 3.1 by providing more details on the pros and cons of various MDO paradigms. In this table, three qualitative metrics are defined over which MDO framework paradigms are evaluated. These metrics represent framework needs derived from the barriers to industry adoption that are identified in Figure 2-2 and the remainder of Chapter 2. Here, we define those metrics more precisely:

1. **Ease of implementation:** How much effort is required to implement a typical aircraft design problem (from "concept idea" to "working code") in this paradigm? How much optimization or programming expertise is required, beyond the basics needed to write engineering analysis code?

2. **Computational speed and scalability:** How fast is the resulting design optimization problem to run, and how does this scale with problem size and number of disciplines? Are there other fundamental limits to scaling up analysis fidelity (e.g., poor memory scaling)?

3. **Mathematical flexibility:** What kinds of restrictions are present on the mathematical form of the optimization problem? This has important follow-on effects for backwards-compatability, as highly-restrictive frameworks preclude the use of existing engineering code and instead require from-scratch rewrites of analysis code.

From here, we can provide some notes on how various MDO paradigms are assessed:

Black-Box Optimization

Black-box optimization refers to the standard approach of taking an existing performance analysis toolchain and wrapping it in an optimizer. An example of this is shown in Figure A-1. Here, the user has a "black box" function that takes in a vector of design variables and outputs a scalar objective and a vector of constraints. The user then wraps this function in an optimizer, without providing any other information (e.g., gradient or sparsity) about the function. The moniker "black box" refers to the fact that the internal workings of this analysis function are essentially opaque to the optimizer.

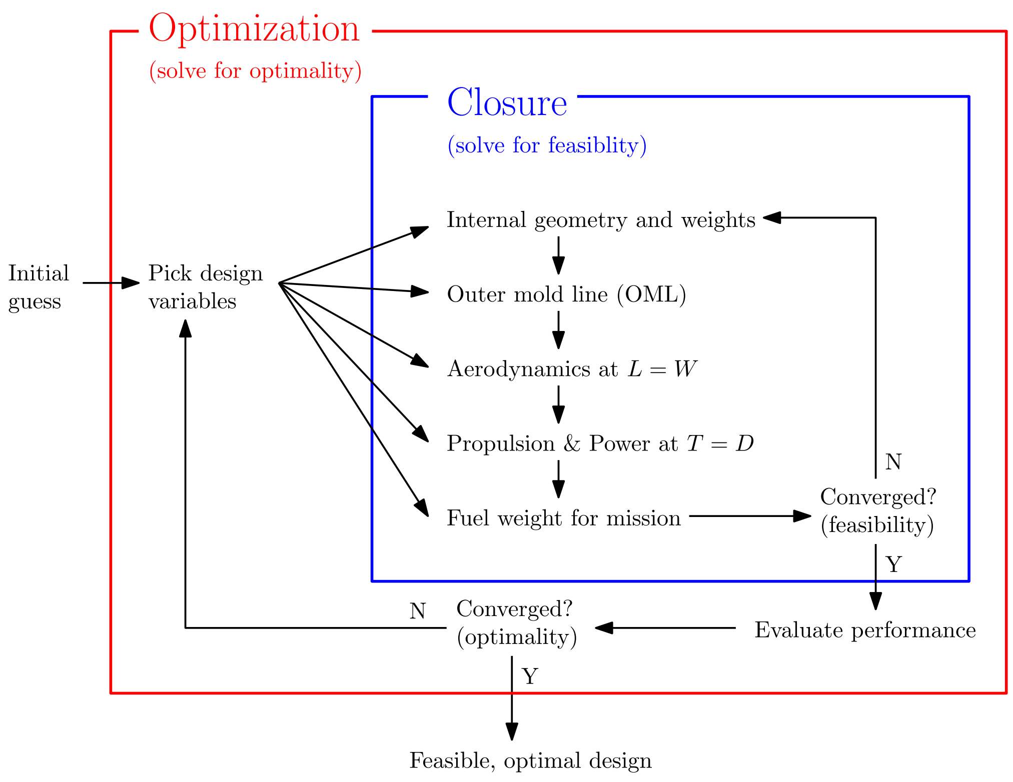


Figure A-1: A representation of a traditional "black-box" wrapped-optimization approach, which remains the dominant MDO paradigm used by industry practitioners today. Figure adapted from [4].

The main benefit of this approach is its conceptual simplicity: a black-box optimization paradigm tends to map most directly onto the "mental model" of optimization that most practicing engineers have. In this mental model, the forward problem (i.e., analysis) is the basis, and an inverse solve (i.e., optimization) is bolted on afterwards to wrap this. Because of this, black-box optimization is assessed to have the easiest path to implementation in industry, and indeed it is the most common optimization paradigm in industry today. Modeling flexibility is likewise excellent due to the minimal information required by the optimizer—almost all mathematical model forms that would be seen in typical analysis codes can be interfaced with the solver.

However, this approach has some drawbacks as well:

1. Computational performance invariably degrades rapidly as the number of design variables is

increased. If a gradient-free optimizer is used, this is due to the curse of dimensionality, where the size of the feasible set of the design space grows exponentially with the number of design variables. With a gradient-based optimizer, scaling is better, but black-box optimization still falls sharply behind more advanced paradigms due to the bottleneck of gradient computation via finite differencing.

2. Convergence issues may exist if the wrapped analysis requires internal closure loops to be satisfied. If a wrapped analysis cannot achieve closure (i.e., feasibility) with a given set of inputs, the optimizer receives essentially no information that allows it to recover from this; this can render the optimization process brittle. Likewise, these closure loops can be inefficient – lots of computational effort is spent closing iterative feasibility loops early in the design process, when the design is far from optimal.

3. Finally, the black-box optimization approach typically forces a functional coding style (i.e., a callable data structure with defined inputs and outputs) in order to interface with an external optimizer. This can be unnatural to read and write, as engineers typically write analysis code in a procedural style. While simple analyses may allow easy conversion between functional and procedural coding styles, this task becomes substantially harder for larger codebases split across multiple files or modules.

Because of these reasons, the runtime speed and scalability of black-box optimization methods is deemed relatively poor.

Gradient-based with Analytic Gradients

Gradient-based optimization methods that are manually augmented with user-provided analytic gradients offer substantial runtime performance improvements over black-box optimization methods – indeed, to-date this is the dominant approach in deep MDO methods where runtime speed is the primary limitation [81], like RANS-based shape optimization. However, deriving and implementing derivatives for each model can be a Herculean effort that is challenging to scale to wide MDO methods with hundreds of constituent models [85]. Moreover, this process requires significant end-user expertise and is often tedious and error-prone. Indeed, erroneous gradient calculations can be some of the most persistent and difficult errors to detect and debug [71]. Due to the user expertise and engineering time required to implement each model in such a framework, this paradigm faces an uphill battle for conceptual design in industry.

The modeling flexibility of these methods approaches the total freedom that black-box optimization affords, although the requirement for differentiability (more precisely, C^1-continuity) can preclude certain types of conditional logic. In practice, this restriction is usually not overly burdensome. If analysis code is scratch-written to be compatible with such a framework, workarounds are usually possible; however, compatibility with existing codes may not be possible in all cases.

Disciplined Optimization Paradigm

Disciplined optimization methods, such as geometric programming or convex programming methods, offer the ease-of-use of black-box optimization methods with the runtime speed of gradient-based methods with analytic gradients [7, 83]. Moreover, they carry notable additional benefits by virtue of their convex formulation: no initial guesses are required, and any optimum must be a global one.

These benefits are achieved by restricting the space of mathematical operators, which limits the models that can be implemented into such a framework. Because of this restriction, model flexibility is scored low. Fortunately, geometric programming maps relatively well onto many sizing relations found in aircraft conceptual design, since many of these are power-law-like relations [46]. However, even in conceptual aircraft design relations, a significant number of exceptions to GP-compatibility exist, and this can be labor-intensive to resolve [240]. Furthermore, this model inflexibility leaves little ability to scale up the level of fidelity to include common mid-fidelity analysis elements like nonlinear systems of equations or integrators. For this reason, disciplined optimization methods are scored lower on scalability, although they are quite competitive on runtime speed when the nature of the problem allows it to be compatible with such a framework.

Code Transformations

On the metric of ease-of-implementation, code transformations are scored as identical to disciplined optimization paradigm. Both approaches allow a procedural modeling-language-like approach that tends to map closely onto existing engineering analysis code. Optimization components, like the variables, constraints, and objective, can be specified in natural-language syntax. Both are scored slightly below black-box optimization methods, since they both require understanding of certain paradigm-specific concepts. (In the case of disciplined optimization methods, this is convexity or GP-compatibility. In the case of code transformations, this is traceability.)

Code transformations offer runtime speeds that equals or exceed those of dedicated disciplined optimization solvers on relevant problems. This paradigm also offers a pathway to medium-fidelity analysis that is often not possible with disciplined optimization methods. However, code transformations are relatively memory-hungry due to the storage of a complete computational graph; this precludes the high-fidelity analyses (e.g., RANS CFD) that are possible in an analytic-gradient paradigm. Likewise, hand-derived gradients can implement accelerations that are not yet possible with code transformations, such as more aggressive sparsity accounting, subtractive cancellation, and common subexpression elimination [71, 96, 133]. For these reasons, runtime speed and scalability falls somewhere between that of disciplined optimization methods and gradient-based methods with analytic gradients.

Finally, modeling flexibility is vastly improved over disciplined optimization methods, as the only fundamental requirement is that of C^1-continuity, similar to that of other gradient-based methods. However, mathematical operators should be drawn from a set of pre-defined primitives [97], which has the potential to reduce modeling flexibility if the numerics framework does not make appropriate syntax choices to mitigate this [103]. For these reasons, flexibility is assessed as slightly below that of gradient-based methods with analytic gradients.

Appendix B

Design Optimization Rules of Thumb

This appendix includes content from the author's prior Master's thesis, revised with new insights [1].

Here, we share some general advice for practical engineering design optimization that the author has collected over the years, across various aircraft development programs. These are especially applicable to the conceptual design of engineering systems, but these guidelines can be considered in any part of the design cycle:

1. **Engineering time is often part of the objective function.**

   (a) As the saying goes, 80% of the results come from the first 20% of the work: low-fidelity models go exceptionally far.

   (b) Make convenient modeling assumptions often and judiciously. (Of course, track these assumptions and revise models if needed.) It is often much more time-efficient to start low-fidelity and only increase fidelity as needed.

   (c) Identify early which models, requirements, and assumptions are sensitive, and estimate uncertainties associated with each of these. Allocate the vast majority of engineering time to risk reduction of only these sensitive elements.

   (d) Hands-on hardware prototyping of subsystems is one of the most direct ways to reduce risk; consider freezing the design early and often to facilitate this. Be wary about falling into "analysis paralysis".

2. **For conceptual design, modeling "wide" rather than "deep" often yields more useful design insight.**

   (a) Generally, the conceptual design studies that are the most practical, useful, and robust are those that model a vast number of disciplines at low fidelity, rather than those that model one or two disciplines at a high fidelity. Value simple physics models, and let the complexity come naturally from emergent cross-disciplinary behavior.

   (b) In instances where high fidelity is truly required, consider surrogate modeling and reduced-order modeling. It is of paramount importance that the optimization problem can be solved in seconds or minutes. If this is not the case, interactive design becomes prohibitively tedious, and extracting engineering intuition becomes difficult.

3. **When doing conceptual engineering design, be principled about how uncertainty is bookkept.**

    (a) Analysis results used in MDO should be an *unbiased* estimator of the author's belief state about some quantity. This is often *not* the raw output of a first-principles physics model, since these often bias optimistic due to assumptions[1]. A good rough litmus test for biased outputs is to consider whether one would be more surprised to learn that the model was erring high versus low—these should ideally carry roughly equal surprise.

    (b) Account for margin explicitly, and usually only in top-level closure loops and parameters (weight, drag, power, load factor, etc.). Remember that margin is performance beyond the *limit* case, not the *baseline* case[2]. Also, corrections for suspected bias do not count as margin: margin is for mitigating unknown unknowns, not known unknowns.

4. **Do not blindly trust an optimizer.**

    (a) An optimizer solves the problem given to it (ideally!), not necessarily the problem intended. Often, it is easy to forget constraints that seem intuitively obvious.

    (b) When one is developing models, one's relationship with the optimizer is adversarial. Models should extrapolate sensibly and generally be parsimonious—errors will be *actively* exploited.

[1]As aerospace examples: a) a RANS CFD study on defeatured geometry that ignores flap track fairings, skin gaps, or protuberances; b) an XFoil analysis that assumes laminar flow is still achievable beyond a line of leading-edge rivets; or c) a wing weight build-up that neglects *expected* weight growth during detailed design. These *expected* biases should be corrected before inclusion in a design tool, and these corrections *do not count as margin*.

[2]For example, in aircraft design, thrust margin is the excess beyond what is needed for *climb* (possibly with one engine inoperative), not *cruise*. If performance is *ever* intended to be used during nominal limit-case operation of the system, this cannot be counted as margin.

Because of this, it takes much more finesse to write an analysis tool that is amenable to optimization than one that merely solves the analysis problem.

(c) Consider a design's robustness to off-nominal conditions early. In nature[3], optima are usually not near extremes.

5. **Strange results are nearly always the "right solution to the wrong problem", rather than the "wrong solution to the right problem".**

   (a) A strange solution, error, or infeasibility/unboundedness (where this was not expected) usually indicates a mistake in the problem formulation[4]. Look for cunning discrepancies between the spirit and the letter of the design code. Most of the time, the issue is in missing, ill-posed, overly-tight, or inadvertent constraints.

   (b) When using gradient-based optimizers, user-specified models that mathematically violate C^1-continuity are another frequent source of non-convergence.

   (c) Another common source of strange results is very-high-dimensional design problems that optimize around a single operating point [33]. This can lead to extremely brittle designs that ostensibly perform well in a narrow region around the design point but poorly elsewhere (and in practice).

   (d) If initial guesses or problem scales are off by many orders of magnitude, this can cause slow convergence[5]. Strong nonconvexities (e.g., a model interpolating noisy data) can also cause problems.

[3]Mother Nature being arguably history's most successful optimizer

[4]If you see a weird design, 9 times out of 10 there's a formulation mistake. But, 1 out of 10 times, you've stumbled upon a clever non-intuitive design breakthrough.

[5]However, typically these factors will not affect the value of the optimum, if it is found. This is true if a) an optimizer like IPOPT is used, which only terminates when KKT-like local optimality conditions are satisfied, and b) the problem is not multimodal enough that convergence to a different local minimum is a concern.

6. **Build models incrementally, and track changes.**

   (a) Resist the tendency to build a large design tool from the start “in one fell swoop”, without testing the code along the way. Doing so makes it enormously difficult to debug when the problem formulation is inevitably found to be flawed. Instead, build a simple minimum-viable design tool first, where the only constraints are high-level design closure[6]. (In other words, start by just performing basic napkin-math sizing, not optimization.) Then, add new variables, constraints, and analyses incrementally, testing the code with each new feature.

   (b) Use a version control system (e.g., Git) to track this iterative development of the problem formulation. This allows rapid identification of when and where a mistake was introduced.

7. **Optimization is just one tool in the design toolbox.**

   (a) An optimizer will answer sizing questions posed by an engineer, but it will not ask new questions on its own or sanity-check these results.

   (b) Resist the urge to justify design decisions with “because the optimizer said so”. The true goal of design optimization is less to arrive at the final design, and more to provide a way for the engineer to explore and understand the design space (by collapsing its dimensionality in a principled way[7]). In most cases, taking the time to understand why the optimizer produced a certain result is more valuable than the result itself.

[6]Usually, this involves weight/lift, thrust/drag, and power closures.

[7]by projecting from a very high-dimensional original design space to a low-dimensional manifold containing an “optimal” subset, based on some trade-space parameters of interest.

Appendix C

Extended Code References

C.1 AeroSandbox Benchmark on Beam Static Structural Analysis

This Python code is used for the AeroSandbox performance benchmark used in Section 3.3.2, with associated runtimes shown as the line labeled "AeroSandbox" in Figure 3-6. In this analysis problem, a cantilever beam is subject to a distributed load, and the goal is to compute the state of the deflected beam. The code is shown in Listing 5.

```python
import aerosandbox as asb
import aerosandbox.numpy as np

N = 50  # Number of discretization nodes
L = 6  # Overall length of the beam [m]
EI = 1.1e4  # Bending stiffness [N*m^2]
q = 110 * np.ones(N)  # Distributed load [N/m]

x = np.linspace(0, L, N)  # Node locations along beam length [m]

opti = asb.Opti()  # Initialize an optimization environment

w = opti.variable(init_guess=np.zeros(N))  # Displacement [m]

th = opti.derivative_of(  # Slope [rad]
    w, with_respect_to=x,
    derivative_init_guess=np.zeros(N),
)

M = opti.derivative_of(  # Moment [N*m]
    th * EI, with_respect_to=x,
    derivative_init_guess=np.zeros(N),
)

V = opti.derivative_of(  # Shear force [N]
    M, with_respect_to=x,
    derivative_init_guess=np.zeros(N),
)

opti.constrain_derivative(  # Shear integration
    variable=V, with_respect_to=x,
    derivative=q,
)

opti.subject_to([  # Boundary conditions
    w[0] == 0,
    th[0] == 0,
    M[-1] == 0,
    V[-1] == 0,
])

sol = opti.solve()

print(sol(w[-1]))  # Prints the tip deflection; should be 1.62 m.
```

Listing 5: AeroSandbox code for a static structural analysis of a beam. Written in Python.

C.2 Simple Aircraft (SimpleAC)

The Simple Aircraft (SimpleAC) problem described in Section 4.1 can be solved using the following AeroSandbox code, reproduced from Sharpe [1]:

```python
import aerosandbox as asb
import aerosandbox.numpy as np

### Constants
g = 9.81  # gravitational acceleration, m/s^2
mu = 1.775e-5  # viscosity of air, kg/m/s
rho = 1.23  # density of air, kg/m^3
rho_f = 817  # density of fuel, kg/m^3
C_Lmax = 1.6  # stall CL
e = 0.92  # Oswald's efficiency factor
k = 1.17  # form factor
N_ult = 3.3  # ultimate load factor
S_wetratio = 2.075  # wetted area ratio
tau = 0.12  # airfoil thickness to chord ratio
W_W_coeff1 = 2e-5  # first of two assumed constants used in the wing weight model
W_W_coeff2 = 60  # second of two assumed constants used in the wing weight model
Range = 1000e3  # aircraft range, m
TSFC = 0.6 / 3600  # thrust specific fuel consumption, 1/sec
V_min = 25  # takeoff speed, m/s
W_0 = 6250  # aircraft weight excluding wing, N

opti = asb.Opti()  # Initialize an optimization environment

### Free variables
AR = opti.variable(init_guess=10, log_transform=True)  # aspect ratio
S = opti.variable(init_guess=10, log_transform=True)  # total wing area, m^2
V = opti.variable(init_guess=100, log_transform=True)  # cruise speed, m/s
W = opti.variable(init_guess=10000, log_transform=True)  # total aircraft weight, N
C_L = opti.variable(init_guess=1, log_transform=True)  # lift coefficient
W_f = opti.variable(init_guess=3000, log_transform=True)  # fuel weight, N
V_f_fuse = opti.variable(init_guess=1, log_transform=True)  # fuel volume in the fuselage,
↪  m^3

### Wing weight model
W_w_surf = W_W_coeff2 * S
W_w_strc = W_W_coeff1 / tau * N_ult * AR ** 1.5 * np.sqrt(
    (W_0 + V_f_fuse * g * rho_f) * W * S
)
```

```python
W_w = W_w_surf + W_w_strc

### Weight closure constraint
opti.subject_to(W >= W_0 + W_w + W_f)

### Lift force closure constraint
opti.subject_to([
    W_0 + W_w + 0.5 * W_f <= 0.5 * rho * S * C_L * V ** 2,
    W <= 0.5 * rho * S * C_Lmax * V_min ** 2,
])

### Flight duration
T_flight = Range / V

### Drag model
Re = (rho / mu) * V * (S / AR) ** 0.5
C_f = 0.074 / Re ** 0.2

CDA0 = V_f_fuse / 10

C_D_fuse = CDA0 / S
C_D_wpar = k * C_f * S_wetratio
C_D_ind = C_L ** 2 / (np.pi * AR * e)
C_D = C_D_fuse + C_D_wpar + C_D_ind
D = 0.5 * rho * S * C_D * V ** 2

opti.subject_to(W_f >= TSFC * T_flight * D)  # Fuel weight closure constraint

### Fuel volume model
V_f = W_f / g / rho_f
V_f_wing = 0.03 * S ** 1.5 / AR ** 0.5 * tau
V_f_avail = V_f_wing + V_f_fuse
opti.subject_to(V_f_avail >= V_f)  # Fuel volume closure constraint

opti.minimize(W_f)  # Minimize fuel weight

sol = opti.solve()  # Solve the optimization problem
```

C.3 Efficient Computational Implementation of an Arbitrary-Order Noise Estimator

While the given equation for an arbitrary-order data-driven noise estimator (Equation 8.4) is mathematically correct, it is nontrivial to computationally implement. This is because an implementation of the math as-written involves combinatorial coefficients that grow exponentially with the order of the estimator, quickly exceeding standard double-precision floating-point overflow. A naive implementation also has a runtime complexity that scales linearly with both with the number of samples in the dataset and the order of the estimator, making it computationally unattractive for large datasets or high-order estimators.

An efficient computational implementation of the arbitrary-order noise estimator is given in Listing 6. This example is based on syntax for NumPy 1.24.3 and SciPy 1.11.1 within Python 3. This implementation uses the log-gamma function, which is much faster for high-order estimators and delays numerical overflow until much higher orders. The implementation also uses a convolutional kernel to vectorize the summation, enabling further speedups.

One possible way to verify the correctness of the code in Listing 6 is to generate a synthetic signal (e.g., a sampled sine wave), add some normally-distributed, uncorrelated, homoscedastic noise to it, and then use this code to reconstruct the standard deviation of this noise from the dirty signal.

```python
import numpy as np
from scipy.special import gammaln

ln_factorial = lambda x: gammaln(x + 1)  # Shorthand for the function f(x) = ln(x!)

def estimate_noise_standard_deviation(
        data: np.ndarray,
        estimator_order: int = 10
    ) -> float:
    """
    Estimates the standard deviation of noise in a time-series dataset, where the dataset
    is the additive summation of some meaningful signal and some noise. Assumes that this
    noise is both stationary and white (in other words, i.i.d. across samples).

    Args:

        data: A 1D NumPy array of time-series data, assumed to consist of signal + noise.

        estimator_order: The order of the estimator to use. A positive integer. Higher
            values are better at spectrally-separating signal from noise, especially
            when the signal has relatively high-frequency components compared to the
            sample rate.

    Returns: The estimated standard deviation of the noise in the data.
    """
    # For speed, cache ln(x!) for x ∈ [0, estimator_order]
    ln_f = ln_factorial(np.arange(estimator_order + 1))

    # Create a convolutional kernel to vectorize the summation
    coefficients = np.exp(
        2 * ln_f[estimator_order] - ln_f - ln_f[::-1] - 0.5 * ln_factorial(2 *
        ↪  estimator_order)
    ) * (-1) ** np.arange(estimator_order + 1)

    # Remove any bias introduced by floating-point error
    coefficients -= np.mean(coefficients)

    # Convolve the data with the kernel
    sample_stdev = np.convolve(data, coefficients[::-1], 'valid')

    # Return the standard deviation of the result
    return np.mean(sample_stdev ** 2) ** 0.5
```

Listing 6: Example efficient implementation of the arbitrary-order noise estimator using NumPy/SciPy in Python 3.

Bibliography

[1] P. D. Sharpe, "AeroSandbox: A Differentiable Framework for Aircraft Design Optimization," Master's thesis, Massachusetts Institute of Technology, 2021.

[2] A. H. Gebremedhin, F. Manne, and A. Pothen, "What Color Is Your Jacobian? Graph Coloring for Computing Derivatives," *SIAM Review*, vol. 47, pp. 629–705, Jan. 2005. `http://epubs.siam.org/doi/10.1137/S0036144504444711`.

[3] M. Drela, "Development of the D8 Transport Configuration," in *29th AIAA Applied Aerodynamics Conference*, (Honolulu, Hawaii), American Institute of Aeronautics and Astronautics, June 2011. `https://arc.aiaa.org/doi/10.2514/6.2011-3970`.

[4] M. Drela, "Simultaneous Optimization of the Airframe, Powerplant, and Operation of Transport Aircraft," in *RAeS Aircraft Structural Design Conference*, Oct. 2010.

[5] SMG Consulting, "AAM reality index," 2023. `https://aamrealityindex.com/aam-reality-index`.

[6] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[7] E. Burnell, N. B. Damen, and W. Hoburg, "GPkit: A human-centered approach to convex optimization in engineering design," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020.

[8] N. Burnell, "GPkit documentation: "Simple Beam" example," 2020. GitHub repository. `https://gpkit.readthedocs.io/en/latest/examples.html#simple-beam`.

[9] J. S. Gray, J. T. Hwang, J. R. R. A. Martins, K. T. Moore, and B. A. Naylor, "OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization," *Structural and Multidisciplinary Optimization*, vol. 59, pp. 1075–1104, Apr. 2019. `http://link.springer.com/10.1007/s00158-019-02211-z`.

[10] O. D. Team, "OpenMDAO documentation: Optimizing the thickness distribution of a cantilever beam using the adjoint method," 2022. `https://openmdao.org/newdocs/versions/latest/examples/beam_optimization_example.html`.

[11] M. T. Vernacchia, *Development of Low-Thrust Solid Rocket Motors for Small, Fast Aircraft Propulsion*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, Sept. 2020. `https://dspace.mit.edu/handle/1721.1/127069`.

[12] J. C. Gaubatz and R. J. Hansman, "Design and Development of Stability and Control Systems for Small, Deployable Aircraft," Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, Jan. 2024.

[13] P. Sharpe, A. J. Dewald, and J. Hansman, "An Optimization Approach to Mapping the Feasible Mission Space of a High-Altitude Long-Endurance Solar Aircraft," in *AIAA AVIATION 2021 FORUM*, (VIRTUAL EVENT), American Institute of Aeronautics and Astronautics, Aug. 2021. `https://arc.aiaa.org/doi/10.2514/6.2021-2451`.

[14] Electra.aero, "Electra flies solar electric hybrid research aircraft," September 2022.

[15] B. E. Tinling and W. R. Kolk, "The Effects of Mach Number and Reynolds Number on the Aerodynamic Characteristics of Several 12-Percent-Thick Wings Having 35 Degrees of Sweepback and Various Amounts of Camber," NACA Research Memorandum RM A50K27, Ames Aeronautical Laboratory, Moffett Field, California, Feb. 1951.

[16] M. Drela, "TASOPT: Transport Aircraft System Optimization. Technical Description." `http://web.mit.edu/drela/Public/web/tasopt/TASOPT_doc.pdf`, Mar. 2010.

[17] B. M. Kulfan, "Universal Parametric Geometry Representation Method," *Journal of Aircraft*, vol. 45, pp. 142–158, Jan. 2008. `https://arc.aiaa.org/doi/10.2514/1.29958`.

[18] B. Kulfan, "Modification of CST Airfoil Representation Methodology." `https://www.researchgate.net/publication/343615711_Modification_of_CST_Airfoil_Representation_Methodology`, Aug. 2020.

[19] D. A. Masters, N. J. Taylor, T. C. S. Rendall, C. B. Allen, and D. J. Poole, "Geometric Comparison of Aerofoil Shape Parameterization Methods," *AIAA Journal*, vol. 55, pp. 1575–1589, May 2017. `https://arc.aiaa.org/doi/10.2514/1.J054943`.

[20] C. C. Critzos, H. H. Heyson, and R. W. Boswinkle, "Aerodynamic Characteristics of NACA 0012 Airfoil Section at Angles of Attack from 0 to 180," Technical Note 3361, National Advisory Committee for Aeronautics, Washington, Jan. 1955. `https://apps.dtic.mil/sti/pdfs/ADA377080.pdf`.

[21] M. Drela, "Athena Vortex Lattice (AVL)." Massachusetts Institute of Technology, Sept. 2004. `https://web.mit.edu/drela/Public/web/avl/`.

[22] X. Du and P. He, "Rapid Airfoil Design Optimization via Neural Network-based Parameterization and Surrogate Modeling," *Aerospace Science and Technology*, 2021.

[23] M. A. Bouhlel, S. He, and J. R. R. A. Martins, "Scalable gradient–enhanced artificial neural networks for airfoil shape design in the subsonic and transonic regimes," *Structural and Multidisciplinary Optimization*, vol. 61, pp. 1363–1376, Apr. 2020. `http://link.springer.com/10.1007/s00158-020-02488-5`.

[24] W. Peng, Y. Zhang, E. Laurendeau, and M. C. Desmarais, "Learning aerodynamics with neural network," *Scientific Reports*, vol. 12, p. 6779, Apr. 2022. `https://www.nature.com/articles/s41598-022-10737-4`.

[25] United States Environmental Protection Agency, "The EPA Automotive Trends Report 2023," 2023.

[26] M. Drela, "Making an extraordinary machine better: The D8 aircraft concept." TEDxNewEngland, Nov. 2012.

[27] M. Drela, "Design Drivers of Energy-Efficient Transport Aircraft," *SAE International Journal of Aerospace*, vol. 4, pp. 602–618, Oct. 2011. `https://www.sae.org/content/2011-01-2495/`.

[28] J. R. R. A. Martins and A. B. Lambe, "Multidisciplinary Design Optimization: A Survey of Architectures," *AIAA Journal*, vol. 51, pp. 2049–2075, Sept. 2013. `https://arc.aiaa.org/doi/10.2514/1.J051895`.

[29] H. Ashley, "On Making Things the Best-Aeronautical Uses of Optimization," *Journal of Aircraft*, vol. 19, pp. 5–28, Jan. 1982. `https://arc.aiaa.org/doi/10.2514/3.57350`.

[30] G. Vanderplaats, "Automated optimization techniques for aircraft synthesis," in *Aircraft Systems and Technology Meeting*, (Dallas,TX,U.S.A.), American Institute of Aeronautics and Astronautics, Sept. 1976. `https://arc.aiaa.org/doi/10.2514/6.1976-909`.

[31] T. Haftka, "Multidisciplinary Aerospace Design Optimization: Survey of Recent Developments," *Springer, Structural Optimization*, Aug. 1997.

[32] I. Kroo, "Multidisciplinary Optimization Applications in Preliminary Design - Status and Directions," in *38th Structures, Structural Dynamics, and Materials Conference*, (Kissimmee, Florida), American Institute of Aeronautics and Astronautics, Apr. 1997. `https://arc.aiaa.org/doi/10.2514/6.1997-1408`.

[33] M. Drela, *Pros & Cons of Airfoil Optimization*, pp. 363–381. World Scientific, Nov. 1998. `http://www.worldscientific.com/doi/abs/10.1142/9789812815774_0019`.

[34] J. Agte, O. De Weck, J. Sobieszczanski-Sobieski, P. Arendsen, A. Morris, and M. Spieck, "MDO: Assessment and direction for advancement—an opinion of one international group," *Structural and Multidisciplinary Optimization*, vol. 40, pp. 17–33, Jan. 2010. `http://link.springer.com/10.1007/s00158-009-0381-5`.

[35] A. Gazaix, F. Gallard, V. Gachelin, T. Druot, S. Grihon, V. Ambert, D. Guénot, R. Lafage, C. Vanaret, B. Pauwels, N. Bartoli, T. Lefebvre, P. Sarouille, N. Desfachelles, J. Brézillon, M. Hamadi, and S. Gurol, "Towards the Industrialization of New MDO Methodologies and Tools for Aircraft Design," in *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization*

Conference, (Denver, Colorado), American Institute of Aeronautics and Astronautics, June 2017. https://arc.aiaa.org/doi/10.2514/6.2017-3149.

[36] A. Salas and J. Townsend, "Framework Requirements for MDO Application Development," in *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, (St. Louis,MO,U.S.A.), American Institute of Aeronautics and Astronautics, Sept. 1998. https://arc.aiaa.org/doi/10.2514/6.1998-4740.

[37] Y. Ma, S. Gowda, R. Anantharaman, C. Laughman, V. Shah, and C. Rackauckas, "ModelingToolkit: A Composable Graph Transformation System for Equation-Based Modeling," *arXiv:2103.05244 (cs)*, 2021.

[38] P. Sharpe, "AeroSandbox," 2019. GitHub repository. https://github.com/peterdsharpe/AeroSandbox.

[39] N. Radovcich and D. Layton, "The F-22 structural/aeroelastic design process with MDO examples," in *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, (St. Louis,MO,U.S.A.), American Institute of Aeronautics and Astronautics, Sept. 1998. https://arc.aiaa.org/doi/10.2514/6.1998-4732.

[40] J. H. McMasters and R. M. Cummings, "Airplane Design - Past, Present, and Future," *Journal of Aircraft*, vol. 39, pp. 10–17, Jan. 2002. https://arc.aiaa.org/doi/10.2514/2.2919.

[41] G. Belie, "Non-Technical Barriers to Multidisciplinary Optimization in the Aerospace Industry," in *9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, (Atlanta, Georgia), American Institute of Aeronautics and Astronautics, Sept. 2002. http://arc.aiaa.org/doi/10.2514/6.2002-5439.

[42] A. Yildirim, J. S. Gray, C. A. Mader, and J. R. Martins, "Performance Analysis of Optimized STARC-ABL Designs Across the Entire Mission Profile," in *AIAA Scitech 2021 Forum*, (VIRTUAL EVENT), American Institute of Aeronautics and Astronautics, Jan. 2021. https://arc.aiaa.org/doi/10.2514/6.2021-0891.

[43] N. Bons, Joaquim R. R. A. Martins, Charles A. Mader, M. McMullen, and M. Suen, "High-fidelity Aerostructural Optimization Studies of the Aerion AS2 Supersonic Business Jet," in *AIAA Aviation 2020 Forum*, 2020.

[44] N. E. Antoine and I. M. Kroo, "Framework for Aircraft Conceptual Design and Environmental Performance Studies," *AIAA Journal*, vol. 43, pp. 2100–2109, Oct. 2005. https://arc.aiaa.org/doi/10.2514/1.13017.

[45] E. Torenbeek, *Advanced Aircraft Design: Conceptual Design, Analysis, and Optimization of Subsonic Civil Airplanes*. Aerospace Series, Chichester: Wiley, 2013.

[46] W. Hoburg and P. Abbeel, "Geometric Programming for Aircraft Design Optimization," *AIAA Journal*, vol. 52, pp. 2414–2426, Nov. 2014. https://arc.aiaa.org/doi/10.2514/1.J052732.

[47] I. Van Gent, B. Aigner, B. Beijer, J. Jepsen, and G. La Rocca, "Knowledge architecture supporting the next generation of MDO in the AGILE paradigm," *Progress in Aerospace Sciences*, vol. 119, p. 100642, Nov. 2020. `https://linkinghub.elsevier.com/retrieve/pii/S0376042120300543`.

[48] B. Öztürk and A. Saab, "Optimal Aircraft Design Decisions Under Uncertainty Using Robust Signomial Programming," *AIAA Journal*, vol. 59, pp. 1773–1785, May 2021. `https://arc.aiaa.org/doi/10.2514/1.J058724`.

[49] S. Gudmundsson, "General Aviation Aircraft Design," in *General Aviation Aircraft Design*, p. xiii, Elsevier, 2014. `https://linkinghub.elsevier.com/retrieve/pii/B9780123973085050017`.

[50] L. M. Nicolai and G. Carichner, *Fundamentals of Aircraft and Airship Design. Volume 1: Aircraft Design*, vol. 1 of *AIAA Educational Series*. Reston, VA: American Institute of Aeronautics and Astronautics, 2010.

[51] J. M. Walton, "CD BootCamp: A Unique Approach to Conceptual Design New-Hire Training at Lockheed Martin Skunk Works," in *AIAA Scitech 2020 Forum*, (Orlando, FL), American Institute of Aeronautics and Astronautics, Jan. 2020. `https://arc.aiaa.org/doi/10.2514/6.2020-1774`.

[52] A. M. Stoll, E. V. Stilson, J. Bevirt, and P. P. Pei, "Conceptual Design of the Joby S2 Electric VTOL PAV," in *14th AIAA Aviation Technology, Integration, and Operations Conference*, (Atlanta, GA), American Institute of Aeronautics and Astronautics, June 2014. `https://arc.aiaa.org/doi/10.2514/6.2014-2407`.

[53] T. S. Tao, *Design and Development of a High-Altitude, In-Flight-Deployable Micro-UAV*. S.M., Massachusetts Institute of Technology, 2012. `https://dspace.mit.edu/handle/1721.1/76168`.

[54] S. Shahpar, "Challenges to overcome for routine usage of automatic optimisation in the propulsion industry," *The Aeronautical Journal*, vol. 115, no. 1172, p. 615–625, 2011.

[55] K. Pitta, "Using NAS-developed tools to quiet the boom of supersonic flight." `https://www.nas.nasa.gov/pubs/stories/2021/feature_X-59.html`, March 2021. Accessed 10-25-2023.

[56] M. Nemec and M. Aftosmis, "Minimizing sonic boom through simulation-based design: The X-59 airplane." `https://www.nas.nasa.gov/SC19/demos/demo20.html`, November 2019. Accessed 10-25-2023.

[57] H. Mason, "Digital design, multi-material structures enable a quieter supersonic NASA X-plane." `https://www.compositesworld.com/articles/digital-design-multi-material-structures-enable-a-quieter-supersonic-nasa-x-plane`, May 2022. Accessed 10-25-2023.

[58] S. Choi, J. J. Alonso, I. M. Kroo, and M. Wintzer, "Multifidelity Design Optimization of Low-Boom Supersonic Jets," *Journal of Aircraft*, vol. 45, pp. 106–118, Jan. 2008. https://arc.aiaa.org/doi/10.2514/1.28948.

[59] Z. Lovering, "Vahana configuration trade study – part I." https://acubed.airbus.com/blog/vahana/vahana-configuration-trade-study-part-i/, December 2016. Accessed 11-03-2023.

[60] G. Bower, "Vahana configuration trade study – part II." https://acubed.airbus.com/blog/vahana/vahana-configuration-trade-study-part-ii/, February 2017. Accessed 11-03-2023.

[61] "Vahana trade study." https://github.com/VahanaOpenSource/vahanaTradeStudy, 2017. GitHub repository. Accessed 11-03-2023.

[62] N. Roberts, V. Samuel, and D. Colas, "FBHALE." https://github.com/facebookarchive/FBHALE/wiki, 2018. GitHub repository.

[63] B. Ozturk, W. Hoburg, N. Burnell, and C. Karcher, "Jungle Hawk Owl." https://github.com/convexengineering/jho, 2016. GitHub repository.

[64] S. Wakayama, "Blended-wing-body optimization setup," in *8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, (Long Beach, CA), 2000.

[65] R. Liebeck, M. Page, and B. Rawdon, "Blended-Wing-Body Subsonic Commercial Transport," in *36th AIAA Aerospace Sciences Meeting and Exhibit*, (Reno,NV,U.S.A.), American Institute of Aeronautics and Astronautics, Jan. 1998. https://arc.aiaa.org/doi/10.2514/6.1998-438.

[66] R. Liebeck, "Design of blended wing body subsonic transport," *Journal of Aircraft*, pp. 10–25, January-February 2004.

[67] P. D. Ciampa and B. Nagel, "TOWARDS THE 3RD GENERATION MDO COLLABORATIVE ENVIRONMENT," in *30th Congress of the International Council of the Aeronautical Sciences*, (Daejeon, Korea), Sept. 2016.

[68] P. Piperni, A. DeBlois, and R. Henderson, "Development of a Multilevel Multidisciplinary-Optimization Capability for an Industrial Environment," *AIAA Journal*, vol. 51, pp. 2335–2352, Oct. 2013. https://arc.aiaa.org/doi/10.2514/1.J052180.

[69] R. Heldenfels, "Automating the design process - Progress, problems, prospects, potential," in *14th Structures, Structural Dynamics, and Materials Conference*, (Williamsburg,VA,U.S.A.), American Institute of Aeronautics and Astronautics, Mar. 1973. https://arc.aiaa.org/doi/10.2514/6.1973-410.

[70] R. R. Heldenfels, "Automation of the Aircraft Design Process," in *International Council of the Aeronautical Sciences Congress*, (Haifa, Israel), NASA, Aug. 1974. https://www.icas.org/ICAS_ARCHIVE/ICAS1974/Page%20617%20Heldenfels.pdf.

[71] J. R. R. A. Martins and A. Ning, *Engineering Design Optimization*. Cambridge University Press, 1 ed., Nov. 2021. https://www.cambridge.org/core/product/identifier/9781108980647/type/book.

[72] M. C. Yang, "Observations on concept generation and sketching in engineering design," *Research in Engineering Design*, vol. 20, pp. 1–11, Mar. 2009. http://link.springer.com/10.1007/s00163-008-0055-0.

[73] E. Torenbeek, *Synthesis of Subsonic Aircraft Design*. Delft: Delft University Press, 1976.

[74] J. Roskam, *Airplane Design*. Ottawa, Kan.: Roskam Aviation and Engineering Corp., 1989.

[75] S. A. Niederer, M. S. Sacks, M. Girolami, and K. Willcox, "Scaling digital twins from the artisanal to the industrial," *Nature Computational Science*, vol. 1, pp. 313–320, May 2021. https://doi.org/10.1038/s43588-021-00072-5.

[76] V. Singh and K. Willcox, "Engineering Design with Digital Thread," *AIAA Journal*, vol. 56, Nov. 2018.

[77] J. R. Cruz, "Weight Analysis of the Daedalus Human Powered Aircraft," 1989.

[78] G. K. W. Kenway and J. R. R. A. Martins, "Multipoint High-Fidelity Aerostructural Optimization of a Transport Aircraft Configuration," *Journal of Aircraft*, vol. 51, pp. 144–160, Jan. 2014. https://arc.aiaa.org/doi/10.2514/1.C032150.

[79] P. He, C. A. Mader, J. R. Martins, and K. J. Maki, "An aerodynamic design optimization framework using a discrete adjoint approach with OpenFOAM," *Computers & Fluids*, vol. 168, pp. 285–303, May 2018. https://linkinghub.elsevier.com/retrieve/pii/S0045793018302020.

[80] J. Alonso, P. LeGresley, E. Van Der Weide, J. R. R. A. Martins, and J. Reuther, "pyMDO: A Framework for High-Fidelity Multi-Disciplinary Optimization," in *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, (Albany, New York), American Institute of Aeronautics and Astronautics, Aug. 2004. https://arc.aiaa.org/doi/10.2514/6.2004-4480.

[81] J. R. Martins, J. J. Alonso, and J. J. Reuther, "A Coupled-Adjoint Sensitivity Analysis Method for High-Fidelity Aero-Structural Design," *Optimization and Engineering*, vol. 6, pp. 33–62, Mar. 2005. http://link.springer.com/10.1023/B:OPTE.0000048536.47956.62.

[82] Z. Lyu and Z. Xu, "Benchmarking Optimization Algorithms for Wing Aerodynamic Design Optimization," p. 18, 2014.

[83] P. Kirschen and W. Hoburg, "The power of log transformation: A comparison of geometric and signomial programming with general nonlinear programming techniques for aircraft design optimization," 2018.

[84] E. J. Adler, A. C. Gray, and J. R. Martins, "To CFD or not to CFD? Comparing RANS and viscous panel methods for airfoil shape optimization," in *Congress of the International Council of the Aeronautical Sciences (ICAS 2022)*, (Stockholm, Sweden), Sept. 2022.

[85] B. J. Brelje, *Multidisciplinary Design Optimization of Electric Aircraft Considering Systems Modeling and Packaging*. PhD thesis, University of Michigan, Ann Arbor, MI, 2021. `https://deepblue.lib.umich.edu/handle/2027.42/169658`.

[86] S. S. Chauhan and J. R. R. A. Martins, "Low-fidelity aerostructural optimization of aircraft wings with a simplified wingbox model using OpenAeroStruct," in *Proceedings of the 6th International Conference on Engineering Optimization, EngOpt 2018*, (Lisbon, Portugal), pp. 418–431, Springer, 2018.

[87] M. Grant, S. Boyd, and Y. Ye, "Disciplined Convex Programming," in *Global Optimization* (L. Liberti and N. Maculan, eds.), vol. 84, pp. 155–210, Boston: Kluwer Academic Publishers, 2006. `http://link.springer.com/10.1007/0-387-30528-9_7`.

[88] A. Agrawal, S. Diamond, and S. Boyd, "Disciplined geometric programming," *Optimization Letters*, vol. 13, pp. 961–976, July 2019. `http://link.springer.com/10.1007/s11590-019-01422-z`.

[89] S. Boyd, S.-J. Kim, L. Vandenberghe, and A. Hassibi, "A tutorial on geometric programming," *Optimization and Engineering*, vol. 8, pp. 67–127, May 2007. `http://link.springer.com/10.1007/s11081-007-9001-7`.

[90] S. P. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, UK ; New York: Cambridge University Press, 2004.

[91] R. Haimes and M. Drela, "On The Construction of Aircraft Conceptual Geometry for High-Fidelity Analysis and Design," in *50th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, (Nashville, Tennessee), American Institute of Aeronautics and Astronautics, Jan. 2012. `https://arc.aiaa.org/doi/10.2514/6.2012-683`.

[92] D. S. Lazzara, R. Haimes, and K. Willcox, "Multifidelity geometry and analysis in aircraft conceptual design," in *19th AIAA Computational Fluid Dynamics Conference*, (San Antonio, TX), American Institute of Aeronautics and Astronautics, June 2009.

[93] R. Haimes and J. Dannenhoffer, *The Engineering Sketch Pad: A Solid-Modeling, Feature-Based, Web-Enabled System for Building Parametric Geometry*.

[94] R. A. McDonald and J. R. Gloudemans, "Open vehicle sketch pad: An open source parametric geometry and analysis tool for conceptual aircraft design," in *AIAA SCITECH 2022 Forum*, 2022. `https://arc.aiaa.org/doi/abs/10.2514/6.2022-0004`.

[95] M. Drela, "XFOIL: An Analysis and Design System for Low Reynolds Number Airfoils," in *Low Reynolds Number Aerodynamics* (C. A. Brebbia, S. A. Orszag, and T. J. Mueller, eds.), vol. 54, (Indiana, USA), pp. 1–12, Springer, 1989. `http://link.springer.com/10.1007/978-3-642-84010-4_1`.

[96] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi – A software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, 2019.

[97] C. Rackauckas, "Generalizing Automatic Differentiation to Automatic Sparsity, Uncertainty, Stability, and Parallelism." `https://www.stochasticlifestyle.com/generalizing-automatic-differentiation-to-automatic-sparsity-uncertainty-stability-and-parallelism/`, Mar. 2021.

[98] A. Lavin, D. Krakauer, H. Zenil, J. Gottschlich, T. Mattson, J. Brehmer, A. Anandkumar, S. Choudry, K. Rocki, A. G. Baydin, C. Prunkl, B. Paige, O. Isayev, E. Peterson, P. L. McMahon, J. Macke, K. Cranmer, J. Zhang, H. Wainwright, A. Hanuka, M. Veloso, S. Assefa, S. Zheng, and A. Pfeffer, "Simulation Intelligence: Towards a New Generation of Scientific Methods." `http://arxiv.org/abs/2112.03235`, Nov. 2022.

[99] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey." `http://arxiv.org/abs/1502.05767`, Feb. 2018.

[100] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Philadelphia: SIAM, 2. ed ed., 2008.

[101] A. Griewank, "On Automatic Differentiation," in *Mathematical Programming: Recent Developments and Applications*, Argonne, Illinois: Argonne National Laboratory, 1988. `https://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR89003.pdf`.

[102] D. Maclaurin, D. Duvenaud, and R. P. Adams, "Autograd: Effortless Gradients in Numpy," in *AutoML Workshop*, p. 3, 2015.

[103] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018.

[104] A. B. Lambe and J. R. R. A. Martins, "Extensions to the design structure matrix for the description of multidisciplinary design, analysis, and optimization processes," *Structural and Multidisciplinary Optimization*, vol. 46, pp. 273–284, Aug. 2012. `http://link.springer.com/10.1007/s00158-012-0763-y`.

[105] A. H. Gebremedhin, A. Tarafdar, A. Pothen, and A. Walther, "Efficient Computation of Sparse Hessians Using Coloring and Automatic Differentiation," *INFORMS Journal on Computing*, vol. 21, pp. 209–223, May 2009. `https://pubsonline.informs.org/doi/10.1287/ijoc.1080.0286`.

[106] S. Gowda, V. Churavy, A. Edelman, Y. Ma, and C. Rackauckas, "Sparsity Programming: Automated Sparsity-Aware Optimizations in Differentiable Programming." `https://openreview.net/pdf?id=rJlPdcY38B`, 2019.

[107] J. E. Marte and D. W. Kurtz, "A Review of Aerodynamic Noise From Propellers, Rofors, and Lift Fans," Technical Report 32-1462, NASA Jet Propulsion Laboratory, 1970.

[108] S. Sudhakar, S. Karaman, and V. Sze, "Balancing Actuation and Computing Energy in Motion Planning," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, (Paris, France), pp. 4259–4265, IEEE, May 2020. `https://ieeexplore.ieee.org/document/9197164/`.

[109] P. Mechanics, “MIT developing Mach 0.8 rocket drone for the Air Force.” https://www.popularmechanics.com/military/aviation/a13938789/mit-developing-mach-08-rocket-drone-for-the-air-force/, 2017.

[110] K. J. Mathesius and R. J. Hansman, “Manufacturing methods for a solid rocket motor propelling a small, fast flight vehicle,” Master’s thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2019.

[111] J. Nocedal and S. J. Wright, *Numerical Optimization*. Springer Series in Operations Research, New York: Springer, 2nd ed ed., 2006.

[112] Y. Hu, “Taichi: An Open-Source Computer Graphics Library.” http://arxiv.org/abs/1804.09293, Apr. 2018.

[113] C. Rackauckas, “Engineering Trade-Offs in Automatic Differentiation: From TensorFlow and PyTorch to Jax and Julia.” https://www.stochasticlifestyle.com/engineering-trade-offs-in-automatic-differentiation-from-tensorflow-and-pytorch-to-jax-and-julia/, Dec. 2021.

[114] L. Hascoet and V. Pascual, “The Tapenade automatic differentiation tool: principles, model, and specification,” *ACM Transactions on Mathematical Software*, vol. 39, no. 3, 2013.

[115] D. Maclaurin, *Modeling, Inference, and Optimization With Composable Differentiable Procedures*. PhD thesis, Harvard University, Cambridge, Massachusetts, Apr. 2016. https://dash.harvard.edu/bitstream/handle/1/33493599/MACLAURIN-DISSERTATION-2016.pdf.

[116] R. Frostig, M. J. Johnson, and C. Leary, “Compiling machine learning programs via high-level tracing,” in *SYSML ’18*, (Stanford, CA, USA), Feb. 2018. https://mlsys.org/Conferences/2019/doc/2018/146.pdf.

[117] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” 2019. https://arxiv.org/abs/1912.01703.

[118] G. K. Kenway, C. A. Mader, P. He, and J. R. Martins, “Effective adjoint approaches for computational fluid dynamics,” *Progress in Aerospace Sciences*, vol. 110, Oct. 2019. https://linkinghub.elsevier.com/retrieve/pii/S0376042119300120.

[119] M. Innes, “Don’t Unroll Adjoint: Differentiating SSA-Form Programs,” *arXiv:1810.07951 [cs]*, Mar. 2019. http://arxiv.org/abs/1810.07951.

[120] T. MacDonald, M. Clarke, E. M. Botero, J. M. Vegh, and J. J. Alonso, *SUAVE: An Open-Source Environment Enabling Multi-Fidelity Vehicle Optimization*.

[121] ISAE-SUPAERO and ONERA, “FAST-GA: Future aircraft sizing tool - overall aircraft design - general aviation,” 2021. GitHub repository. Accessed 11-13-2023.

[122] L. A. Mccullers, "Aircraft configuration optimization including optimized flight profiles," in *Recent Experiences in Multidisciplinary Analysis and Optimization, Part 1*, (Hampton, VA, United States), NASA Langley Research Center, January 1984.

[123] C. Karcher and R. Haimes, "A method of sequential log-convex programming for engineering design," *Optimization and Engineering*, vol. 24, pp. 1719–1745, Sept. 2023. `https://doi.org/10.1007/s11081-022-09750-3`.

[124] R. Fourer, D. M. Gay, and B. W. Kernighan, "AMPL: A mathematical programing language," in *Algorithms and Model Formulations in Mathematical Programming* (S. W. Wallace, ed.), (Berlin, Heidelberg), pp. 150–151, Springer Berlin Heidelberg, 1989.

[125] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. Van Kerkwijk, M. Brett, A. Haldane, J. F. Del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020. `https://www.nature.com/articles/s41586-020-2649-2`.

[126] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, pp. 25–57, Mar. 2006. `http://link.springer.com/10.1007/s10107-004-0559-y`.

[127] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi: A software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, pp. 1–36, Mar. 2019. `http://link.springer.com/10.1007/s12532-018-0139-4`.

[128] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2006.

[129] H. H. Rosenbrock, "An Automatic Method for Finding the Greatest or Least Value of a Function," *The Computer Journal*, vol. 3, pp. 175–184, January 1960.

[130] S. Kok and C. Sandrock, "Locating and characterizing the stationary points of the extended rosenbrock function," *Evolutionary Computation*, vol. 17, pp. 437–453, September 2009.

[131] M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. Cambridge, Massachusetts: The MIT Press, 2019.

[132] K.-L. Lai and J. Crassidis, "Extensions of the first and second complex-step derivative approximations," *Journal of Computational and Applied Mathematics*, vol. 219, pp. 276–293, Sept. 2008. `https://linkinghub.elsevier.com/retrieve/pii/S0377042707004086`.

[133] J. R. R. A. Martins, P. Sturdza, and J. J. Alonso, "The complex-step derivative approximation," *ACM Transactions on Mathematical Software*, vol. 29, pp. 245–262, Sept. 2003. `https://dl.acm.org/doi/10.1145/838250.838251`.

[134] B. Ozturk, *Conceptual Engineering Design and Optimization Methodologies Using Geometric Programming*. PhD thesis, Massachusetts Institute of Technology, 2018.

[135] M. Vernacchia, "Modeling compressible aerodynamics of aircraft in geometric programs," Unpublished manuscript, part of the "convexengineering/gplibrary" GitHub repository. `https://github.com/convexengineering/gplibrary/blob/compress-aero/gpkitmodels/SP/aircraft/compressible_aero/docs/compressible_aero.pdf`.

[136] M. S. Andersen, J. Dahl, and L. Vandenberghe, "Cvxopt: Python software for convex optimization," 2023. Version 1.3.2.

[137] MOSEK ApS, *MOSEK Optimization Suite*. MOSEK ApS, 2024.

[138] C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, A. Ramadhan, and A. Edelman, "Universal Differential Equations for Scientific Machine Learning." `http://arxiv.org/abs/2001.04385`, Nov. 2021.

[139] C. Rackauckas, "Direct Automatic Differentiation of (Differential Equation) Solvers vs Analytical Adjoints: Which is Better?." `http://www.stochasticlifestyle.com/direct-automatic-differentiation-of-solvers-vs-analytical-adjoints-which-is-better/`, Oct. 2022.

[140] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations," in *Advances in Neural Information Processing Systems*, pp. 6571–6583, 2018.

[141] A. Griewank and A. Walther, "Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation," *ACM Transactions on Mathematical Software*, vol. 26, pp. 19–45, Mar. 2000. `https://dl.acm.org/doi/10.1145/347837.347846`.

[142] R. T. Haftka, "Simultaneous analysis and design," *AIAA Journal*, vol. 23, pp. 1099–1103, July 1985. `https://arc.aiaa.org/doi/10.2514/3.9043`.

[143] B. J. Brelje and J. R. R. A. Martins, "Development of a conceptual design model for aircraft electric propulsion with efficient gradients," in *Proceedings of the AIAA/IEEE Electric Aircraft Technologies Symposium*, (Cincinnati, OH), July 2018.

[144] E. J. Adler and J. R. R. A. Martins, "Efficient aerostructural wing optimization considering mission analysis," *Journal of Aircraft*, December 2022.

[145] C. for Python Data API Standards, *Python Array API Standard*, December 2023. Version 2023.12.

[146] H. Grecco and contributors, "Pint: Operate and manipulate physical quantities in Python," 2024. Version 0.23.

[147] A. Keller and S. Stock, "Unitful.jl," 2024. Version 1.20.0.

[148] M. Drela and H. Youngren, "AVL: Athena vortex lattice." `https://web.mit.edu/drela/Public/web/avl/`, 1989–2023. Accessed 11-09-2023.

[149] T. D. Economon, F. Palacios, S. R. Copeland, T. W. Lukaczyk, and J. J. Alonso, "SU2: An Open-Source Suite for Multiphysics Simulation and Design," *AIAA Journal*, vol. 54, pp. 828–846, Mar. 2016. https://arc.aiaa.org/doi/10.2514/1.J053813.

[150] W. Hoburg and P. Abbeel, "Geometric Programming for Aircraft Design Optimization," *AIAA Journal*, 2014.

[151] B. Ozturk, "Conceptual Engineering Design and Optimization Methodologies using Geometric Programming," 2018.

[152] K. J. Mathesius, *Integrated Design of Solid Rocket Powered Vehicles Including Exhaust Plume Radiant Emission*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 2023. https://dspace.mit.edu/handle/1721.1/151348.

[153] M. Kelly, "An Introduction to Trajectory Optimization: How to Do Your Own Direct Collocation," *SIAM Review*, vol. 59, pp. 849–904, Jan. 2017. https://epubs.siam.org/doi/10.1137/16M1062569.

[154] M. Drela, "Flight Vehicle Aerodynamics," p. 270, 2013.

[155] A. Dewald and R. J. Hansman, "A Multidisciplinary Analysis of a Stratospheric Airborne Climate Observatory System for Key Climate Risk Areas," in *AIAA SCITECH 2023 Forum*, (National Harbor, MD & Online), American Institute of Aeronautics and Astronautics, Jan. 2023. https://arc.aiaa.org/doi/10.2514/6.2023-2603.

[156] B. A. Avery, N. Bain, D. Barea, K. Carlson, A. Dewald, KJ. Hardrict, B. Ö. Kristinsson, D. Lee, J. Liao, T. Long, M. Nasir, C. Oneci, A. Peraire-Bueno, P. Sharpe, M. Xu, and J. Zavala, "16.82 "Dawn" Solar Airplane Critical Design Review," p. 154.

[157] J. A. Dykema, S. Bianconi, C. Mascarenhas, and J. Anderson, "Feasibility study of a total precipitable water IPDA lidar from a solar-powered stratospheric aircraft," *Applied Optics*, vol. 62, pp. 6724–6736, Sept. 2023. https://opg.optica.org/ao/abstract.cfm?URI=ao-62-25-6724.

[158] P. Sharpe, D. Ulker, and M. Drela, "Tailerons for Aeroelastic Stability and Control of Flexible Wings," in *AIAA AVIATION 2023 Forum*, (San Diego, CA and Online), American Institute of Aeronautics and Astronautics, June 2023. https://arc.aiaa.org/doi/10.2514/6.2023-3951.

[159] J. R. Cruz and M. Drela, "Structural Design Conditions for Human Powered Aircraft," in *21st OSTIV Conference*, (Austria), 1989. https://web.mit.edu/drela/Public/web/hpa/hpa_structure.pdf.

[160] J. S. Langford, "The Feasibility of a Human-Powered Flight Between Crete and the Mainland of Greece," tech. rep., 1986.

[161] J. Langford, "The Daedalus project - A summary of lessons learned," in *Aircraft Design and Operations Meeting*, (Seattle,WA,U.S.A.), American Institute of Aeronautics and Astronautics, July 1989. https://arc.aiaa.org/doi/10.2514/6.1989-2048.

[162] Z. Spakovszky, E. M. Greitzer, and I. A. Waitz, "MIT Unified Engineering Course Notes, Thermodynamics and Propulsion: Performance of propellers," 2007.

[163] M. Drela, "QPROP Formulation," tech. rep., Massachusetts Institute of Technology, Cambridge, MA, 2006.

[164] H. Hersbach, B. Bell, P. Berrisford, S. Hirahara, A. Horányi, J. Muñoz-Sabater, J. Nicolas, C. Peubey, R. Radu, D. Schepers, A. Simmons, C. Soci, S. Abdalla, X. Abellan, G. Balsamo, P. Bechtold, G. Biavati, J. Bidlot, M. Bonavita, G. De Chiara, P. Dahlgren, D. Dee, M. Diamantakis, R. Dragani, J. Flemming, R. Forbes, M. Fuentes, A. Geer, L. Haimberger, S. Healy, R. J. Hogan, E. Hólm, M. Janisková, S. Keeley, P. Laloyaux, P. Lopez, C. Lupu, G. Radnoti, P. de Rosnay, I. Rozum, F. Vamborg, S. Villaume, and J.-N. Thépaut, "The era5 global reanalysis," *Quarterly Journal of the Royal Meteorological Society*, vol. 146, no. 730, pp. 1999–2049, 2020.

[165] A. B. Lambe and J. R. R. A. Martins, "Extensions to the Design Structure Matrix for the Description of Multidisciplinary Design, Analysis, and Optimization Processes," tech. rep., 2012.

[166] B. M. Yutko, "Approaches to representing aircraft fuel efficiency performance for the purpose of a commercial aircraft certification standard," Master's thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, 2011.

[167] Air Transportation Action Group, "Waypoint 2050," 2021.

[168] J. Gaubatz, E. Martin, A. Miyamoto, B. Murga, P. Sharpe, M. Travnik, A. Tsay, Z. J. Wang, and R. J. Hansman, "Estimating the Energy Demand of a Hydrogen-Based Long-Haul Air Transportation Network," in *2023 International Conference on Future Energy Solutions (FES)*, (Vaasa, Finland), pp. 1–6, IEEE, June 2023. https://ieeexplore.ieee.org/document/10182543/.

[169] Boeing, "Boeing Cascade climate impact model," June 2024.

[170] G. D. Brewer, *Hydrogen Aircraft Technology*. Boca Raton: CRC Press, 1991.

[171] S. Tiwari, M. J. Pekris, and J. J. Doherty, "A review of liquid hydrogen aircraft and propulsion technologies," *International Journal of Hydrogen Energy*, vol. 57, pp. 1174–1196, 2024. https://www.sciencedirect.com/science/article/pii/S0360319923065631.

[172] Boeing, "777-200LR/-300ER/-Freighter airplane characteristics for airport planning," tech. rep., Boeing, August 2009.

[173] M. Drela, "Apogee hand launch glider plans," August 1999. Accessed: 2024-06-06.

[174] W. F. Phillips and D. O. Snyder, "Modern Adaptation of Prandtl's Classic Lifting-Line Theory," *Journal of Aircraft*, vol. 37, pp. 662–670, July 2000. https://arc.aiaa.org/doi/10.2514/2.2649.

[175] J. T. Reid, *A General Approach to Lifting-Line Theory, Applied to Wings with Sweep*. PhD thesis, Utah State University, Aug. 2020. https://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=8982&context=etd.

[176] J. Weissinger, "The lift distribution of swept-back wings," Tech. Rep. Technical Memorandum 1120, National Advisory Committee for Aeronautics (NACA), Washington, DC, 1947.

[177] R. Fink, "USAF stability and control DATCOM," Tech. Rep. AFWAL-TR-83-3048, McDonnell Douglas Corporation, Douglas Aircraft Division, 1960. for the Flight Controls Division, Air Force Flight Dynamics Laboratory, Wright-Patterson AFB, Ohio. October 1960, revised November 1965, revised April 1978.

[178] M. Drela, "ASWING user guide." `https://web.mit.edu/drela/Public/web/aswing/`, 2012. Accessed: 11-09-2023.

[179] A. Deperrois, "XFLR5: An analysis tool for airfoils, wings and planes operating at low Reynolds numbers." `http://www.xflr5.tech/xflr5.htm`, 2011. Version 6.0.

[180] J. Katz and A. Plotkin, "Low-Speed Aerodynamics, Second Edition," *Journal of Fluids Engineering*, vol. 126, pp. 293–294, Mar. 2004. `https://asmedigitalcollection.asme.org/fluidsengineering/article/126/2/293/458666/LowSpeed-Aerodynamics-Second-Edition`.

[181] M. Drela and M. B. Giles, "Viscous-inviscid analysis of transonic and low Reynolds number airfoils," *AIAA Journal*, vol. 25, pp. 1347–1355, Oct. 1987. `https://arc.aiaa.org/doi/10.2514/3.9789`.

[182] M. Ranneberg, "Viiflow—A New Inverse Viscous-Inviscid Interaction Method," *AIAA Journal*, vol. 57, pp. 2248–2253, June 2019. `https://arc.aiaa.org/doi/10.2514/1.J058268`.

[183] K. J. Fidkowski, "A Coupled Inviscid–Viscous Airfoil Analysis Solver, Revisited," *AIAA Journal*, vol. 60, pp. 2961–2971, May 2022. `https://arc.aiaa.org/doi/10.2514/1.J061341`.

[184] S. Zhang, *Three-Dimensional Integral Boundary Layer Method for Viscous Aerodynamic Analysis*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2022. `https://dspace.mit.edu/handle/1721.1/147502`.

[185] J. D. Anderson, "Fundamentals of Aerodynamics," p. 1131, 2009.

[186] S. C. Smith and I. M. Kroo, "Computation of induced drag for elliptical and crescent-shaped wings," *Journal of Aircraft*, vol. 30, pp. 446–452, July 1993. `https://arc.aiaa.org/doi/10.2514/3.46365`.

[187] R. Jacobs, H. Ran, M. Kirby, and D. Mavris, "Extension of a Modern Lifting-Line Method to Transonic Speeds and Application to Multiple-Lifting-Surface Configurations," in *30th AIAA Applied Aerodynamics Conference*, (New Orleans, Louisiana), American Institute of Aeronautics and Astronautics, June 2012. `https://arc.aiaa.org/doi/10.2514/6.2012-2889`.

[188] W. F. Phillips, "Lifting-Line Predictions for Induced Drag and Lift in Ground Effect," p. 21.

[189] L. H. Jorgensen, "Method for Estimating Static Aerodynamic Characteristics for Slender Bodies of Circular and Noncircular Cross Section Alone and With Lifting Surfaces at Angles of Attack from 0° to 90°," *NASA Technical Note D-7228*, p. 40, Apr. 1973. `https://ntrs.nasa.gov/citations/19730012271`.

[190] L. H. Jorgensen, "Prediction of aerodynamic characteristics for slender bodies alone and with lifting surfaces to high angles of attack," in *NASA Technical Report R-474*, Jan. 1979. `https://ntrs.nasa.gov/citations/19790013852`.

[191] "Design of Aerodynamically Stabilized Free Rockets," Military Handbook MIL-HDBK-762, US Army Missile Command, Department of the Army, July 1990. `http://mae-nas.eng.usu.edu/MAE_5900_Web/5900/USLI_2010/PDF_files/rocket_handbook.pdf`.

[192] M. Drela, "Transonic low-Reynolds number airfoils," *Journal of Aircraft*, vol. 29, pp. 1106–1113, Nov. 1992. `https://arc.aiaa.org/doi/10.2514/3.46292`.

[193] J. Yost, "AeroSandbox 3.5.12 viscous calculation," December 2022. Email correspondence.

[194] D. Prosser, "LORAAX: Low Reynolds number aircraft analysis with XFoil," 2018. Github repository. `https://github.com/montagdude/loraax`.

[195] "Top Gun: Maverick," 2022. Film starring Tom Cruise.

[196] U.S. Geological Survey, "1 arc-second digital elevation models – USGS national map, 3DEP downloadable data collection." Raster digital data, 2022. Accessed: 2024-06-15.

[197] P. Sharpe and R. J. Hansman, "Core Components of an Optimization Framework for Engineering Systems based on Automatic Differentiation," in *AIAA AVIATION 2022 Forum*, (Chicago, IL & Virtual), American Institute of Aeronautics and Astronautics, June 2022. `https://arc.aiaa.org/doi/10.2514/6.2022-3937`.

[198] P. Váňa, A. A. Neto, J. Faigl, and D. G. Macharet, "Minimal 3D Dubins Path with Bounded Curvature and Pitch Angle," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 8497–8503, IEEE, 2020.

[199] Institute of Electrical and Electronics Engineers, *IEEE Standard for Floating-Point Arithmetic*, 2019. Revision of IEEE Std 754-2008.

[200] P. Prakash and N. Gomez, "TASOPT.jl," 2022. GitHub repository. `https://github.com/MIT-LAE/TASOPT.jl`.

[201] M. Kubale, ed., *Graph Colorings*. No. 352 in Contemporary Mathematics, Providence, R.I: American Mathematical Society, 2004.

[202] P. D. Sharpe and R. J. Hansman, "NeuralFoil: An airfoil aerodynamics analysis tool using physics-informed machine learning," 2024. Pre-print manuscript. `https://github.com/peterdsharpe/NeuralFoil/blob/master/paper/out/main.pdf`.

[203] P. Sharpe, "NeuralFoil." `https://github.com/peterdsharpe/NeuralFoil`, 2023. GitHub repository. Accessed 5-12-2024.

[204] M. E. Eleshaky and O. Baysal, "Airfoil shape optimization using sensitivity analysis on viscous flow equations," 1993.

[205] M. S. Selig and J. J. Guglielmo, "High-Lift Low Reynolds Number Airfoil Design," *Journal of Aircraft*, vol. 34, pp. 72–79, Jan. 1997. `https://arc.aiaa.org/doi/10.2514/2.2137`.

[206] R. H. Liebeck, "A class of airfoils designed for high lift in incompressible flow," *Journal of Aircraft*, vol. 10, no. 10, pp. 610–617, 1973.

[207] M. Drela, "Low-Reynolds-number airfoil design for the M.I.T. Daedalus prototype- A case study," *Journal of Aircraft*, vol. 25, pp. 724–732, Aug. 1988. `https://arc.aiaa.org/doi/10.2514/3.45650`.

[208] R. Eppler, "Airfoil design and data (PROFIL)," 1990.

[209] T. S. Tao, "Design of a series of airfoils for the PSU Zephyrus human-powered aircraft," 2010.

[210] M. Drela, *A User's Guide to MSES 3.05*. Massachusetts Institute of Technology, Cambridge, MA, USA, 2007.

[211] X. He, J. Li, C. A. Mader, A. Yildirim, and J. R. Martins, "Robust aerodynamic shape optimization—from a circle to an airfoil," *Aerospace Science and Technology*, vol. 87, pp. 48–61, 2019.

[212] J. Morgado, R. Vizinho, M. Silvestre, and J. Páscoa, "XFOIL vs. CFD performance predictions for high lift low Reynolds number airfoils," *Aerospace Science and Technology*, vol. 52, pp. 207–214, 2016.

[213] M. Drela, *Aerodynamics of Viscous Fluids*. Cambridge, Massachusetts: Massachusetts Institute of Technology, pre-publication draft ed., 2019.

[214] S. Zhang, *A Non-parametric Discontinuous Galerkin Formulation of the Integral Boundary Layer Equations with Strong Viscous/Inviscid Coupling*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 2017.

[215] A. Jameson, "Airfoils admitting non-unique solutions of the Euler equations," in *22nd Fluid Dynamics, Plasma Dynamics and Lasers Conference*, (Honolulu,HI,U.S.A.), American Institute of Aeronautics and Astronautics, June 1991. `https://arc.aiaa.org/doi/10.2514/6.1991-1625`.

[216] A. Kuzmin, "Non-unique transonic flows over airfoils," *Computers & Fluids*, vol. 63, pp. 1–8, 2012.

[217] S. L. Brunton and J. N. Kutz, *Data Driven Science & Engineering*. 2017.

[218] V. K. Truong, "An analytical model for airfoil aerodynamic characteristics over the entire 360° angle of attack range," *Journal of Renewable and Sustainable Energy*, vol. 12, p. 033303, May 2020. `http://aip.scitation.org/doi/10.1063/1.5126055`.

[219] K. Xu, M. Zhang, J. Li, S. S. Du, K.-i. Kawarabayashi, and S. Jegelka, "How Neural Networks Extrapolate: From Feedforward to Graph Neural Networks." `http://arxiv.org/abs/2009.11848`, Mar. 2021.

[220] S. F. Hoerner, *Fluid-Dynamic Lift*. Albuquerque/N.M: Hoerner, 2. ed ed., 1992.

[221] B. M. de Silva, K. Manohar, E. Clark, B. W. Brunton, J. N. Kutz, and S. L. Brunton, "PySensors: A Python package for sparse sensor placement," *Journal of Open Source Software*, vol. 6, no. 58, p. 2828, 2021.

[222] E. V. Laitone, "New Compressibility Correction for Two-Dimensional Subsonic Flow," *Journal of the Aeronautical Sciences*, vol. 18, pp. 350–350, May 1951. `https://arc.aiaa.org/doi/10.2514/8.1951`.

[223] M. Cranmer, "Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl." `http://arxiv.org/abs/2305.01582`, May 2023.

[224] W. H. Mason, "Transonic Aerodynamics of Airfoils and Wings," p. 24, Mar. 2006.

[225] M. Selig, "UIUC airfoil data site," 1996.

[226] JC. Etiemble, "TraCFoil and Free Pack Airfoils," Dec. 2023.

[227] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the Variance of the Adaptive Learning Rate and Beyond," 2019. `https://arxiv.org/abs/1908.03265`.

[228] M. Drela, "A User's Guide to MSES 3.05," July 2007.

[229] P. Spalart and S. Allmaras, "A one-equation turbulence model for aerodynamic flows," in *30th Aerospace Sciences Meeting and Exhibit*, 1992. `https://arc.aiaa.org/doi/abs/10.2514/6.1992-439`.

[230] F. Menter, RB. Langtry, S. Likki, Y. Suzen, P. Huang, and S. Völker, "A Correlation-Based Transition Model Using Local Variables—Part I: Model Formulation," *ASME J. Turbomach*, vol. 128, July 2006.

[231] F. R. Menter, P. E. Smirnov, T. Liu, and R. Avancha, "A One-Equation Local Correlation-Based Transition Model," *Flow, Turbulence and Combustion*, vol. 95, pp. 583–619, Dec. 2015. `http://link.springer.com/10.1007/s10494-015-9622-4`.

[232] S. Brunton, "Physics informed machine learning: High level overview of AI and ML in science and engineering," 2024.

[233] M. Drela and J. S. Langford, "Human-Powered Flight," *Scientific American*, vol. 253, pp. 144–151, Nov. 1985. `https://www.scientificamerican.com/article/human-powered-flight`.

[234] G. Wahba, *Spline Models for Observational Data*. No. 59 in CBMS-NSF Regional Conference Series in Applied Mathematics, Philadelphia, Pa: Society for Industrial and Applied Mathematics, 6. print ed., 2007.

[235] P. Sharpe and R. J. Hansman, "Physics-Informed Regression and Uncertainty Quantification of Aircraft Performance from Minimal Flight Data," in *AIAA SCITECH 2024 Forum*, (Orlando, FL), American Institute of Aeronautics and Astronautics, Jan. 2024. `https://arc.aiaa.org/doi/10.2514/6.2024-1306`.

[236] S. F. Hoerner, *Fluid-Dynamic Drag*. 1965.

[237] R. B. Gramacy, *Surrogates: Gaussian Process Modeling, Design and Optimization for the Applied Sciences*. Boca Raton, Florida: Chapman Hall/CRC, 2020. `http://bobby.gramacy.com/surrogates/`.

[238] Trevor Hastie, Robert Tibshirani, and Jerome Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics, 2 ed., 2017.

[239] J. Cook, "Basic properties of the soft maximum." `https://www.johndcook.com/soft_maximum.pdf`, Sept. 2011.

[240] T. S. Tao, *Design, Optimization, and Performance of an Adaptable Aircraft Manufacturing Architecture*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, Feb 2018.