AeroSandbox: A Differentiable Framework for Aircraft Design Optimization

by

Peter Sharpe

B.S., Washington University in St. Louis

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science in Aeronautics and Astronautics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author . . . . . .
Department of Aeronautics and Astronautics
May 18, 2021

Certified by . . . . . .
R. John Hansman
T. Wilson Professor in Aeronautics
Thesis Supervisor

Accepted by . . . . . .
Arthur C. Chairman
Chairman, Department Committee on Graduate Theses

AeroSandbox: A Differentiable Framework for Aircraft Design Optimization

by

Peter Sharpe

Submitted to the Department of Aeronautics and Astronautics
on May 18, 2021, in partial fulfillment of the
requirements for the degree of
Master of Science in Aeronautics and Astronautics

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam quis neque et erat laoreet finibus at ac leo. Curabitur pellentesque, diam quis dignissim finibus, enim dui feugiat leo, nec porttitor sapien mi ac felis. Nam aliquam pretium nibh, quis dapibus dolor gravida sit amet. Cras porttitor dui quis elementum pulvinar. Nulla id pulvinar massa. Nullam ut diam non lorem venenatis faucibus. Vivamus lacus ante, pellentesque vitae nisl sit amet, bibendum facilisis purus.

Thesis Supervisor: R. John Hansman
Title: T. Wilson Professor in Aeronautics

Acknowledgments

This is the acknowledgements section.

Contents

Chapter 1

Introduction

1.1 Engineering Design Optimization

Optimization is all around us, as it is a formalization of the ubiquitous process of decision-making: "How do we make the best decision with limited resources, under some assumptions and models of the world around us?" Indeed, nearly every designed system in our lives is the result of an optimization process, be it formal or informal.

In the past century, advances in optimization algorithms and computing have made it tractable to formulate and solve increasingly sophisticated problems within an optimization paradigm. This new lens has fundamentally changed nearly every field of engineering and commerce, and mathematical optimization underlies everything from Google Maps to economic policy to neural networks to aircraft design[1]. Today, optimization is one of the workhorse tools of scientific computing, and it is one of the problems consuming the most CPU cycles around the world at this very moment. Its practical importance has not gone unnoticed: in a revised list of the "Top Ten Algorithms of the 20th Century" presented by Nick Higham (former SIAM President), optimization claims two distinguished spots[2] [22].

Optimization is particularly useful in engineering, where design goals naturally

[1] As a colleague once quipped, "optimization is the practice of turning math theorems into money."
[2] via Newton/quasi-Newton methods, ranked #1; and the simplex algorithm, ranked #9.

lend themselves to this framework. To paraphrase R. John Hansman, the goal of engineering design is to find an *optimal* mapping from a project's *function* to its *form*. Engineers begin the design process by identifying a problem or need, and through conceptualization and optimization they arrive at a physical form that can solve that problem.

1.1.1 Configuration and Sizing

The process of engineering design optimization can be broadly partitioned into two halves, adapting terminology from Raymer [36]:

1. **Configuration**. Configuration involves the qualitative identification of viable solution *archetypes*, and eventually, the optimal topology of the solution. Design choices here are often discrete; for example, aircraft configuration might involve a choice between an airplane and a helicopter, or between propeller and jet propulsion. Colloquially, we ask "what should the system look like?"

   Because this configuration stage is discrete and principally concerned with defining the viable solution space itself, it is a highly creative process best served by human intuition and experience. In general, configuration is not well-served by formal optimization: any attempt at transcribing the configuration problem to a discrete optimization problem requires one to make unnecessary assumptions about the solution space[a].

2. **Sizing**. Sizing forms the other half of design. Here, one assumes a given specific configuration and aims to find the optimal choice of size, weight, and power ("SWaP") for system components. Design choices here are often continuous; for example, aircraft sizing might involve the choice of a wingspan or a desired cruise thrust. Colloquially, we ask "how big should the system be?", or "what is

the *best possible version* of the given configuration?"

When compared with configuration, sizing is a much more quantitative process - in fact, sizing is primarily based on rigorous performance analysis. Because of this, formal optimization methods hold great potential in addressing the sizing problem, and this the domain addressed by the new design framework presented in this thesis.

[a]Often, this occurs unintentionally. For example, a formal configuration optimization problem might ask "how many wings should the airplane have?", when in reality, the true optimal solution is not an airplane, but a train.

In existing literature, this dichotomy between "configuration" and "sizing" has been addressed most often in the context of early-stage conceptual aircraft design. However, this distinction is not limited to either conceptual design or aerospace applications; all engineering design can be split into configuration (the "what") and sizing (the "how much").

While it may initially seem that sizing is addressed exclusively after a configuration has been selected, the relationship between configuration and sizing is actually quite symbiotic. For example, in order to decide between configurations A and B, one needs to compare the *best* version of configuration A with the *best* version of configuration B; here, a configuration decision depends on the result of a series of sizing studies.

Because of this, any computational tool that aims to aid in solving the quantitative "sizing optimization problem" must seamlessly integrate with the human "configuration design problem". This means that a sizing optimization tool must be fast, accessible, robust, flexible, and accurate enough to allow *interactive design*, a process by which an engineer works in tandem with an optimizer to continuously pose and answer questions in order to understand the design space.

1.2 The Aircraft Design Problem

In this work, we focus on developing a new software tool to address the problem of *conceptual aircraft design*, and in particular that of aircraft sizing. Raymer [36] provides an excellent summary of the sizing problem:

> "To [others], our process of aircraft sizing seems backwards. Most people would assume that we draw a new aircraft design and then determine how far it goes.
>
> We do it the other way around. We know how far it goes – it goes as far as the requirements say it goes. What we do not know ...is how big to draw it."
>
> - Raymer [36]

Here, Raymer makes the crucial observation that sizing (and, in reality, all aircraft design) is intrinsically driven by requirements. As the requirements change, the optimal design does too; sizing is a dynamic process.

Raymer also begins to hint at a distinction between the forward problem and the inverse problem, and the relative utility of these approaches. It is an unfortunate but necessary reality that the majority of engineering education focuses on the *forward problem*, also termed "analysis" or "simulation": "Given some design, what performance is achieved?" In real-world practice, it is often far more useful to turn the problem around and work with the *inverse problem*, also termed "design" or "optimization": "Given some required performance, what does the design look like?" Indeed, engineering design at its core is an inverse problem, not a forward one.

1.2.1 A Canonical Design Problem

Because problem formulation is by far the most important step of any optimization problem, it is instructive to give a canonical example of the sizing problem before discussing methods to solve it.

The sizing problem can be made more quantitative via formulation within a standard mathematical optimization framework. The sizing problem is generally representable as a continuous nonlinear program (NLP) of the form:

$$\begin{aligned} \underset{\vec{x}}{\text{minimize}} \quad & J(\vec{x}) \\ \text{subject to} \quad & \vec{g}(\vec{x}) \geq 0, \\ & \vec{h}(\vec{x}) = 0 \end{aligned} \tag{1.1}$$

This standard-form mathematical optimization problem consists of a few key elements: variables, objective, constraints, and parameters. We typically map the given inputs to the sizing problem to these elements as follows. Canonical examples of each formulation element are presented, as might be applicable to the design of an aerospace system.

1. **Variables** $\vec{x}$. The degrees of freedom in the system, or the quantities that are used to describe any particular design in the chosen parameterization. The number of independent variables sets the dimensionality of the design space.

   Canonical examples include variables parameterizing the vehicle's outer mold line (OML), propulsion sizing, mechanism design. For mission-driven optimization, variables parameterizing the vehicle's state and control inputs over time throughout the nominal mission might be included here as well.

2. **Objective function** $J(\vec{x})$. The design objective or performance metric, expressed as a quantity that is to be minimized.

   Canonical examples include size, weight, power, and cost. If the quantity is to be maximized (e.g. range), the quantity is negated in order to convert the problem into a standard-form minimization problem.

3. **Constraints** $\vec{g}(\vec{x}) \geq 0, \vec{h}(\vec{x}) = 0$. Constraints bound the feasible space. Constraints are typically drawn from two sources: physical models and given requirements. Because of this, constraints tend to be the most interesting part

of a design optimization problem as they encode the vast majority of problem information.

A problem may contain both active[3] constraints and inactive constraints. Active constraints are those that limit the objective function at the optimal point. Each constraint is associated a dual variable that effectively quantifies the sensitivity of the objective to an incremental scalar addition to one side of the constraint; for active constraints, this sensitivity is nonzero by definition.

Equality constraints can be thought of as a dimensionality reduction of the design space: adding an equality constraint effectively projects the design space onto the hypersurface that satisfies that equation.

4. **Parameters**. Parameters typically represent mutable assumptions embedded within the objective function or constraints. Parameters are quantities that an engineer might later perform a sweep over in order to study the feasible space under varying assumptions.

   Canonical examples might include technology factors (e.g. battery specific energy), economic factors (e.g. fuel specific cost), risk metrics (e.g. structural factor of safety over design loads), or mission requirements (e.g. required range).

In this work, we present a new computational framework for solving this aircraft design optimization problem based on modern advances in optimization methods. Chapter 2 motivates the need for a specialized, high-performance optimization framework for aircraft design. Chapter 3 introduces the central contribution of this thesis: a new optimization framework. Chapters 4, 5, and 6 elaborate more on various components of this framework, with each new chapter adding a level of abstraction that build upon previous work. Finally, Chapters 7 and 8 are aircraft design case studies that serve to illustrate specific features of this new framework and the framework's applicability to real problems.

[3] also referred to in some literature as "tight" or "driving"

Chapter 2

Challenges of Aircraft Design Optimization

Aircraft design optimization problems tend to present several unique challenges when compared to general engineering design optimization. Here, we identify a few of these challenges that have significant ramifications for the architecture of a general-purpose computational framework for aircraft design optimization.

2.1 Modeling and Optimizing Dynamic Systems

Perhaps the first major challenge with aircraft design optimization is that we are optimizing systems that are inherently time-dependent and dynamic: colloquially, performance depends not only on *what* you fly, but also *how* you fly. Therefore, we require a means to optimize parameters of dynamical systems. An example of this would be optimizing the wing area of an airplane throughout a prescribed mission - a single, fixed value must provide optimal performance when integrated across all flight conditions.

This capability to optimize systems that must operate at many different operating points is critical in aircraft design. With rare exceptions, nearly all aircraft operate at very different operating points throughout their missions. Examples of this time-dependency in aerospace problems abound.

Examples of Time-Dependency in Aerospace Design

- A Boeing 787 Dreamliner has a fuel mass fraction of 44.4%[a] [1]. To first order[b], this implies that the aircraft's design lift coefficient also changes by a similar amount over the duration of a max-range flight. An analogous fuel-burn consideration will apply to the design of any combustion-powered flight vehicle, which represent the vast majority of aerospace design problems.

- A solar-powered airplane relies on a cyclical injection of energy into the airborne system via solar insolation. Given the availability of gravitational potential energy as a means of energy storage, energy-optimal trajectories need not be steady-state over a daily cycle.

- The performance of an orbital launch vehicle is highly dependent on its ascent profile; complex trades between altitude, engine performance, dynamic pressure, and vehicle recoverability abound, with strong implications for even first-order design.

- The performance of urban air mobility vehicles depends heavily on takeoff profiles, landing profiles, and emergency cases; these often represent max-power flight conditions that size vehicle propulsion and power systems.

[a] With a max fuel load of 223,378 lbs. and a max design takeoff weight of 502,500 lbs.

[b] The cruise airspeed of a commercial airliner is typically roughly constant-Mach, and cruise altitude is assumed constant.

All of these examples point to the same conclusion, which is that engineering systems typically must operate well at a variety of conditions. In other words, optimality at a point condition is insufficient - designs must be *robust* across some range of expected operating conditions. There are several ways that this time-dependency is traditionally handled in engineering design optimization:

2.1.1 Approach 1: Steady-State Point Reduction

In the most general sense, the time-evolution of any dynamical system (such as an aircraft) can be expressed as a system of nonlinear, coupled ordinary differential equations [4]:

$$\frac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \begin{bmatrix} \frac{\mathrm{d}\left(x_1(t)\right)}{\mathrm{d}t} \\ \frac{\mathrm{d}\left(x_2(t)\right)}{\mathrm{d}t} \\ \cdots \end{bmatrix} = \begin{bmatrix} f_1\left(\vec{x}(t), \vec{u}(t), t\right) \\ f_2\left(\vec{x}(t), \vec{u}(t), t\right) \\ \cdots \end{bmatrix} \tag{2.1}$$

where: t = Time

$\vec{x}(t)$ = State vector, which fully describes the state of the system at time t

$\vec{u}(t)$ = External control inputs, such as a surface deflection or throttle setting

$f_i()$ = Functions that describe system evolution (often, equations of motion analogous to $F = ma$)

The simplest method of handling time-dependency here is to make the steady approximation that the system remains at some *trim state* for the entire duration of the mission; for an aircraft, this typically means that all state variables except for position are constant. This implies that the control vector $\vec{u}(t)$ is set in order to make this statement true; for example, throttle setting is set so that airspeed is held constant. Under these assumptions, one can reduce the unsteady system dynamics down to a single, steady operating point that is expected to be representative of the operating envelope.

An example of this approach is found in *SimpleAC*, a canonical problem for passenger aircraft design presented as part of GPKit, a library for aircraft design optimization using geometric programming; a full description of this design problem is given later in Equation 4.13. The relevant example from SimpleAC here is the constraint associated with lift-weight balance, which is given as:

$$W_0 + W_w + \frac{1}{2}W_f \leq \frac{1}{2}\rho V^2 C_L S \tag{2.2}$$

where: W_0 = Empty weight

W_w = Wing weight

W_f = Fully-loaded fuel weight

ρ = Air density

V = Airspeed

C_L = Cruise lift coefficient

S = Wing area

In other words, the constraint assumes that the aircraft's cruise weight throughout flight is well-approximated by the weight of the aircraft with half of its fuel burnt. This performance relation is a steady-state point reduction that aims to address the inherent time-dependency of the fuel-burn relation. Unfortunately, because the effect of aircraft weight on fuel burn rate has significant nonlinearity (especially given the large fuel fractions typical of a passenger aircraft), this is a relatively poor assumption.

Point Reduction by Closed-Form Approximation

A more sophisticated way to address the time-dependency of fuel burn is the Breguet range equation[1] [36]. The Breguet equation is a closed-form model for the fuel-burn time dependency over a given mission segment; its derivation uses energy balance to make an implicit closure for the fuel burn rate that can then conveniently be integrated analytically. The equation is stated in its common form in Equation 2.3:

$$R = V\left(\frac{L}{D}\right) I_{sp} \ln\left(\frac{W_i}{W_f}\right) \tag{2.3}$$

where: R = Range over a given mission segment

L/D = Lift-to-drag ratio

I_{sp} = Propulsor specific impulse

W_i = Initial total aircraft weight (beginning of mission segment)

W_f = Final total aircraft weight (end of mission segment, after fuel burn)

[1] The reason that SimpleAC uses the half-fuel-burn assumptions rather than the Breguet equation is that the GPKit geometric programming framework is unable to model logarithms.

Although the Breguet equation does attempt to model the higher-order dynamic effect of fuel burn, it is still effectively a point relation because it inevitably leads to the question of which operating point the inputs V, L/D, and I_{sp} should be evaluated at.

Furthermore, while the Breguet equation is more accurate than a point reduction at the half-fuel-burn state, it too has serious limitations. For example, the Breguet range equation assumes that L/D is independent of wing loading, despite the fact that in our 787 Dreamliner example, wing loading decreases by approximately 44% throughout flight. Furthermore, it is assumed that propulsor I_{sp} is independent of the fuel burn rate, a quantity that the Breguet model also predicts will decrease by approximately 44% over the course of the flight. The Breguet equation also leaves no clear strategy for addressing climb or descent: even if a vertical speed energy correction is made, the effect of continuously-varying ambient air density and associated downstream effects on vehicle performance is not captured. Finally, path constraints (such as an arbitrary time-varying Mach number schedule or airspace limitations) are impossible to implement with this approach.

There is also a deeper and more subtle problem with any attempt to address dynamics by reduction to a single operating point: design optimization within a high-dimensional design space at a single operating point inevitably leads to highly-sensitive designs that perform poorly in practice. This phenomenon is elegantly demonstrated by Drela in several airfoil design case studies, where an optimizer exploits a single given operating point to find a high-performing design with exceptionally-poor off-design performance [9]. As one might suspect, multi-point optimization is a strategy to mitigate this, and Drela concludes that robust design optimization of smooth geometry generally requires a number of operating points roughly comparable to the number of degrees of freedom.

For all these reasons, design optimization around a single fixed operating point is clearly insufficient for any study beyond first-order in the vast majority of aerospace cases.

2.1.2 Approach 2: Multiple Segments with Point Reduction (Multipoint Optimization based on Segments)

For slightly improved fidelity, one can split the given mission up into segments. For an aircraft, a prototypical mission might consist of the following segments [36]:

1. Takeoff and Climb
2. Cruise
3. Loiter
4. Cruise
5. Descent and Landing

For each of these segments, a representative operating point (or closed-form approximation to one analogous to the Breguet equation) is chosen. In addition, each of these operating points might be augmented with various perturbations about this point, allowing consideration of off-design performance.

Performance is analyzed at each of these individual points. Finally, to obtain a scalar objective function, some reduction (e.g. integration with respect to time, linear combination, or worst-case performance) is then done by combining performance metrics across all analyzed operating points.

This is a basic example of *multipoint optimization*, an approach to robust design where a design is optimized based on an aggregate of its performance at a finite number of operating conditions.

2.1.3 Approach 3: Full Simulation via ODE Integration

Both of these methods presented thus far are steady-state or piecewise-steady-state point reductions to what is truly an unsteady system of ODEs (as illustrated in Equation 2.1). We have demonstrated that these simple approaches lead to inaccurate performance analysis and drive an optimization study to brittle, overly-sensitive designs.

An alternative approach that alleviates both of these problems is to directly simulate the flight vehicle throughout its mission, without any kind of *a priori* segmentation. This approach significantly improves modeling fidelity and flexibility on problems where dynamics are important (which represent the vast majority of aerospace problems).

This "full simulation" approach is the one most studied in the present work, because the optimization framework described in Chapter 3 is especially suited to solving these problems.

Selection of State Variables

To simulate the full vehicle dynamics (as something like a flight simulator might do), we first need to identify the relevant state variables. There are several common sources for these state variables:

- **Flight Dynamics:** Any rigid free-flying body has 12 state variables that describe its motion at a given moment in time [13]. These can be grouped into four categories, each of which is a vector containing three degrees of freedom[2]: position, orientation, velocity, and angular velocity. A Cartesian illustration of these 12 state variables is given in Figure 2-1. Each of these 12 state variables has an associated ODE related to the equations of motion.

- **Power Systems Accounting:** Any aircraft with a depletable onboard power source (e.g. stored fuel or a battery) will have a state variable that describes the energy remaining[3]. The associated ODE relates energy depletion rate to a control input such as throttle setting.

- **Propulsion State:** Aerospace propulsion systems exhibit hysteresis (i.e. "lag") in response to control inputs. Many notable examples, such as turbofan spool time and quadcopter propeller inertia, result in dynamic performance limits

[2] In a Cartesian sense, each category (e.g. position) has x, y, and z components in some frame

[3] In reality, a system might have multiple state variables here to correspond to multiple fuel tanks or batteries, but this consideration is neglected here for simplicity.

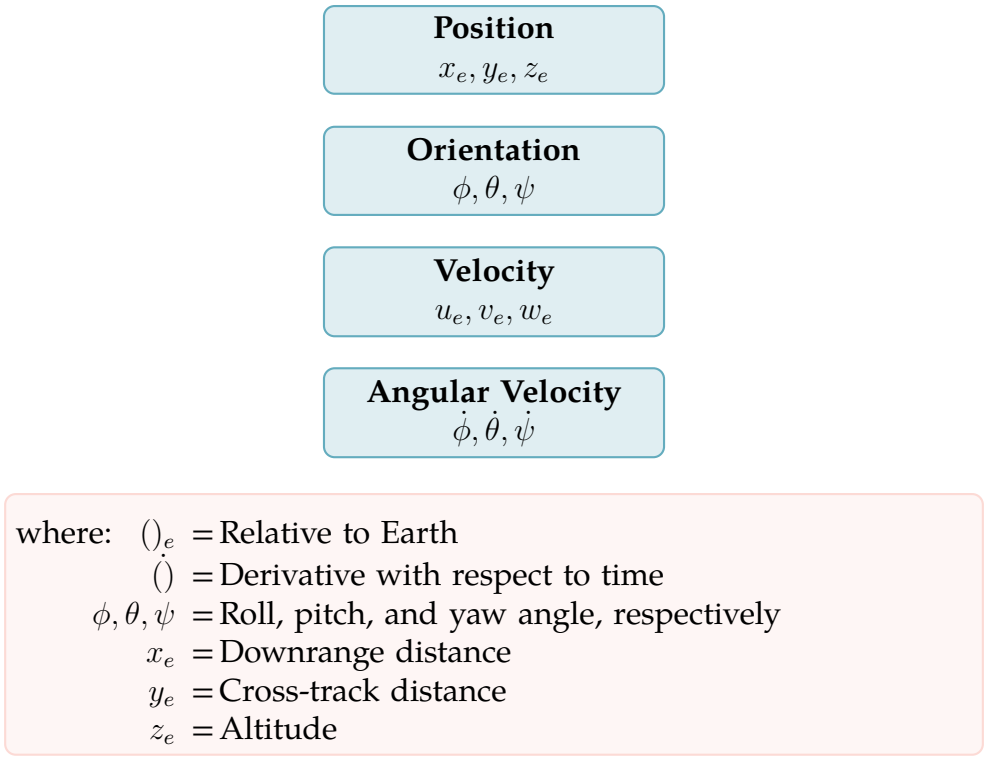

Figure 2-1: One possible parameterization of the 12 flight dynamics state variables. Notation adapted from [13].

that are significant enough to have direct operational consequences. However, these lag effects typically do not affect vehicle performance in a way that is significant to design optimization, so they are neglected for the remainder of this work.

- **Flexible Airframes:** Flexible bodies (e.g. a wing experiencing considerable aeroelastic effects) contribute an infinite number of state variables to a dynamic system. (In practice, this would be discretized to a large but finite number of structural degrees of freedom.) Due to the difficulty of modeling this, we often assume that any coupling between aeroelastic effects and rigid body flight dynamics can be adequately treated with surrogate models as in Section 5.2; thus, this coupling is neglected for the remainder of this work.

Thus, flight dynamics and power systems accounting represent the two categories of state variables that we nearly always need to account for. Between these

two categories, a basic aerospace system typically has 13 state variables.

However, many of the 12 flight dynamics state variables can be removed with minimal loss of fidelity for the purposes of design optimization. Recalling that the sizing problem is effectively performance analysis, we can remove variables that have insignificant impact on overall vehicle performance.

The first simplifying assumption that can be made is the *2D reduction*, where the 3D dynamics are projected onto the 2D range-altitude space. In other words, in the notation described in Figure 2-1, the state variables of roll ϕ, yaw ψ, roll rate $\dot{\phi}$, yaw rate $\dot{\psi}$, cross-track position y_e, and cross-track velocity v_e are all assumed to be zero and neglected. A side effect of this is that any lateral flight dynamics modes[4] are neglected. This leaves six flight dynamics state variables: downrange distance x_e, altitude z_e, downrange speed u_e, vertical speed w_e, pitch θ, and pitch rate $\dot{\theta}$. A prerequisite assumption for making this 2D reduction is that cross-track dynamics have no impact on key performance metrics such as range and endurance.

A further assumption that can be made is the *quasi-steady* assumption, where the pitch rate $\dot{\theta}$ is assumed to be negligibly small. In the $\dot{\theta} \to 0$ limit, the net pitching moment is zero. For an airplane, this quasi-steady simplification essentially assumes that the short-period and phugoid flight-dynamics modes are sufficiently *spectrally-separated*[5] that the short-period mode can be neglected.

This assumption of an instantaneous short-period mode implies that any trimmable flight condition[6] is reachable instantaneously; in other words, the angle of attack $\alpha(t)$ can be directly prescribed as a control input [7]. It is therefore useful to decompose the pitch angle θ into a flight path angle γ and an angle of attack α, as shown in equation 2.4:

$$\theta(t) = \gamma(t) + \alpha(t) \tag{2.4}$$

[4] e.g. spiral mode or Dutch roll

[5] A full treatment of this spectral separation assumption is available in [13]

[6] defined as any condition where $\sum F = \sum M = 0$ can be achieved with allowable control inputs

[7] subject to the zero-net-moment constraint, which is typically satisfied via control surface deflections.

where: γ(t) = arctan $\left(\frac{w_e}{u_e}\right)$, the flight path angle

α(t) = Angle of attack

With this decomposition, we eliminate one more state variable, as $\alpha(t)$ is a control input and $\gamma(t)$ is a pure function of the existing state variables u_e and z_e. So, after both of these assumptions, only the four flight dynamics state variables listed in Equation 2.5 remain:

$$\begin{aligned} x_e(t)&\text{: Downrange distance} \\ z_e(t)&\text{: Altitude} \\ u_e(t)&\text{: Downrange speed} \\ w_e(t)&\text{: Vertical speed} \end{aligned} \tag{2.5}$$

In numerical schemes, the two velocity variables u_e, w_e are often instead parameterized as airspeed $V(t)$ and flight path angle $\gamma(t)$. This is because the V-γ velocity parameterization tends to be more energy-conserving upon numerical ODE integration than the Cartesian parameterization[8], as $V(t)$ maps directly onto vehicle kinetic energy.

Trajectory Optimization, Transcription, and Discretization

Using this ODE approach, the state of the modeled aerospace system can be represented as a series of four functions of time:

$$\begin{aligned} x_e(t)&\text{: Downrange distance} \\ z_e(t)&\text{: Altitude} \\ V(t)&\text{: Airspeed} \\ \gamma(t)&\text{: Flight path angle} \end{aligned} \tag{2.6}$$

These state variables are general functions of time, and a general function space such as this is an infinite-dimensional: specifically, it is a Hilbert space. In other

[8]For simplicity here, ambient wind speed is assumed to be zero, so $V = \sqrt{u_e^2 + w_e^2}$. Nonzero wind is discussed in Section **??**.

words, an infinite amount of information is required to fully describe any general function. Even functions with a finite domain (e.g. airspeed over a fixed-duration mission) are infinite-dimensional, as a general representation still requires an infinite amount of information[9].

A further complication here is that the optimal functions for the state variables (collectively referred to as the *trajectory*) are not known *a priori*; indeed, this represents a *trajectory optimization* problem that must be solved simultaneously with the vehicle optimization problem.

General numerical optimization in an infinite-dimensional space is intractable[10], so we must perform some *transcription* (a generalization of discretization) that converts[11] the continuous problem to a tractable finite-dimensional nonlinear program. Many transcription approaches are possible; an excellent review is presented by Kelly in [26] and [25]. Examples in the present work generally transcribe ODEs with direct trapezoidal collocation; a full description of this algorithm and broader ODE treatment as relevant to the present work follows in Section 4.7.

We make a final note here that trajectory optimization is a much more challenging task than simple numerical integration of an ODE. This is primarily because every numerical ODE integrator has *error* due to discretization. In traditional ODE integration, this is a nuisance that can be resolved with increased resolution or higher-order schemes (h- and p-refinement strategies, respectively). However, in trajectory optimization, the optimizer acts as an adversary that is perennially seeking to exploit and magnify this discretization error. In essence, the optimizer will seek to find any flaw with the ODE integrator and break it in a way that suits the optimization objective. Section 4.7 details several approaches to combat this.

[9] An illustration of this is the fact that it takes an infinite number of Fourier terms to exactly represent an arbitrary function on a finite domain.

[10] Calculus of variations is a powerful strategy for infinite-dimensional optimization in some cases, though this is an analytical approach rather than a numerical one.

[11] or, more precisely, *projects*

Implications for Design Optimization

This means that the aircraft sizing problem can be decomposed into two strongly-coupled subproblems:

1. **Vehicle Design**, which concerns vehicle and component sizing.

2. **Mission Design**, which is effectively a trajectory optimization problem.

2.2 High-Dimensional Optimization

A second common challenge of aircraft design optimization is that design spaces tend to be quite high-dimensional. While first-order sizing studies might only consider a few variables (e.g. wing aspect ratio, wing area, and cruise airspeed), more sophisticated studies can easily have hundreds or thousands of design variables.

There are two primary culprits for the high number of design variables in aerospace problems: dynamics parameterization and outer mold line[12] (OML) parameterization. The rationale for the high dimensionality of dynamics parameterization was previously addressed in Section 2.1.3: the state variables $x_e(t), V(t)$, etc. are functions of time, and time discretization results in many degrees of freedom.

Outer mold line (OML) parameterization is high-dimensional for a very similar reason: the curving, spline-like surfaces common in aerospace vehicles are also described by functions[13], and many design variables are required to represent these functions to sufficient accuracy.

A simple and common aerospace example that illustrates this challenge is the problem of airfoil geometry parameterization. There are several commonly used parameterization methods, such as the Kulfan CST parameterization, B-splines, and Hicks-Henne bump functions. However, Masters demonstrates that all methods require a minimum of approximately 40 design variables to capture a reasonable

[12] The outer mold line represents the outermost, air-facing surface of the vehicle - essentially, the "vehicle shape".

[13] So, a general representation of a curved surface once again requires infinite information.

airfoil design space to within engineering tolerance[14] [32]. Even still, 40 degrees of freedom yields a relatively coarse representation by analysis standards, as shown in Figure 2-2.

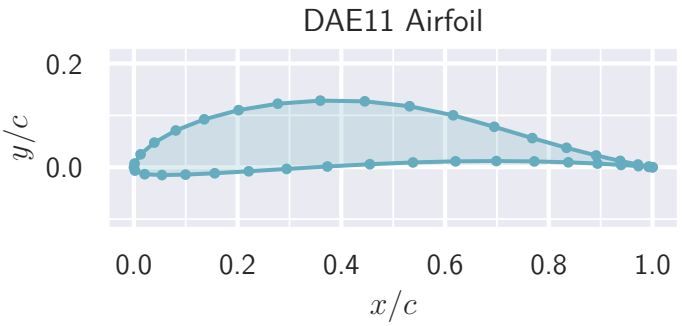

Figure 2-2: Illustration of the coarseness of an airfoil parameterized by 40 degrees of freedom[16], roughly the minimum needed for airfoil design optimization [32].

When we consider that an airplane often has several different airfoils along its span, each of which is optimized for a different local flow field, it is clear that reasonably-accurate parameterization of an aircraft OML might easily require hundreds of variables.

2.2.1 Addressing the Curse of Dimensionality

The high dimensionality of aircraft design optimization problems is a challenge, because the volume[17] of the feasible space tends to grow exponentially with respect to the number of dimensions. This phenomenon is referred to in the literature as the "curse of dimensionality" [31, 25]. The curse of dimensionality makes many optimization approaches wholly intractable, so it is important to consider which optimization methods might be effective on high-dimensional problems.

[14] defined here as corresponding to an aerodynamic performance error of less than one lift count and one drag count.

[16] Assumes each node contributes one degree of freedom; a reasonable approximation because only normal node movement affects OML shape.

[17] Colloquially defined here as "the number of meaningfully-different feasible points"

All optimization methods are a tradeoff between *exploration* and *exploitation* [17, 24]. Exploratory algorithms "leave no stone unturned", spending many function evaluations exploring the design space to increase the confidence that an optima is a global one. Exploitative algorithms hone in on a local optima as fast as possible, dramatically reducing the number of function evaluations required at the expense of reduced confidence in global optimality. In practice, this classification is a spectrum: this is visualized in Figure 2-3, where several popular optimization algorithms from Kochenderfer [28] and Nocedal [33] are roughly labeled.

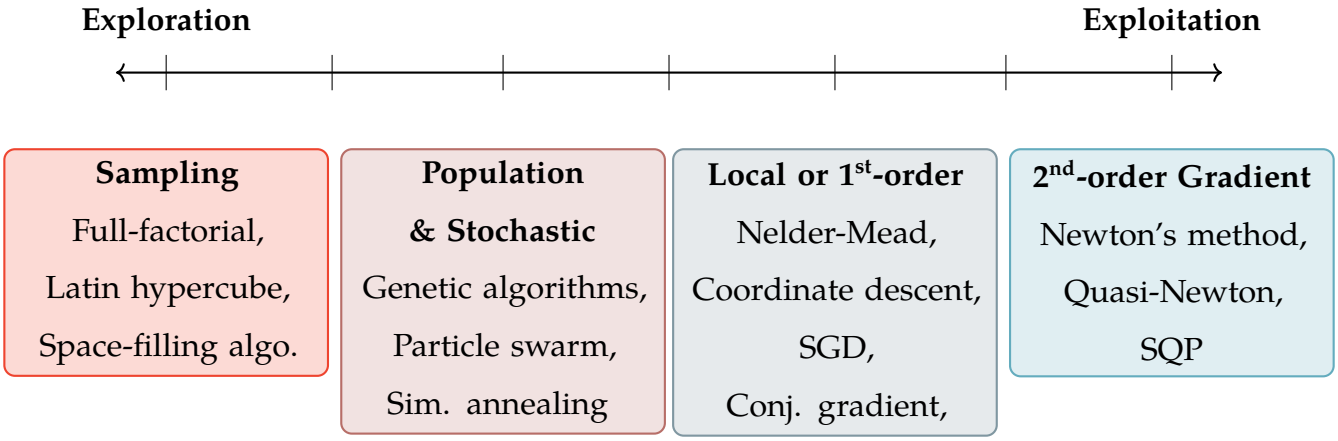

Figure 2-3: A Short Taxonomy of Optimization Methods: Exploration vs. Exploitation.

Neither approach is generally superior to the other, and problems that are trivial with one approach may be wildly intractable with another. In fact, the aptly-named "No Free Lunch" theorem of optimization, as demonstrated by Wolpert and Macready [40], proves that increased optimizer performance on one class of problems will always come at the expense of performance in another. In effect, proper selection of an optimization algorithm for a problem requires some *a priori* intuition about the objective function and constraints[18].

When solving aircraft design optimization problems, there are two primary reasons that it becomes prudent to lean as heavily as possible towards the exploitation

[18] In some optimization methods, namely Bayesian optimization, this *a priori* intuition is formally specified as a Bayesian prior, allowing a continuous choice between exploration and exploitation.

side of the optimization spectrum:

1. High-dimensional optimization is completely intractable with exploratory methods due to the curse of dimensionality. This difference in tractability is extreme: in a review of optimization methods for aerodynamic design optimization, Lyu et al. found that population-based methods required approximately 10^6 times as many function evaluations to reach optimality compared to second-order gradient-based methods [30] for a 100-variable problem[19].

2. Objective and constraint functions tend to be smooth, or they can be well-approximated by smooth functions with minimal labor. More precisely, functions tend to be at least C^1-continuous, which makes them much more amenable to second-order gradient-based methods.

Because of this, second-order gradient-based methods are easily the tool of choice for aircraft design optimization. Although critics often point out that no guarantees of global optimality can be made, this is often far less of a concern than perhaps initially perceived. As it turns out, most aerospace sizing problems are unimodal despite their complexity. Martins concludes in [31] that "it is assumed far too often that any complex problem is multimodal, but that is often not the case" and "therefore, one should assume that a function is unimodal until proven otherwise."

2.3 Addressing Coupled Problems

A final challenge of aircraft design optimization is that problems are highly coupled between disciplines. This phenomenon is illustrated in Figure 2-4, which shows the driving dependencies that might be considered in the design of a passenger aircraft. Many of these interactions are examined thoroughly by Drela in literature related to the TASOPT design code [12, 14].

[19] which is a relatively small problem by most aircraft design standards

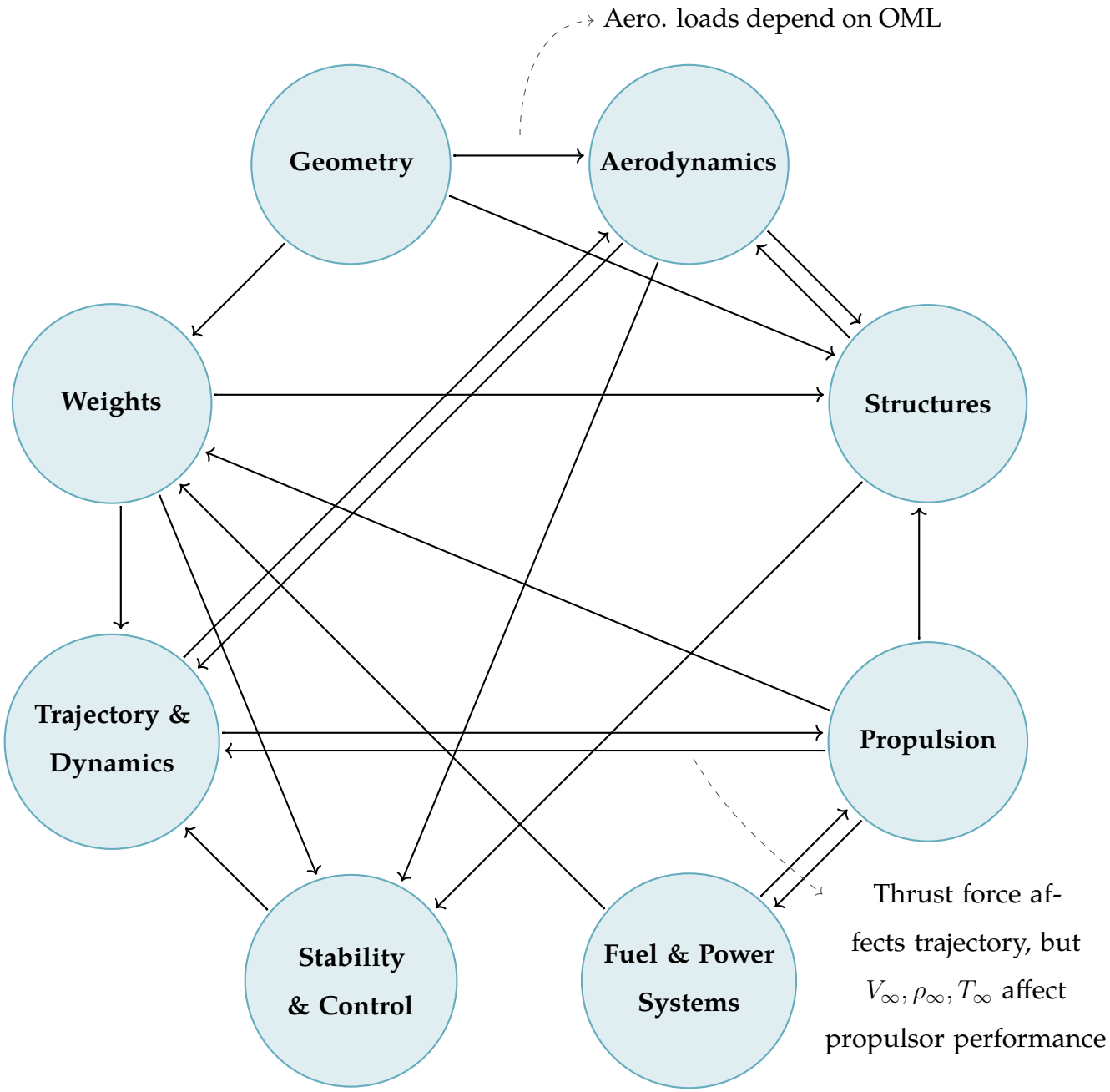

Figure 2-4: Commonly-studied subsystem dependencies for the design of a passenger aircraft. Arrows denote influence of one subsystem on another.

A key challenge that is observed in Figure 2-4 is the presence of internal *closure loops*, or implicit relationships between subsystems. Implicit relations are those that cannot be enforced by explicit, step-by-step calculation; instead, they represent a nonlinear relationship that must be enforced iteratively.

Traditional aircraft design methodologies (such as the one described in detail by Raymer [36]) resolve these implicit relationships with a "guess and check" approach: to "close" any given design, an initial guess is assumed at some point, and the loop is iterated until closure. Mathematically, this is analogous to solving a system of nonlinear equations by Gauss-Seidel iteration[20]; it is not the worst approach for small problems, but it scales exceptionally poorly to problems with many interacting disciplines. Furthermore, closing these implicit relations via "guess and check" means that we would effectively perform "sub-iterations" at each iteration of our optimization algorithm; this is obviously far from ideal if computational performance is desired. Superior methods of solving this closure problem are implemented in the present work and presented in Section 4.6.1.

2.3.1 The Origins of Coupling

Coupling between disciplines is not unique to aerospace design, although we claim here that it is generally far more of a challenge in aerospace than in other engineering applications. We posit this hypothesis for two reasons:

1. On a free-flying aerospace system, there is rarely something that can act as a global source or sink for conserved quantities such as mass, momentum, energy, heat, and charge. If we imagine some classical engineering design problems, we observe a similar trend:

    - The design of a typical cantilevered beam assumes a fixed support
    - The design of a typical circuit assumes an electrical ground
    - The design of a typical automobile engine assumes some place for heat rejection.

    By contrast, aerospace systems have nothing to "push off of": each force and moment must be perfectly balanced with another to achieve steady, level

[20] Another analogue to the "guess-and-check" approach is coordinate descent optimization, which mimics the same pattern of sequential, orthogonal, axis-aligned movement throughout the variable space.

flight. This requirement for balance between disciplines often manifests as a cross-discipline constraint, leading to coupling.

2. Aerospace systems often exhibit high performance sensitivities with respect to size, weight, and power. These sensitivities conspire to encourage multi-functional and highly-integrated design. While this design philosophy leads to enhanced performance, it inevitably means that any subsystem's allocation to size, weight, or power comes at the direct expense of another's.

2.4 Summary

Here, we have discussed several key challenges that are especially prominent in aircraft design optimization:

1. Aerospace systems are inherently dynamic, and hence require careful modeling.

2. Aerospace systems are high-dimensional, which precludes the tractable use of many kinds of optimization algortihms.

3. Aerospace systems are highly-coupled, which means an optimizer must be scalable to large problems and robust to strong nonlinearities.

Because of all these challenges, computational aircraft design optimization is a problem where careful algorithmic choices are critical for success.

Chapter 3

AeroSandbox: A Differentiable Optimization Framework

3.1 Overview

In recognition of the challenges with aircraft design optimization described in Chapter 2, we introduce a new computational framework called *AeroSandbox* (ASB). AeroSandbox is a tool for solving design optimization problems for large, multi-disciplinary engineered systems. Examples and data structures tend to focus on aerospace problems, but the underlying numerics are generalizable to aid in the design of any complex engineered system.

AeroSandbox is a collection of interoperable tools that can be grouped into three broad categories, listed here in increasing order of abstraction:

1. **Core Tools:** An optimization framework that allows for the formulation and solution of engineering problems, and a numerics framework that enables seamless automatic differentiation for efficient gradient computation.

2. **Modeling Tools:** Tools for representing geometry (the backbone of engineering design) in an optimizer-friendly format, and tools that aid in creating custom surrogate models for user-defined physics via interpolation or fitting.

3. **Discipline-Specific Tools:** Collections of analysis tools aimed at specific aircraft design disciplines, such as aerodynamics, propulsion, and structures. These tools and are typically low- and medium-fidelity analyses (vortex lattice aerodynamics, blade-element propeller modeling, Euler-Bernoulli beam modeling, etc.) that are written from scratch to be optimizer-friendly and modular.

Each category of tools acts as an abstraction layer built upon the foundation below it. Relationships between these various tools are represented visually in Figure 3-1, which also serves as a good outline for the remainder of this work. All of these tools are described in further detail throughout Chapters 4, 5, and 6.

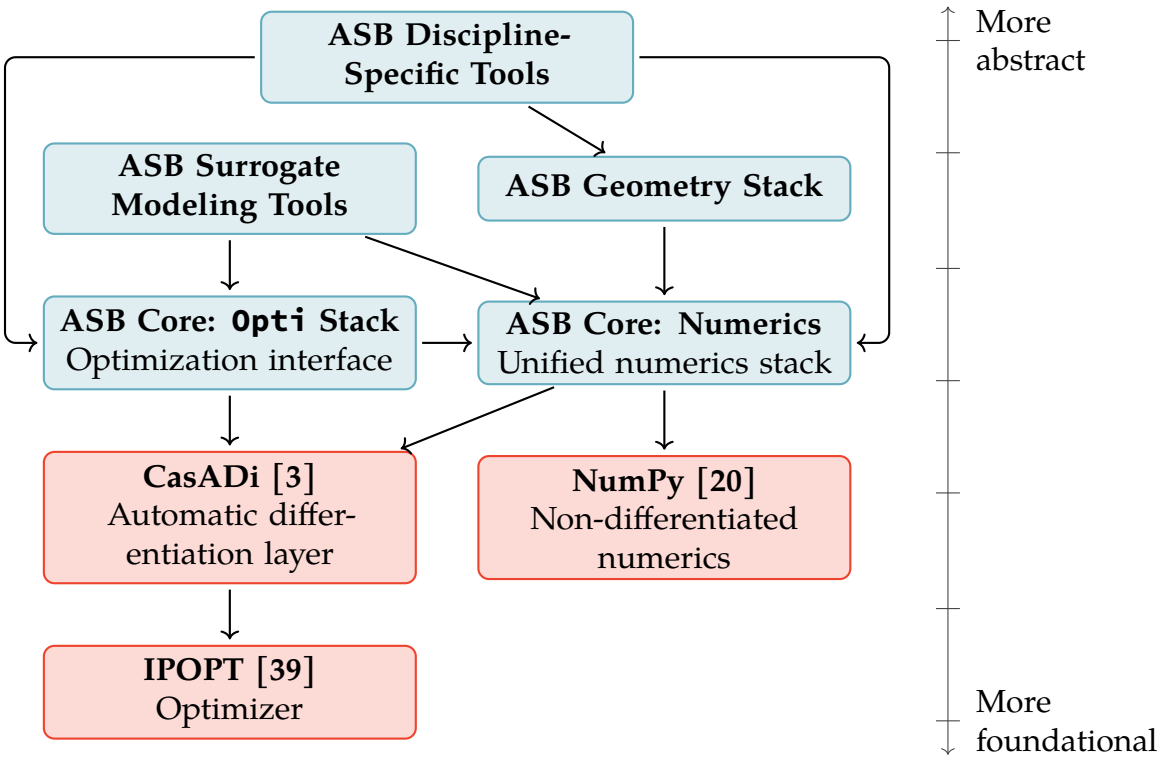

Figure 3-1: Dependency relationships between **AeroSandbox (ASB) components** and **external libraries**. Arrows point toward dependencies.

3.2 Implementation Details

3.2.1 Availability

AeroSandbox is primarily written in Python 3, the lingua franca of scientific computing at the time of writing. It is cross-platform, and distribution via the Python Package Index (PyPI) makes it a one-command installation on all platforms; full installation instructions are available in Appendix A. The source code is freely available on GitHub[1] and released under the permissive MIT License[2].

3.2.2 Documentation and Tutorials

At the time of writing, AeroSandbox is documented in several ways:

1. A tutorial[3] consisting of a series of dozens of Jupyter notebooks introduces the features and syntax of AeroSandbox. Each lesson is incremental and concise, starting with the very basics and gradually building up more complex features. This tutorial can be downloaded and viewed locally in the `/tutorials/` subdirectory, or it can be viewed online without any downloads.

2. A user guide[4] provides a comprehensive introduction to the key submodules within AeroSandbox and their relationships. This user guide can be opened at any time from the Python shell with:

```
import aerosandbox as asb
asb.docs()  # Opens the ASB documentation in a web browser.
```

3. Every class and function of AeroSandbox is documented *extensively* inline.

[1] Available at `https://github.com/peterdsharpe/AeroSandbox`

[2] The MIT License effectively allows any form of noncommercial or commercial use, modification, or redistribution.

[3] Available at `https://github.com/peterdsharpe/AeroSandbox/tree/master/tutorial`

[4] Available at `https://github.com/peterdsharpe/AeroSandbox/blob/master/aerosandbox/README.md`

At the time of writing, AeroSandbox has a comment:code ratio[5] of 0.96:1. Documentation can be produced for any AeroSandbox object using syntax analogous to:

```
import aerosandbox as asb
help(asb.Opti)  # Prints documentation for the `Opti` object.
```

3.2.3 Testing, Versioning, and Reliability

Hundreds of unit tests are written throughout AeroSandbox's code to verify correct functionality. After any code change to the AeroSandbox repository, all of these tests are automatically run[6] on a new Linux installation to ensure no unintentional breaking changes were made. This process is then repeated for the three most recent versions of the Python interpreter supported by the popular Anaconda distribution for scientific computing with Python [2].

Upon version increment, upload to distribution servers is also contingent on unit test success. Every effort is made to increment versions in accordance with the principles of *semantic versioning*, where the version number of a new release carries meaningful information about backwards-compatibility at various levels. This ensures that code can be meaningfully version-locked as needed in production environments.

Together, these features make AeroSandbox a reliable codebase for use on real problems "in the wild". This reliability had led to significant adoption, with over 160,000 downloads of AeroSandbox from PyPI at the time of writing.

[5] Ratio between number of lines. Ignores blank lines. Counts each Jupyter notebook text paragraph as a single comment line.

[6] Continuous integration provided via GitHub Actions

Chapter 4

Core Tools: Optimization and Numerics

4.1 Optimization Stack

The heart of AeroSandbox is the `Opti` stack: an object-oriented framework for formulating and solving the continuous, nonlinear, nonconvex optimization problems that occur frequently in engineering design.

The `Opti` stack is explicitly designed to be easy to learn and use for users who are not optimization specialists: design problems can be specified in natural, human-readable syntax, as demonstrated in Listing 1. Furthermore, many problem transformations (e.g. scaling heuristics, equation re-ordering, sparsity exploitation) are employed without user input in order to enhance solver speed and numerical stability on ill-posed problems.

4.1.1 Optimization Algorithms

The underlying optimization code used in AeroSandbox is IPOPT, a large-scale optimizer for general nonlinear programming written in C++ by Wächter [39]. IPOPT is a modern second-order gradient-based (quasi-Newton) filter line-search optimizer. Constraints are enforced using a primal-dual interior-point method, and

a restoration algorithm is used to allow infeasible starts and stabilize the iteration process. More details about interior-point methods for nonlinear programming are available in Nocedal and Wright [33].

A second-order gradient-based method was chosen due to the high dimensionality of aerospace design problems, as described in Section 2.2. Empirically, this leads to good performance. Lyu et al. benchmarked a suite of optimization algorithms including IPOPT on an optimization test problem as well as an engineering design optimization problem; IPOPT was found to perform competitively, even amongst other quasi-Newton algorithms such as SNOPT and SLSQP [30]. Wächter tested the IPOPT code on 954 canonical problems from the CUTEr test set and found a success rate of 93.8%, markedly above other competitive codes at the time of publication [39]. Furthermore, IPOPT is generously released under an open-source license, enabling it to be freely bundled with CasADi, and, by extension, AeroSandbox.

Yet another reason that IPOPT is an attractive choice of optimizer is that it gracefully handles sparsity in the constraint Jacobian matrix. Time-dependent engineering problems, such as trajectory optimization problems, often lead to sparsity in the constraint Jacobian; exploiting this by integrating with sparse linear algebra libraries can result in significant speed improvements.

One drawback of using IPOPT is that, unlike most interior-point optimziation algorithms, intermediate iterates do not necessarily stay feasible, even if the initial guess is feasible. This is demonstrated in Lyu et al., where the Rosenbrock example using IPOPT shows infeasible intermediate iterates[1] despite a feasible start [30]. One implication of this attribute is that physics models used in AeroSandbox should *extrapolate sensibly*: physically-plausible (even if not strictly-speaking correct) results should be returned for any design variable inputs; regardless of feasibility. This increases the likelihood that the associated gradient information at the iterate will send subsequent iterates back to the feasible space.

[1] Of course, the algorithm quickly returns to the feasible space and terminates at a point that is both feasible and optimal.

4.1.2 Automatic Differentiation for Efficient Derivatives

IPOPT, like any gradient-based optimization algorithm, relies on accurate computation of gradient (and Hessian) information in order to progress its search. Traditionally, this gradient computation has been the severe computational bottleneck limiting performance of large-scale gradient-based optimization.

Over the years, many strategies have been developed in order to compute this gradient. AeroSandbox employs a technique called *automatic differentiation* (AD) that allows highly efficient, accurate, and scalable gradient computation, implemented through the CasADi differentiation library [3]. Here, we describe automatic differentiation as it is used in AeroSandbox and how this method compares to other gradient computation techniques.

It is instructive to first examine several other popular gradient computation methods to clarify what automatic differentiation is not:

Finite-Differencing (Numerical Differentiation)

One of the more intuitive methods of gradient computation comes from the limit definition of the derivative. First considering a scalar-valued function $f(x) : \mathbb{R}^1 \to \mathbb{R}^1$, we note that:

$$\frac{\partial f}{\partial x}\Big|_{x=x_0} = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h} \tag{4.1}$$

For numerical computation of the derivative, some small but nonzero h value is used, providing an approximation to the local derivative. This idea can also be generalized to functions with multiple inputs by repeated application of this method. Specifically, if $\hat{e}_i$ represents the unit vector in the i^{th} direction, and we take some function $f(\vec{x}) : \mathbb{R}^n \to \mathbb{R}^1$:

$$\frac{\partial f}{\partial x_i}\Big|_{\vec{x}=\vec{x}_0} \approx \frac{f(\vec{x}_0 + h\hat{e}_i) - f(\vec{x}_0)}{h} \tag{4.2}$$

This calculation is then repeated for each of the n directions, at which point a

gradient vector can be constructed:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \cdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \tag{4.3}$$

The finite-difference method has the significant advantage that $f(\vec{x})$ can be any *black-box function*; that is, no information needs to be known about the inner workings of the function in order to compute a derivative with this method. This makes finite-differencing an excellent "last-resort" method when other, more sophisticated methods fail. However, there are also several catastrophic drawbacks with finite-differencing.

The most significant drawback is that finite differencing scales extremely poorly to gradient computation of high-dimensional functions, such as those that often occur in aircraft design. For a function $f(\vec{x}) : \mathbb{R}^n \rightarrow \mathbb{R}^1$, gradient computation requires n evaluations of Equation 4.2. Of course, the $f(\vec{x}_0)$ term can be reused after the first evaluation, but the $f(\vec{x}_0 + h\hat{e}_i)$ term must be evaluated n times. Thus, every gradient computation requires $n + 1$ evaluations of the function f, leading to $\mathcal{O}(n)$ time scaling.

This problem is exacerbated for higher derivatives, such as the local Hessian of $f(\vec{x})$ at $\vec{x}_0$. Computation of the Hessian requires $\mathcal{O}(n^2)$ function evaluations, which is even more intractable for large n.

A second drawback of gradient computation via finite differences is that it leads to inaccurate gradient computation. Following Equation 4.2, the gradient approximation clearly becomes more accurate as h is decreased (i.e. the truncation error is reduced). However, in the small-h limit, computational evaluation of this equation requires the subtraction of two numbers that are infinitesimally close. This leads to large floating-point error that dwarfs the truncation error. Thus, there is an optimal step size h that balances these two error sources and leads to minimum error. If we consider computation of a first-order forward difference (as in Equation 4.2)

using double-precision floating-point arithmetic with precision[2] $\epsilon = 2^{-53} \approx 10^{-16}$, this optimal h is roughly[3] $\mathcal{O}(10^{-8})$. More generally, first-order finite differences with optimal step size generally have an accuracy of $\mathcal{O}(\sqrt{\epsilon})$. This phenomenon is illustrated graphically in [31].

One might try to increase gradient precision by using a second-order difference scheme, such as the central difference method described in Equation 4.4:

$$\left.\frac{\partial f}{\partial x_i}\right|_{\vec{x}=\vec{x}_0} \approx \frac{f(\vec{x}_0 + h\hat{e}_i) - f(\vec{x}_0 - h\hat{e}_i)}{2h} \tag{4.4}$$

However, this has the unfortunate side-effect of exacerbating the first drawback of finite-difference methods; namely, gradient computation in n dimensions now requires $2n$ function evaluations. The accuracy improvement is also mediocre; although the optimal step size drops, gradient accuracy remains only at $\mathcal{O}(\epsilon^{2/3})$, or $\approx \mathcal{O}(10^{-10})$ for double-precision arithmetic.

Complex-Step Differentiation

A modification of the finite-difference approach that attempts to alleviate the floating-point arithmetic problem is the complex-step method. The complex-step method works on any complex-differentiable (also known as *analytic* or *holomorphic*) function. Fortunately, most mathematical operators used in engineering design (e.g. elementary functions) are analytic, and compositions of analytic functions are also analytic. For any such analytic function, the Cauchy-Riemann equations hold. For the purposes of complex-step differentiation, only the first Cauchy-Riemann equation is of particular importance; it is listed in Equation 4.5:

$$\begin{aligned} &\text{For the complex decomposition } f(x+iy) = u(x,y) + iv(x,y), \\ &\qquad\qquad \frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} \end{aligned} \tag{4.5}$$

[2] defined here as the difference between 1 and the next-largest representable number
[3] for an example function with a Hessian locally equal to the identity matrix

Noting that $\frac{\partial u}{\partial x} = \frac{\partial f}{\partial x}$ holds for points on the real line[4] (which are the points of interest in engineering design), we can rewrite a forward-difference derivative rule as:

$$\frac{\partial f}{\partial x}\Big|_{x=x_0} \approx \lim_{h \to 0} \frac{f(x_0 + ih) - f(x_0)}{ih} \tag{4.6}$$

Or, after simplification (noting that for practical f, $f(x) \in \mathbb{R}$ if $x \in \mathbb{R}$):

$$\frac{\partial f}{\partial x}\Big|_{x=x_0} \approx \lim_{h \to 0} \frac{\text{Im}\left(f(x_0 + ih)\right)}{h} \tag{4.7}$$

This complex-step differentiation has several advantages over finite-differencing. First, it is higher-order - a complex-step forward-difference has $\mathcal{O}(h^2)$ truncation error, as opposed to the $\mathcal{O}(h)$ error seen with a real-step forward-difference. However, more importantly, floating-point error does not occur as there is no subtractive cancellation in the derivative computation. This means that an arbitrarily small step size h can be taken, invariably leading to a gradient error at machine precision (i.e. $\mathcal{O}(\epsilon)$).

However, some problems of finite-differencing remain. Most prominently, gradient computation for functions of n dimensions still requires $\mathcal{O}(n)$ function evaluations, which is unacceptably slow for large problems.

On the other hand, complex-step differentiation introduces several new disadvantages.

First, there are very few situations where use of the complex-step derivative is the optimal choice: it requires the bizarre scenario of a black-box function that happens to already support complex math with proper analytic continuation. This is almost never the case, as engineering tools in aerospace[5] rarely expect complex inputs, and most black-box analysis tools are written in statically-typed languages. In statically-typed languages (e.g. C++, C, Fortran), enabling complex math requires editing the source code, something that is not possible for black-box functions. In dynamically-

[4] for functions of engineering design interest

[5] barring special cases such as signal analysis, controls, or radar-cross-section computation

typed languages (e.g. Python, Julia, MATLAB), complex-step differentiation can usually be performed easily enough, although operator overloading with dual numbers[6] is more accurate and generally much more computationally efficient.

Secondly, adding complex math support often results in a significant computational bottleneck, as compilers and libraries are often not optimized for high-performance complex math, instead focusing on real floating-point arithmetic.

Thirdly, no clear extension exists to use complex-step differentiation to compute higher-order derivatives, such as the Hessian. The Cauchy-Riemann equations only constrain on the first partial derivative, so different or extended techniques are needed for higher-order derivatives.

Finally, the claimed precision advantage of complex-step over finite-differencing is misleading, as switching from real to complex numbers typically doubles the number of stored bits for a floating-point number. Consider the following illustrative example:

> A real double-precision floating-point arithmetic number ("double") is represented by 64 bits[a]. A real forward finite difference with optimal step size produces an error of size $\mathcal{O}(\sqrt{\epsilon}) \approx \mathcal{O}(10^{-8})$.
>
> A complex double is represented by 128 bits (64 for the real part, and 64 for the imaginary part). A complex-step derivative produces error of size $\mathcal{O}(\epsilon) \approx \mathcal{O}(10^{-16})$. It is more accurate, but computation is slower because all variables take up twice memory as before.
>
> A fairer comparison is to examine a quad-precision floating-point arithmetic number, which is also represented by 128 bits. A real forward finite difference here produces an error again of size $\mathcal{O}(\sqrt{\epsilon})$, although since ϵ has shrunk to $\approx 10^{-34}$, this corresponds to an error of $\mathcal{O}(10^{-17})$.
>
> [a] as described in the IEEE 754 standard

So, a complex-step derivative does not produce any additional derivative precision over finite-difference when variable bit count (and accordingly, computational

[6] effectively a basic implementation of *forward-mode automatic differentiation*, described later

speed) is controlled for. Therefore, the major claimed benefit of complex-step differentiation is lost, and this method is largely a mathematical curiosity with little practical benefit.

Symbolic Differentiation

Symbolic differentiation is the process of computing an explicit derivative (either by hand or with the aid of a computational symbolics engine) and hard-coding that as the derivative to a function of interest. For example:

$$\begin{aligned} \text{Function of interest: } L(\rho, V, C_L, S) &= \frac{1}{2}\rho V^2 C_L S \\ \text{Gradient: } \nabla L &= \begin{bmatrix} \frac{\partial L}{\partial \rho} \\ \dots \\ \frac{\partial L}{\partial S} \end{bmatrix} = \begin{bmatrix} \frac{1}{2}V^2 C_L S \\ \dots \\ \frac{1}{2}\rho V^2 C_L \end{bmatrix} \end{aligned} \tag{4.8}$$

These gradients can then be directly coded to be used as needed. An advantage of this approach is that gradients can generally be computed at machine precision.

However, the $\mathcal{O}(n)$ scaling problem still exists, as a gradient computation requires the evaluation of n partial derivatives.

Furthermore, symbolic differentiation suffers from "expression swell". Examination of any elementary differentiation table in a calculus table will reveal that derivatives of functions tend to be longer than the functions themselves. Therefore, explicit representations of gradients of complicated functions tend to grow to massive length, rendering them intractably large to manage and evaluate.

Closer examination of this expression swell reveals another interesting point, however: often, many of the subexpressions within the gradient equation are repeated. The reason for this is best shown by example: if we consider the product rule $\frac{\mathrm{d}[a(x)\cdot b(x)]}{\mathrm{d}x} = a'(x)b(x) + a(x)b'(x)$ and the fact that $a(x)$ and $a'(x)$ will naturally share many subexpressions, it follows that a symbolic derivative will tend to have many repeated subexpressions.

As it turns out, we can alleviate the problem of expression swell entirely by re-using intermediate evaluations of subexpressions. In practice, this means relaxing the requirement that all steps of the derivative computation be "flattened" into a single, explicit representation. As a consequence, our gradient is no longer represented by an *equation* that can be evaluated, but rather by a *procedure* that can be followed in order to evaluate the gradient at a given point: our gradient is a *code function* rather than a *mathematical function*. This approach happens to be identical to "forward-mode automatic differentiation", which is covered in the following section.

Automatic Differentiation

Automatic differentiation (also known as *autodiff*, AD, *backpropagation*, or *algorithmic differentiation*) is a method of computing derivatives that was pioneered in the machine learning (specifically, neural network) and optimal control communities.

AD works by directly examining the source code of the function being differentiated. It exploits the fact that any computational function, no matter how complex, can be broken down into compositions of a small set of elementary[7] functions. These functions can then be differentiated individually using known derivative rules and chained together using the chain rule.

This decomposition of a function of interest into its individual operations leads to the idea of a *computational graph*, which depicts the flow of information from through a function in code. This graph is also known as a *trace* in some literature, which is an apt term as it allows one to trace a computer's path through a function. An example of such a graph is depicted in Figure 4-1.

[7]Paraphrasing Liouville, this includes arithmetic, trigonometric, power/logarithmic, and hyperbolic functions, along with inverses thereof.

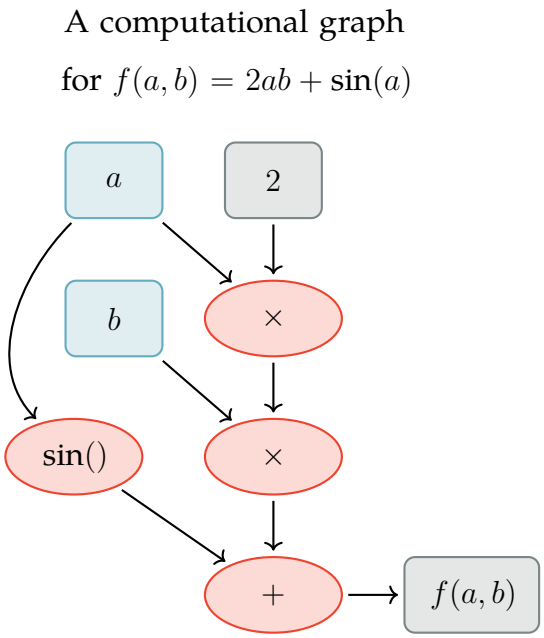

Figure 4-1: A computational graph, as would be used for automatic differentiation.

Computing a function value given inputs is straightforward using this computational graph; one supplies the known inputs and progressively works through the graph to produce an answer.

Computing a derivative given inputs using this method is just as simple. To compute $\frac{\partial f}{\partial a}$ in the example of Figure 4-1, one first evaluates derivative with respect to inputs: $\frac{\partial a}{\partial a} = 1$, $\frac{\partial b}{\partial a} = 0$, and so on. Then, each function is evaluated using its respective derivative rule (e.g. the product rule), and the result is $\frac{\partial f}{\partial a}$.

The procedure just described is termed "forward-mode automatic differentiation", as the derivative is propagated *forward* through the computational graph. Forward-mode AD computes the gradient of a function $\vec{f}(\vec{x}) : \mathbb{R}^m \to \mathbb{R}^n$ in $\mathcal{O}(m)$ time complexity; that is, the cost is proportional to the number of function inputs.

An alternative and far more powerful method is *reverse-mode* automatic differentiation. Here, one computes a derivative by working *backwards* through the computational graph. Because we are working backwards, this leads to some non-intuitive questions: for example, for each constituent function we must now ask "how do the inputs of this function change with respect to the output?". This is

effectively a function-by-function adjoint approach, which are strung together via the computational graph to produce a derivative.

The performance benefit that can be realized in exchange for the non-intuitive nature of reverse-mode AD is massive. Reverse-mode AD computes the gradient of a function $\vec{f}(\vec{x}) : \mathbb{R}^m \to \mathbb{R}^n$ in $\mathcal{O}(n)$ time complexity; that is, the cost is proportional to the number of function *outputs*. Specifically, Griewank proved that the cost of gradient computation is at most five times the cost of evaluating a scalar function, independent of the number of inputs [18].

Reverse-mode AD is far more useful than forward-mode in practice, as most functions $\vec{f}(\vec{x}) : \mathbb{R}^m \to \mathbb{R}^n$ of practical interest have $n \leq m$ (and often, $n = 1$); that is, most engineering functions map from a high-dimensional space to a low-dimensional one. An example of this is in aerodynamic analysis, where a CFD run might have millions of inputs (e.g. mesh node locations) but only a few outputs of interest (e.g. net lift and drag). It makes intuitive sense that $n \leq m$ typically: optimization problems have many variables but generally just one objective. Furthermore, if this were not the case, the input space could not span the output space[8], so the model would not yield any additional independent information[9].

Automatic differentiation has several other key advantages over other gradient computation techniques. One of these is that the space of elementary functions is closed under differentiation; that is, the derivative of each function is exclusively a composition other (differentiable) elementary functions. Therefore, automatic differentiation allows one to compute arbitrarily high order derivatives (e.g. the Hessian, which is critical for optimization) using the same framework [6].

AD is also capable of following arbitrarily complex control flow like loops, conditional statements, object-oriented data structures, and recursion. This means that nearly any scientific analysis can be differentiated using AD, even in cases where a symbolic gradient would be nearly impossible to write.

Reverse-mode automatic differentiation therefore represents a method to com-

[8] barring functions like space-filling curves that do not typically appear in models of physics phenomena

[9] as measured by the number of independent degrees of freedom

pute gradients of scalar functions to machine precision with $\mathcal{O}(1)$ complexity. This revolutionary improvement over other gradient methods enables efficient scaling to arbitrarily-large design optimization problems.

Because of these reasons, automatic differentiation[10] is the gradient computation method employed by AeroSandbox. This automatic differentiation is provided by CasADi, an open-source tool for automatic differentiation and nonlinear optimization developed by members of the optimal control community [3].

4.2 Syntax and Mathematical Examples

In order to solve a problem with AeroSandbox, one first creates an empty optimization environment by instantiating this `Opti` class. After creating this class instance, one can declare variables, constraints, and parameters in arbitrary order. Upon calling the `Opti.solve()` method, these problem elements are automatically differentiated as-needed, constructed into a standard-form nonlinear program, solved via an interface to IPOPT, and returned to the user.

Despite the sophisticated inner workings of AeroSandbox's `Opti` stack, an extensive effort has been made to abstract these details away from the user unless requested; sensible defaults and heuristics are used throughout AeroSandbox to make usage as easy as possible. For advanced users, it is useful to note that AeroSandbox's `Opti` class inherits and extends the corresponding `Opti` class from CasADi; more sophisticated optimization settings can be accessed directly through syntax shared with CasADi as needed.

4.2.1 Example: Constrained Rosenbrock Problem

Here, we illustrate the simplicity of the `Opti` stack by formulating the constrained Rosenbrock problem [37], a common test problem for gradient-based optimization frameworks. The canonical version of the constrained Rosenbrock problem is as follows:

[10] primarily reverse-mode, but also forward-mode for certain computations

$$\begin{aligned} \underset{x,\,y}{\text{minimize}} \quad & (1-x)^2 + 100(y-x^2)^2 \\ \text{subject to} \quad & x^2 + y^2 \leq 1 \end{aligned} \tag{4.9}$$

The objective function landscape, as shown in Figure 4-2, is a shallow, curved valley that is constrained to the unit disk. A global minimum exists at $(x, y) \approx (0.7864, 0.6177)$; an analytical derivation of this is shown in Appendix C.1.

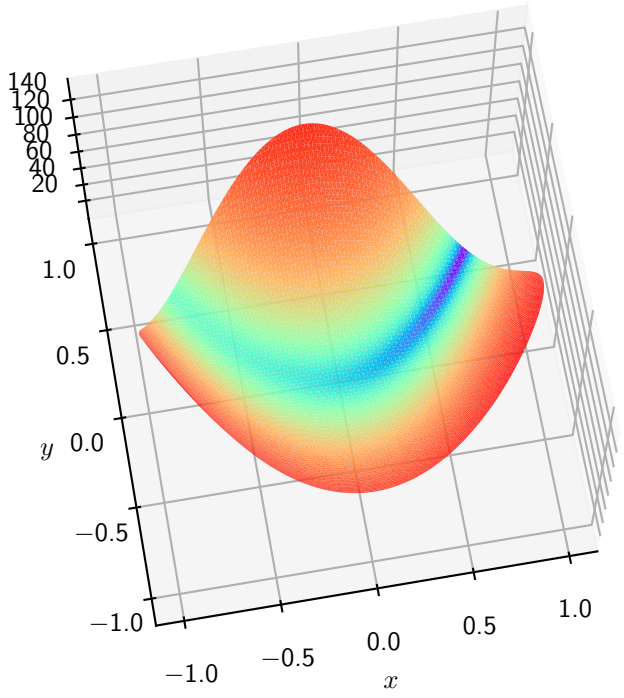

Figure 4-2: Constrained Rosenbrock Function

This problem is a particularly useful example in the context of engineering design optimization, as it has many mathematical qualities which tend to be relevant in engineering problems. Specifically, both the Rosenbrock problem in Eq. 4.9 and many engineering problems are:

1. **Continuous**: Design variables are not constrained to be integer, drawn from an enumeration, or otherwise discrete.

2. **Constrained**: Constraints, typically given by physics or requirements, limit the allowable design space.

3. **Nonlinear**: Both the objective function and all constraints are, in general, nonlinear functions of design variables.

4. **Nonconvex**: The objective function and constraints are not necessarily convex functions of the design variables. In formal terms, the these functions need not satisfy the convex inequality; in informal terms, we effectively make no restrictions on the curvature of these functions. In the case of the Rosenbrock problem, the point $(0, 1)$ is locally nonconvex, with objective Hessian eigenvalues (principal curvatures) of 200 and -398.

5. **Unimodal**: Allowing nonconvex objective functions admits multimodal[11] problems, which raises the concern of global optimality. However, in practice, enough nonconvex aerospace engineering problems are indeed unimodal that current best practices are to assume unimodality until proven otherwise [31, p. 212]. Indeed, the Rosenbrock problem is unimodal despite its nonconvexity.

6. **Poorly-scaled**: At regions of interest in the design space (e.g. initial guess, optimal solution, intermediate points), the Hessian matrix of the Lagrangian (or here, the objective) has a large condition number. In the case of the constrained Rosenbrock problem, $\text{cond}(H) \approx 1054$ at the optimum.

   Poor scaling typically occurs in engineering problems because of differences in orders of magnitude and units between quantities of interest. For example, a wing skin thickness might measure one millimeter, while mission range could be on the order of hundreds of kilometers. Simple linear problem scaling (discussed later) can alleviate this issue to some degree, but it is practically impossible to entirely eliminate scale differences given *a priori* problem knowledge. Furthermore, because the key metric for solver performance here is the Hessian condition number, relying on this approach alone requires an

[11] Definition: having more than one local optima

engineer to accurately estimate the *curvature* of the performance function - a markedly more difficult task than simply guessing the scale of a quantity.[12]

The Rosenbrock problem can be solved in AeroSandbox using the simple Python syntax in Listing 1.

```python
def rosenbrock(x, y):
    return (1 - x) ** 2 + 100 * (y - x ** 2) ** 2

import aerosandbox as asb  # Import the AeroSandbox library.

opti = asb.Opti()  # Create a new optimization environment.

x = opti.variable(init_guess=4)  # Define optimization variables.
y = opti.variable(init_guess=4)

opti.subject_to(x ** 2 + y ** 2 <= 1)  # Define a constraint.

opti.minimize(rosenbrock(x, y))  # Define an objective.

sol = opti.solve()  # Solve the problem.

x_opt, y_opt = sol.value(x), sol.value(y)  # Extract the solution.
print(x_opt, y_opt)  # Display the solution.
```

Listing 1: AeroSandbox solution of the constrained Rosenbrock problem.

Initializing from a guess of $(x, y) = (4, 4)$, this code retrieves the correct solution for the optimum of $(x, y) = (0.7864, 0.6177)$ after 9 iterations.

A point of particular note here is that our initial guess of $(x, y) = (4, 4)$ lies outside the feasible design space. Unlike some interior-point optimization algorithms, IPOPT does not require a feasible initial guess. This is a particularly important feature in aircraft design, where the high number of cross-discipline constraints often makes *a priori* identification of a feasible point laborious.

[12]This technique also requires domain-specific expert knowledge, which may not be available. Thus, an optimizer that gracefully handles poorly-scaled problems is always desirable.

4.2.2 Example: 5,000-Dimensional Rosenbrock Problem

In the previous example, we solved the constrained Rosenbrock problem. This was a 2-dimensional problem, so we individually created two variables: x and y.

For high-dimensional problems, it is both more concise and more computationally-efficient to vectorize variables. A vector of variables is a powerful construct, as it allows vector and matrix operations to be used with identical notation to a standard mathematical optimization formulation.

This can be demonstrated by finding the minimum of the n-dimensional Rosenbrock problem, as given by Virtanen et al. [38]:

$$\underset{\vec{x}}{\text{minimize}} \quad \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \tag{4.10}$$

For any n except $4 \leq n \leq 7$, this function is unimodal [29]. Here, we take $n = 5000$ in order to demonstrate the scalability of automatic differentiation (as described in Sect. 4.1.2) and variable vectorization. This results in a minimum at $\vec{x} = [1, 1, \ldots, 1]$, corresponding to a function value of 0.

This problem can be coded in AeroSandbox as shown in Listing 2.

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

N = 5000

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

x = opti.variable(init_guess=4 * np.ones(shape=N))  # Create a vectorized variable x.

x1 = x[:-1]  #  Effectively x_i
x2 = x[1:]   #  Effectively x_{i+1}

f = np.sum(100 * (x2 - x1 ** 2) ** 2 + (1 - x1) ** 2)

opti.minimize(f)  # Define an objective.

sol = opti.solve()  # Solve the problem.

x_opt = sol.value(x)  # Extract the solution.
```

Listing 2: AeroSandbox solution of the 5,000-dimensional Rosenbrock problem.

The correct solution of $\vec{x} = [1, 1, \dots 1]$ is found after just 25 iterations. Querying this solution further, we find that our objective function was evaluated 49 times[13]. It is instructive to consider that we have effectively found the zero of the 5000-dimensional function $\frac{\partial f}{\partial \vec{x}}$ with only 49 function evaluations, a feat that initially seems to violate some arcane principle of information theory. This apparent paradox is resolved by noticing that the gradient evaluation via reverse-mode automatic differentiation contributes 5000 pieces of information for each pass through the computational graph. This empirically demonstrates why efficient gradient computation is the key to fast optimization: each pass through the computational graph gives us orders of magnitude more useful information than a single function evaluation.

We make particular note of the *speed* of solving this problem, recalling that the problem is a nonlinear, nonconvex optimization problem of 5000 variables. Despite

[13]More than one function evaluation may occur per iteration as a result of the line search that occurs at each iteration.

this, on a single workstation laptop[14], AeroSandbox solves this problem in just 0.284 seconds.

We can quantify this scaling by solving this same n-dimensional problem for various n, as shown in Figure 4-3. We notice here that the asymptotic time complexity is (empirically) roughly $\mathcal{O}(n^{0.9})$, indicating sublinear scaling with respect to dimensionality. We can identify several factors that contribute to the sublinear scaling curve seen in Figure 4-3:

- The cost of a single objective function evaluation scales as $\mathcal{O}(n)$ due to the length of the vectors being operated on.

- The cost of a single gradient evaluation is proportional to the cost of a function evaluation regardless of n, thanks to reverse-mode automatic differentiation. This means that gradient evaluation is also $\mathcal{O}(n)$. (With a finite-difference method, this would be $\mathcal{O}(n^2)$.)

- The number of iterations required for convergence is roughly independent of the problem dimensionality.

- A small overhead cost (here, ≈ 14 milliseconds) is associated with interfacing with IPOPT. For repeated optimization runs (as in the case of a parametric design study), this cost can be eliminated as described in Section **??**.

[14]Intel i7-8750H CPU, Windows 10

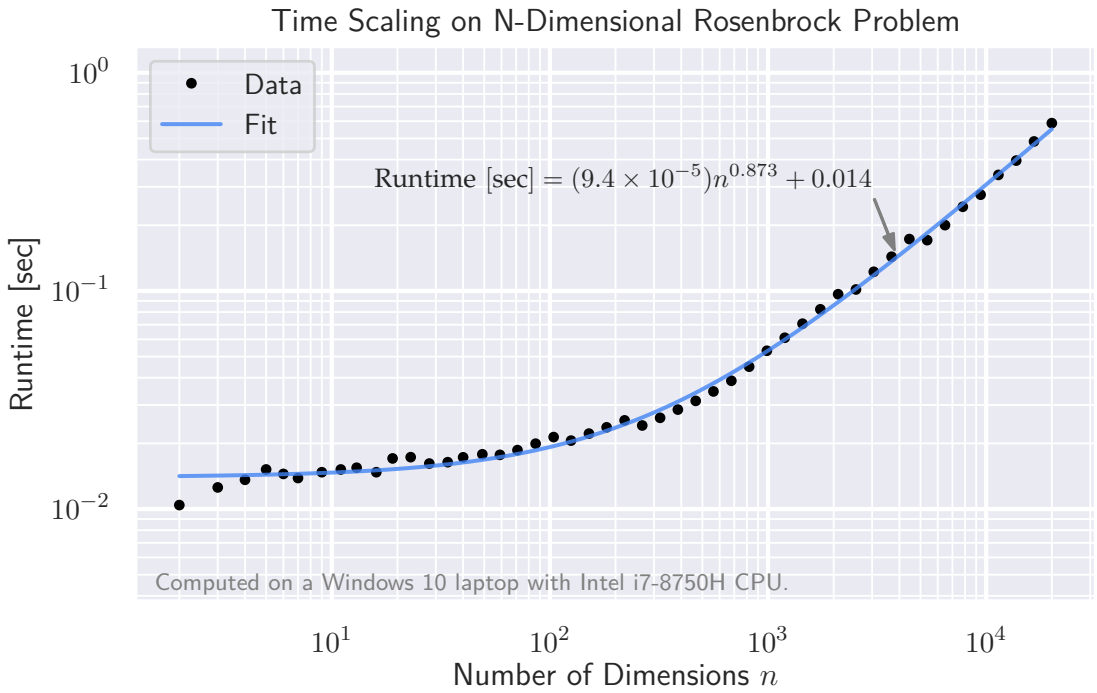

Figure 4-3: N-Dimensional Rosenbrock Performance.

4.3 Simple Aircraft Design Examples

In order to demonstrate that this high-performance optimization framework is not limited to mathematical examples, we present several aircraft design test problems. These also serve as examples to benchmark against other comparable frameworks.

4.3.1 Example: Simple Wing

A common aerospace design problem is wing drag minimization. Here, we implement one such problem in AeroSandbox and demonstrate its solution.

The problem at hand is identical to the one specified in Section III of Hoburg and Abbeel [23], with a few constants tweaked to match the formulation in [27], [34], and other related works. It is restated here in its natural engineering syntax:

Simple Wing

$$\begin{aligned} \underset{Æ, S, V, W, C_L}{\text{minimize}} \quad & D \\ \text{subject to} \quad & W \leq L_{\text{cruise}}, \\ & W \leq L_{\text{takeoff}}, \\ & W = W_{\text{fuselage}} + W_{\text{wing}} \end{aligned} \tag{4.11}$$

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

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 $W_{\text{wing}} = W_{\text{w, structural}} + W_{\text{w, surface}}$
  - $W_{\text{w, structural}} = W_{\text{w, c1}} \cdot \frac{N Æ^{1.5} \sqrt{W_0 W S}}{\tau}$
  - $W_{\text{w, surface}} = W_{\text{w, c2}} \cdot S$

where: k = 1.2, the form factor.

e = 0.95, the Oswald efficiency factor.

μ = 1.78×10^{-5} kg m^{-1} 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 = 3.8, the ultimate load factor.

V_{min} = 22 m/s, the takeoff airspeed.

$C_{L,\text{max}}$ = 1.5, the takeoff lift coefficient.

S_{wet}/S = 2.05, the wetted area ratio.

CDA_0 = 0.031 m^2, the fuselage drag area.

W_0 = 4940 N, the aircraft weight excluding the wing.

$W_{\text{w, c1}}$ = 8.71×10^{-5} m^{-1}, a coefficient used to calculate wing weight.

$W_{\text{w, c2}}$ = 45.24 Pa, another wing weight coefficient.

This problem can be solved in AeroSandbox using the code shown in Appendix C.2. AeroSandbox requires initial guesses, which are chosen as rounded order-of-magnitude estimates following Kirschen [27][15]. Solution via the code in Appendix C.2 produces the following result, which is identical to that found in [23]:

Table 4.1: Solution of Simple Wing (Eq. 4.11), found with AeroSandbox.

Figure of Merit	Optimal Value
Minimum drag D	303.07 N
Aspect ratio $\mathcal{AR}$	8.46
Wing area S	16.44 m^2
Airspeed V	38.15 m/s
Weight W	7341 N
Lift Coefficient C_L	0.4988

This problem, which consists of 5 design variables and 3 constraints, is solved predictably quickly from our initial guess in just 11 milliseconds and 25 iterations.

[15] Specific figures for initial guesses can be viewed in the code in Appendix C.2.

Comparison with solves in [34] and [27] confirms that this is the global optimum; this can also be proven analytically.

On Initial Guesses and Solver Robustness

A common concern for any optimizer when working with practical problems is its robustness to bad initial guesses. The optimization algorithms in AeroSandbox are much more robust to poor initial guesses than other nonlinear program (NLP) optimizers.

For example, Kirschen and Hoburg found that various general NLP solvers performed exceptionally poorly on this same Simple Wing problem [27]. Kirschen gave this problem with the exact same initial guess to commercial interior-point and SQP optimizers built into MATLAB's `fmincon()` solver. Neither optimizer was able to successfully converge to the solution, while AeroSandbox was; a comparison is presented in Table 4.2. In addition, the numbers from Kirschen in Table 4.2 correspond to IP and SQP runs with exact analytical gradients[16], so the stark performance difference between these methods and AeroSandbox is purely due to differences in optimization algorithms, not gradient quality.

In Table 4.2, we also benchmark AeroSandbox against GPKit, a specialized framework for solving geometric programs such as Simple Wing. GPKit is benchmarked here with its open-source `cvxopt` backend in order to facilitate a fair "open-source vs. open-source" comparison. Despite the fact that GPKit is a specialized tool for this particular class of problem, the general-purpose AeroSandbox solver consistently achieves convergence faster on the same hardware.

[16] Runs in Kirschen [27] without exact analytical gradients, i.e. finite-difference gradients, perform even worse.

Table 4.2: Comparison of optimization methods: Simple Wing problem with same initial guesses.

Optimizer	Optimality		Speed	
	Drag (Objective)	Optimal?	Runtime	Func. Evals.
IP, Kirschen [27]	593.76 N		37.7 s [a]	10,530
SQP, Kirschen [27]	438.66 N		0.1 s [a]	83
GPKit	303.07 N	✓	0.026 s [b]	-
AeroSandbox	303.07 N	✓	0.022 s [b]	29

[a] Tested by Kirschen [27] on unspecified hardware.

[b] Both tested on an Intel i7-8750H CPU Windows 10 laptop; median over 10 runs.

Runtime comparisons should be made within groups [a] and [b], not across them.

We can demonstrate the robustness of AeroSandbox further by modifying the initial guesses to more extreme values. Here, we arbitrarily select the airspeed V, which is nominally initialized to a value of 100 m/s. We can change this initial guess by two orders of magnitude in either direction (to 1 m/s and $10,000$ m/s); in both cases, the problem still solves to optimality in 32 iterations or less, and runtime is essentially unaffected.

Although this robustness is obviously problem-dependent in a rigorous sense, in practice we find that this robustness to initial guesses holds consistently. As long as variables are initialized to approximately within two orders of magnitude in either direction of their optimal value, convergence is likely. Guessing a value within this 4-order-of-magnitude window is generally trivial based on engineering intuition.

This extreme robustness is observed in part due to the robustness of the IPOPT algorithm - in particular its line search and restoration phases. This robustness is also due to a series of heuristic problem transformations that are carried out automatically in AeroSandbox; these are detailed further in Section 4.3.2.

4.3.2 Problem Transformations

A variety of heuristic problem transformations are automatically performed by AeroSandbox in an attempt to make problems more amenable to a general-purpose NLP solver. These heuristic transformations can dramatically improve solver performance and can also be manually specified by advanced users. Here, we discuss a few of these heuristics.

Problem Scaling

When formulating optimization problems to be passed to NLP solvers, it is critical to scale design variables so that the problem is *well-scaled*. Here, "scaling" means that a variable is multiplied or divided by some constant before it is passed into the optimizer, so that the optimizer sees a value that is ideally on the order of 10^{-2} to 10^{2}.

The primary step of the IPOPT solver within AeroSandbox is a quasi-Newton line-search direction calculation, which is theoretically scale-invariant. However, due to floating-point arithmetic precision limits, scaling can significantly affect conditioning of the Lagrangian Hessian and constraints Jacobian matrices. Furthermore, IPOPT's interior-point barrier algorithm is not scale-invariant, so scaling is critical on constrained problems (which represent the vast majority of cases).

The issue of scaling is especially important in engineering design optimization, where problems frequently have variables that span many orders of magnitude if left unscaled. For example, a composite skin thickness might be on the order of 10^{-3} m, while an internal stress might be on the order of 10^{8} Pa.

In AeroSandbox, the scale factor for a given variable can be easily set at initialization using the following inline syntax:

```
import aerosandbox as asb

opti = asb.Opti()
x = opti.variable(init_guess = 5, scale = 10)
```

Similarly, the scale of vector variables can be set with either a constant or a vector of equal length (in which case scales are applied element-wise). Because scaling is so critically important, AeroSandbox will automatically scale all variables using heuristics based on the initial guess.

We can illustrate the importance of appropriate problem scaling and some of AeroSandbox's heuristics by example. Here, we recreate a NLP scaling example problem originally formulated by Durbin and the CasADi team [16]. Durbin et al. used this example to illustrate the importance of scaling, although in their study, scales were manually specified. Here, we recreate this study but perform all scaling using AeroSandbox's automatic heuristics. The problem is formulated as follows:

Rocket Optimal Control (formulated by Durbin et al. [16])
A rocket launches from the ground at time $t = 0$ and must reach an altitude $y = 100$ km at time $t = 100$ s. Dynamics are treated as 1D. We seek the trajectory that minimizes fuel use (or equivalently, maximizes final rocket mass, as the initial mass is constrained). A control vector $u(t)$ that specifies the thrust profile is optimized.

$$
\begin{aligned}
\underset{y(t), \dot{y}(t), m(t), u(t)}{\text{minimize}} \quad & -m_{\text{final}} \\
\text{subject to} \quad & \frac{\mathrm{d}y}{\mathrm{d}t} = \dot{y}, \\
& \frac{\mathrm{d}\dot{y}}{\mathrm{d}t} = u/m - (9.81\ \mathrm{m/s^2}), \\
& \frac{\mathrm{d}m}{\mathrm{d}t} = \frac{-u}{(300\ \mathrm{sec}) \cdot (9.81\ \mathrm{m/s^2})}, \\
& y(0) = 0, \\
& \dot{y}(0) = 0, \\
& m(0) = 500 \times 10^3\ \mathrm{kg}, \\
& y_{\text{final}} = 100 \times 10^3\ \mathrm{m}, \\
& m > 0, u \geq 0, y \geq 0
\end{aligned}
\tag{4.12}
$$

For example purposes and following [16], we transcribe this problem using direct collocation with forward-Euler integration across 100 points uniformly spaced in time, a technique described further in Section 4.7. The optimal trajectory after solution is given in Figure 4-4.

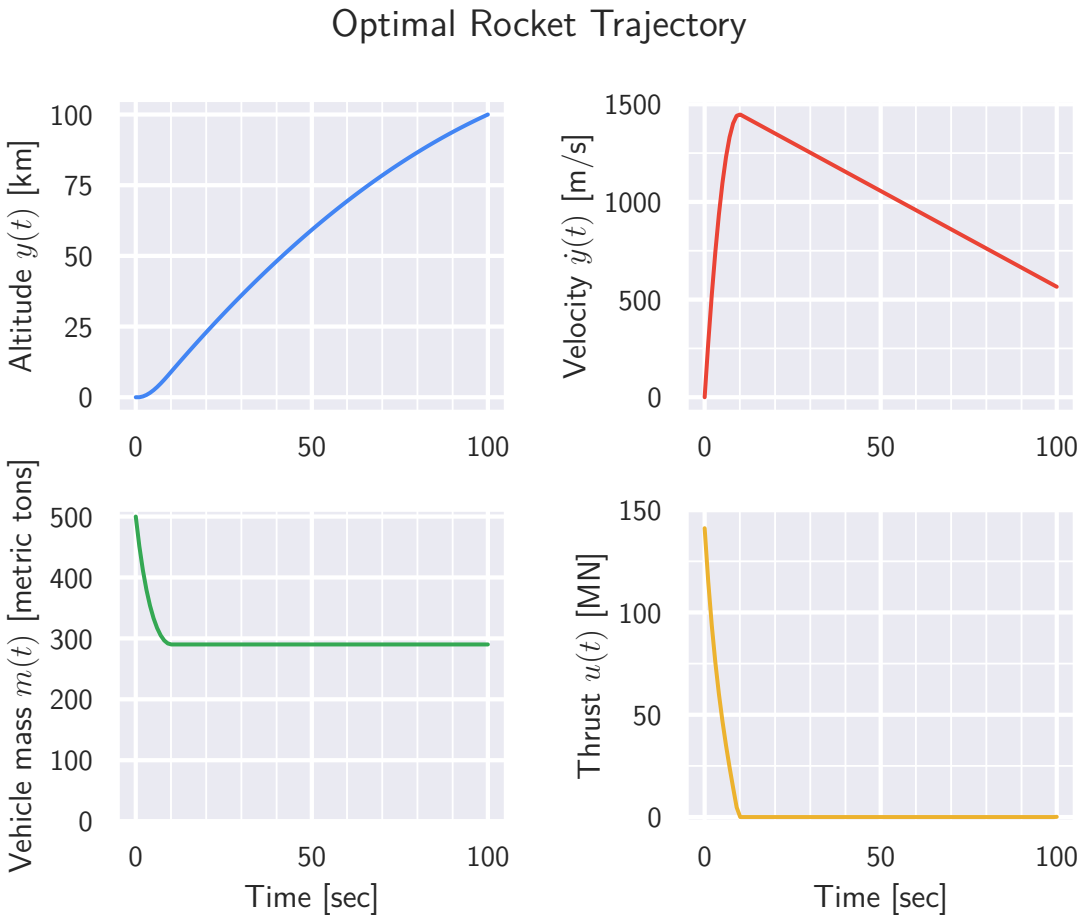

Figure 4-4: Optimal trajectory of the Rocket Optimal Control problem (Eq. 4.12), found with AeroSandbox.

We now examine the effect of scaling on the optimizer performance when solving this problem. We solve this problem once with scaling heuristics disabled[17] and a second time with scaling heuristics enabled[18]. In both cases, we provide initial guesses based on a bit of first-order kinematics intuition from the problem

[17]Implemented by passing `scale=1` at variable initialization.
[18]Enabled by default.

description:

$$
\begin{aligned}
y_{\text{guess}}(t) &= \frac{100 \times 10^3 \text{ m}}{100 \text{ s}} \cdot t \\
\dot{y}_{\text{guess}}(t) &= \frac{100 \times 10^3 \text{ m}}{100 \text{ s}} \\
m_{\text{guess}}(t) &= 500 \times 10^3 \text{ kg} \\
u_{\text{guess}}(t) &= (9.81 \text{ m/s}^2) \times (500 \times 10^3 \text{ kg})
\end{aligned}
$$

Although both the unscaled and scaled runs eventually solve the problem, the difference in solution speed is dramatic; this is illustrated in Figure 4-5. The solution with scaling heuristics is clearly much more stable, and convergence is achieved in 14 rather than 85 iterations.

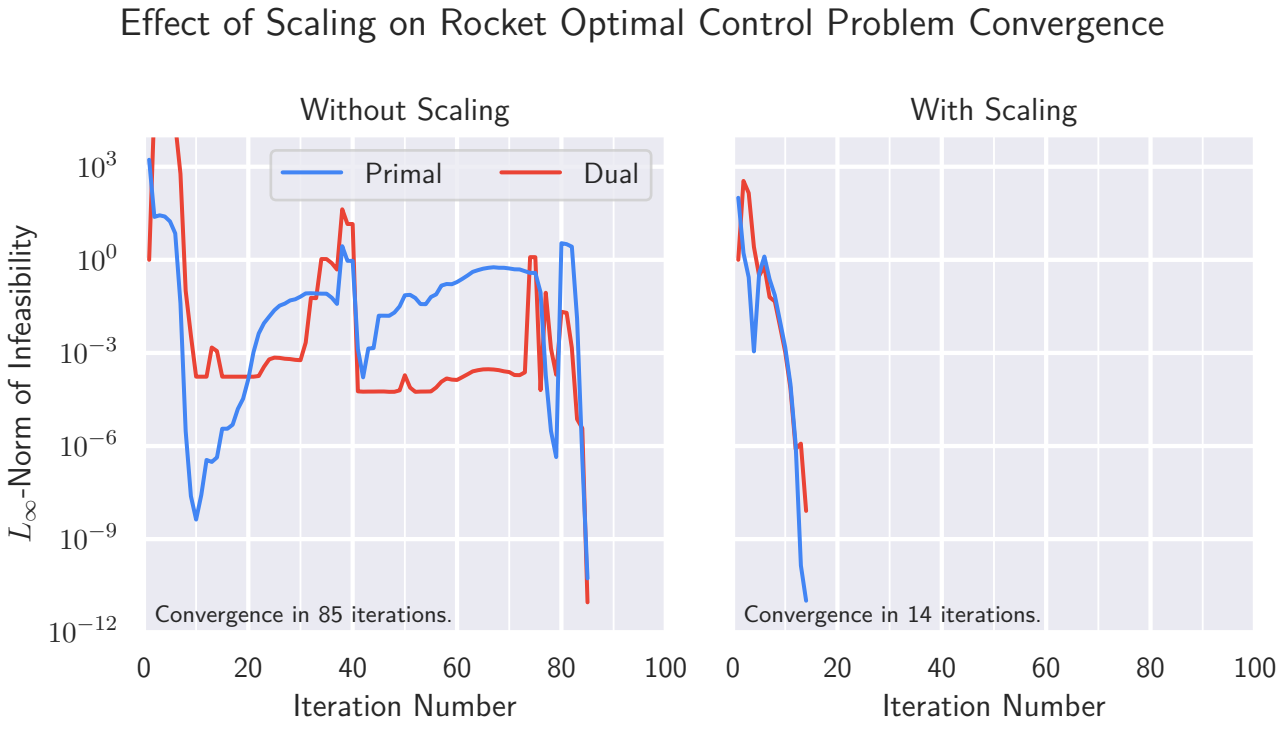

Figure 4-5: Convergence of the Rocket Optimal Control problem (Eq. 4.12), depending on scaling.

A stark difference is also seen in terms of wall-clock time to solve, as presented in Table 4.3; the auto-scaling heuristic results in an 8x speedup.

Table 4.3: Effect of auto-scaling on Rocket Optimal Control problem runtime.

Run Type	Runtime[a]
Without auto-scaling	0.320 s
With auto-scaling	0.038 s

[a] Both tested on an Intel i7-8750H CPU Windows 10 laptop; median over 10 runs.

Log-Transformation

Another transformation that can be easily employed in AeroSandbox is log-transformation. Log-transformation is the process of applying a logarithm to a variable before passing it to the optimizer, such that the algorithm optimizes the quantity $\ln(x)$ rather than x. (The same can also be done to constraints and objectives.)

Log-transformation for engineering design optimization was extensively studied by Boyd [5], with subsequent applications by Hoburg [23], Ozturk [34], Kirschen [27], and many others. The mathematical rationale behind log-transformation is that certain types of expressions[19] that sometimes appear in engineering problems can be guaranteed to become convex under log-transformation. This approach, when taken to the extreme, results in *geometric programming* (GP); a broad comparison of GP with AeroSandbox's more general approach is given in Section 9.1.1.

In the context of a general NLP framework like AeroSandbox, the usefulness of log-transformation can vary widely problem-to-problem. Here, we make some general observations about the pros and cons of this technique:

- Pros:
    - Many engineering expressions become convex or more-convex when log-transformed, which can lead to faster, more stable solutions.

[19] Known as *monomial* and *posynomial* expressions, as defined in [23]

- Many scaling issues disappear under log-transformation, as many orders of magnitude can be spanned with relatively little change in the underlying log-transformed variable.
- We more faithfully represent our design intent. Performance often most intuitively considered multiplicatively, not additively. For example, in aircraft design, a 10% drag decrease is significant no matter the scale; however, the significance of a 10 N drag decrease is entirely dependent on aircraft scale. In other words, log-transformation effectively nondimensionalizes a design variable, as the log-transformed quantity is unitless.
- For quantities that would otherwise need to be constrained to be positive (e.g. vehicle mass), we can eliminate one constraint: the constrained problem has been transformed into an unconstrained one.

- Cons:
  - In some cases, log-transformation can transform an engineering expression from a convex one into a nonconvex one. Therefore, the benefit of log-transformation is highly case-dependent.
  - Log-transformation is yet another nonlinearity, which can make the problem more difficult to solve. In particular, the affine constraints that appear often in engineering design become nonlinear (and sometimes nonconvex) under log-transformation, leading to much worse performance. In practice, this performance loss from nonlinearity becomes quite significant for high-dimensional problems.
  - If the optimal value of a variable is ever zero or negative (e.g. thrust in the Rocket Optimal Control example of Eq. 4.12), convergence on the log-transformed problem is not possible. (The problem becomes unbounded.)

Because of all these reasons, we generally find that it is best to start modeling a design problem without log-transformations, and to only add them in later if

the problem physics justify it. In AeroSandbox, this can be done by passing the `log_transform=True` argument at variable declaration.

4.3.3 Example: Simple Aircraft (SimpleAC)

Now armed with more understanding of the problem transformations available in AeroSandbox, we can also demonstrate performance on a more sophisticated aircraft design problem. This problem, known as *SimpleAC*, models a simple aircraft by extending the Simple Wing problem in Section 4.3.1 with fuel weight and volume models. It was proposed by Hoburg in [23] and is reproduced by both Ozturk [34] and Kirschen [27]. It is restated here in full form:

Simple Airplane (SimpleAC)

$$\begin{aligned} \underset{\text{Æ}, S, V, W, C_L, W_f, V_{\text{f, fuse}}}{\text{minimize}} \quad & W_f \\ \text{subject to} \quad & W \geq W_0 + W_w + W_f, \\ & W_0 + W_w + \frac{1}{2} W_f \leq L_{\text{cruise}}, \\ & W \leq L_{\text{takeoff}}, \\ & W_f \geq \text{TSFC} \cdot t_{\text{flight}} \cdot D, \\ & V_{\text{f, wing}} + V_{\text{f, fuse}} \geq V_f \end{aligned} \tag{4.13}$$

where:
- 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

All of the physics models from Simple Wing (Eq. 4.11) apply, and we

are also given the following new physics models:

- The flight time $t_{\text{flight}} = \text{Range}/V$
- The fuselage drag area is no longer constant, instead scaling 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 $V_{\text{f, wing}} = 0.03S^{1.5}\mathit{AR}^{-0.5}\tau$

Constants are identical to Simple Wing (Eq. 4.11), except for the following redefined ones:

- $k = 1.17$, the form factor.
- $e = 0.92$, the Oswald efficiency factor.
- $\mu = 1.775 \times 10^{-5}\text{ kg m}^{-1}\text{ s}^{-1}$, the sea-level dynamic viscosity of air.
- $N = 3.3$, the ultimate load factor.
- $V_{\text{min}} = 25\text{ m/s}$, the takeoff airspeed.
- $C_{L,\text{max}} = 1.6$, the takeoff lift coefficient.
- $S_{\text{wet}}/S = 2.075$, the wetted area ratio.
- $W_0 = 6250\text{ N}$, the aircraft weight excluding the wing and fuel.
- $W_{\text{w, c1}} = 2 \times 10^{-5}\text{ m}^{-1}$, a wing weight coefficient.
- $W_{\text{w, c2}} = 60\text{ Pa}$, another wing weight coefficient.

The following new constants are added:

- $g = 9.81\text{ m/s}^2$, Earth gravity.
- $\rho_f = 817\text{ kg/m}^3$, the density of fuel.
- $\text{Range} = 1000\text{ km}$, the aircraft mission range.
- $\text{TSFC} = 0.6\,\text{h}^{-1} = 1.67 \times 10^{-4}\,\text{s}^{-1}$, the thrust-specific fuel consumption.

SimpleAC clearly borrows many models from Simple Wing, although it has a different optimization objective: instead of minimizing drag, here we minimize fuel burn for a fixed mission range. Intuitively, we expect this to drive the optimizer to

faster designs, and this is what is observed.

This problem is solved in AeroSandbox, employing log-transformation on all variables, as discussed in Section 4.3.2. Code to solve this problem is available in Appendix C.3. This converges to a solution in 23 milliseconds and 14 iterations. The results of this AeroSandbox solve are shown in Table 4.4.

Table 4.4: Solution of SimpleAC (Eq. 4.13), found with AeroSandbox.

Figure of Merit	Optimal Value
Fuel weight W_f	937.8 N
Aspect ratio $\mathcal{R}$	12.10
Wing area S	14.15 m^2
Cruise airspeed V	57.11 m/s
All-up weight W	8705 N
Cruise lift coefficient C_L	0.2901
Fuel volume in fuselage $V_{\text{f, fuse}}$	0.0619 m^3

We can once again benchmark performance against GPKit, the optimization library that SimpleAC is derived from. This comparison is shown in Table 4.5. Here, AeroSandbox solves the same engineering design optimization problem approximately six times faster than GPKit.

Table 4.5: Comparison of optimization methods: SimpleAC problem.

Optimizer	Optimality		Speed	
	W_f (Objective)	Optimal?	Runtime	Func. Evals.
GPKit	937.8 N	✓	0.141 s	2 (GP iters.)
AeroSandbox	937.8 N	✓	0.023 s	16

Both tested on an Intel i7-8750H CPU Windows 10 laptop; median over 10 runs.

4.4 Numerics Stack

One of the primary reasons for AeroSandbox's speed is its use of end-to-end automatic differentiation, as described in Section 4.1.2. In order for automatic differentiation to work, we need to be able to make a computational graph that contains each mathematical operation that is applied throughout our optimization formulation.

This means that a standard Python numerics library such as NumPy cannot be directly used, because some of these functions break the computational graph - they are not *differentiable*, in a code sense.

Instead, we require a custom differentiable numerics library. Learning a custom numerics library sounds like a daunting task for the end user at first, and in other frameworks, it can be: consider that the two major modern machine learning libraries, PyTorch and TensorFlow, each include custom submodules for differentiable numerics. Both of these submodules use package-specific syntax; in effect, they are a new programming language within a programming language.

AeroSandbox takes a different approach that attempts to make using this numerics library as seamless as possible. AeroSandbox does not introduce new package-specific syntax for its numerics library. Instead, AeroSandbox deliberately overloads syntax from the ubiquitous NumPy package, which has syntax that is already second-nature to any user who performs scientific computing in Python[20] [20]. Thus, the AeroSandbox numerics library can be imported using a drop-in replacement for the standard NumPy import, as shown in Figure 4-6.

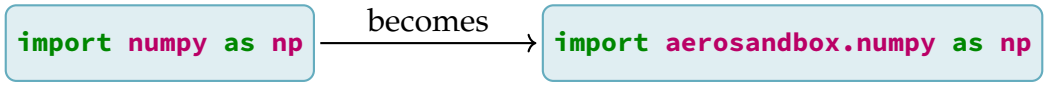

Figure 4-6: Standard import of the AeroSandbox numerics stack.

The `aerosandbox.numpy` numerics stack is a superset of NumPy, so any function contained in the normal NumPy library can be used for general-purpose computing.

[20] This practice of extending NumPy is inspired by similar approaches in the Google JAX [6] and `autograd` packages.

The AeroSandbox numerics stack works by stitching together the (non-differentiable) NumPy package with the (automatic-differentiable) CasADi numerics library, intelligently switching between the two libraries as-needed based on input data types. NumPy is a very large package, and not every NumPy function has been given automatic differentiation support. However, the list of supported differentiable functions is quite extensive, including:

Differentiable Functions using NumPy-like Syntax in the AeroSandbox Numerics Stack

- 1D and 2D array operations (initialization, indexing, concatenation, reshaping, etc.)
    - Higher-dimensional array operations, in the slower "object" mode.
- Elementary operators
    - Arithmetic
    - Trigonometry (including inverse, hyperbolic, and inverse hyperbolic functions)
    - Powers, exponentials, and logarithms
    - Absolute values, `min(x, y)`, `max(x, y)`, and other miscellaneous common operators
- Conditionals, loops, and boolean logical expressions
- Linear algebra
    - Vector operations (dot, cross, outer products etc.)
    - Vector and matrix norms
    - Vector and matrix products
    - Linear solves, Moore-Penrose pseudoinverses
    - Eigenvalue decomposition and other factorizations
- Many common utility functions (e.g. `linspace()`)

- Interpolation, including N-dimensional and higher-order
- Numerical differentiation and integration
- etc.

Advanced solvers for specific subproblems (DAEs, nonlinear iterative rootfinders, QP solvers, SOCP solvers, adaptive-step integrators, etc.) are all available by directly interfacing with the underlying CasADi library; these are mutually-compatible with the AeroSandbox numerics stack. Speaking more broadly, the breadth of functions supported here is largely indebted to the richness of the CasADi library [3].

In practice, this list of differentiable functions covers every single practical engineering design case that has been examined to date.

In the event that a function without differentiation support is called on an object that requires differentiation (e.g. any composition of an optimization variable), an error is thrown. The AeroSandbox numerics stack automatically falls back on the (non-differentiable) NumPy backend on objects that do not require differentiation, as this is more speed- and memory-efficient.

All functions in the AeroSandbox numerics library are continuously tested with both numerics backends and cross-compared to verify reliability; this testing campaign is further detailed in Section 3.2.3.

4.5 Limitations

Although the AeroSandbox core (which consists of the optimization and numerics stacks) is quite powerful, it is not without a few limitations.

4.5.1 Restriction to Glass-Box Models

Models used in AeroSandbox must be *glass-box*. In other words, models must be coded in Python using `aerosandbox.numpy` numerics. Mathematically, nearly every engineering analysis can fit into this framework. In practice, however, users

sometimes wish to use existing legacy *black-box* codes (e.g. a RANS CFD code written in C++), but this is not possible as it breaks the automatic differentiation trace.

Although these black-box models cannot be used as-is, there are still several workarounds that allow these models to be integrated into a differentiable framework:

- **Surrogate modeling:** Instead of directly using the black-box model, we draw samples from the model that are then used to construct a differentiable surrogate model. This approach is tractable when the input space of the black-box model can be *well-sampled*. Specifically, this works well when the model's input dimensionality, runtime, and nonlinearity are relatively low. Surrogate modeling approaches are highly integrated into AeroSandbox; popular surrogate modeling techniques such as fitting and interpolation are discussed in Chapter 5.

- **Rewrite the code:** For simpler black-box models, it is often possible to rewrite these models from scratch in AeroSandbox syntax, allowing them to be directly integrated into the code. This is often less daunting of a task than it may initially appear, as demonstrated with many examples in Chapter 6.

- Future work aims to integrate black-box modeling into AeroSandbox via finite differencing or a user-provided gradient. This is described more in Section 9.2.

4.5.2 Requirements on Differentiability and Continuity

Models used in AeroSandbox should be generally be composed of differentiable functions for good performance. Fortunately all functions of engineering interest are differentiable over almost all[21] of their domain.

However, consider the following trivial optimization problem:

[21] using "almost all" in the mathematical sense

$$\underset{x}{\text{minimize}} \quad |x| \tag{4.14}$$

This problem can be directly posed to AeroSandbox, as `abs()` is a function included in the AeroSandbox numerics stack:

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

opti = asb.Opti()
x = opti.variable(init_guess=1)
opti.minimize(np.abs(x))
sol = opti.solve()
```

Unfortunately, this optimization program fails to converge and throws an error. We can rationalize this by considering the perspective of the optimizer. The second-order IPOPT optimizer is constructing a local quadratic representation[22] of the design space as the optimization process progresses.

For the function $f(x) = |x|$, we can construct a quadratic representation based on local information at almost all points, as $f'(x)$ and $f''(x)$ are defined almost everywhere. However, at $x = 0$, these derivatives are undefined, so there is no clear quadratic approximation[23]. Therefore, the optimizer fails.

From this, we can construct a rule of thumb:

> **Continuity Guidelines for Gradient-Based Optimization**
>
> If the objective function or any of the constraints are not C_1-continuous[a] at the optimum, convergence will almost certainly fail.
>
> [a] meaning they are continuous and have a continuous first derivative

We note that this discontinuity in $|x|$ at $x = 0$ is only troublesome because it is the optimum, so the optimizer asymptotically approaches it. If the discontinuity is not at the optimum, it poses no problem. In the extreme case, it is even possible to

[22] For memory reasons, this representation is deliberately approximate, similar to the L-BFGS algorithm.

[23] Even generalized definitions of the derivative lead to a second derivative that is similar to a Dirac delta function, which causes a numerical singularity.

use a differentiable solver like AeroSandbox to solve problems with infinitely many discontinuous, non-differentiable points, as demonstrated in Appendix C.4.

Unfortunately, optimization problems with discontinuities (in either their objective or constraints) *tend* to have optima at these discontinuities. This is rigorously true in the case of linear programming, where it is guaranteed that an optimum (if it exists) will be at a vertex of the feasible polytope[24]. Furthermore, C_1-discontinuity (localized to specific points) can be found in a non-negligible number of engineering models: vector norms and piecewise functions are often not C_1-continuous.

Therefore, it is worthwhile to identify several methods for addressing discontinuities and non-differentiability in engineering problems.

Method 1: Rewrite the Problem

Often, it is possible to simply rewrite the problem in a way that doesn't use discontinuous functions. The optimization problem described in Eq. 4.14 can be rewritten in a continuous manner by splitting the absolute value into two linear constraints[25]:

$$\begin{aligned} \underset{x,\,y}{\text{minimize}} \quad & y \\ \text{subject to} \quad & y \geq x, \\ & y \geq -x \end{aligned} \tag{4.15}$$

Equation 4.15 is easily solved as-is in AeroSandbox, and this is by far the preferred approach to resolving discontinuities if the original formulation allows.

However, this isn't always possible - to speak precisely, this is not possible when the objective function or constraint is nonconvex[26] at the discontinuity itself. An example of this is the following function:

[24] This optimum may not be unique, depending on problem construction. This observation is the foundational theorem of the *simplex method*.

[25] Subgradient methods are another approach to rewrite convex but not differentiable functions, though they are not detailed here.

[26] If the objective function or constraint is C_0-discontinuous, it is nonconvex by definition.

$$f(x) = \begin{cases} -x & \text{for } x < 0 \\ (x-1)^2 - 1 & \text{for } x \geq 0 \end{cases} \tag{4.16}$$

which is visualized in Figure 4-7. If the non-differentiable point at $x = 0$ needed to be resolved, other methods are required.

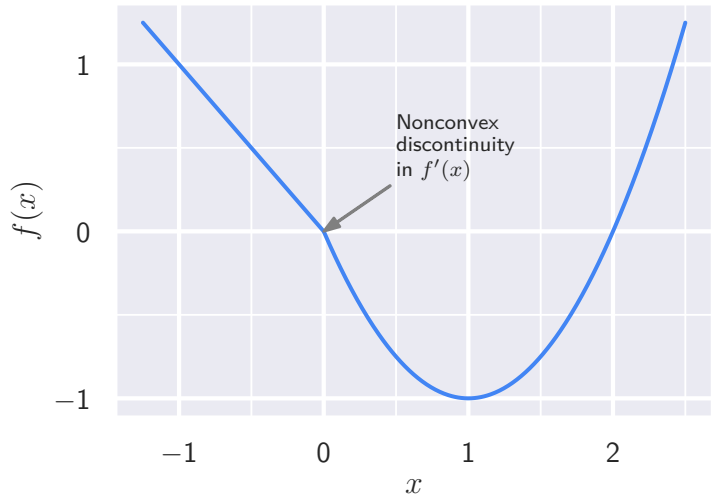

Figure 4-7: The function described in Eq. 4.16.

Method 2: Construct a Continuous Approximation

One such alternative method to resolve discontinuous functions is to approximate them with continuous ones. Note that, unlike the previous method, this is actually changing the problem rather than simply rewriting it, so the optimum will also be only an approximation to that of the original problem.

On the canonical example from Eq. 4.14 of $f(x) = |x|$, we can introduce two possible continuous approximations that are given by Kelly [25]:

$$\begin{aligned} &\text{Approximation 1:} \quad \hat{f}_1(x) = x \tanh(x/\alpha) \\ &\text{Approximation 2:} \quad \hat{f}_2(x) = \sqrt{x^2 + \alpha^2} \end{aligned} \tag{4.17}$$

where, in both cases, α represents a parameter that controls the amount of approxi-

mation. These functions are depicted in Figure 4-8.

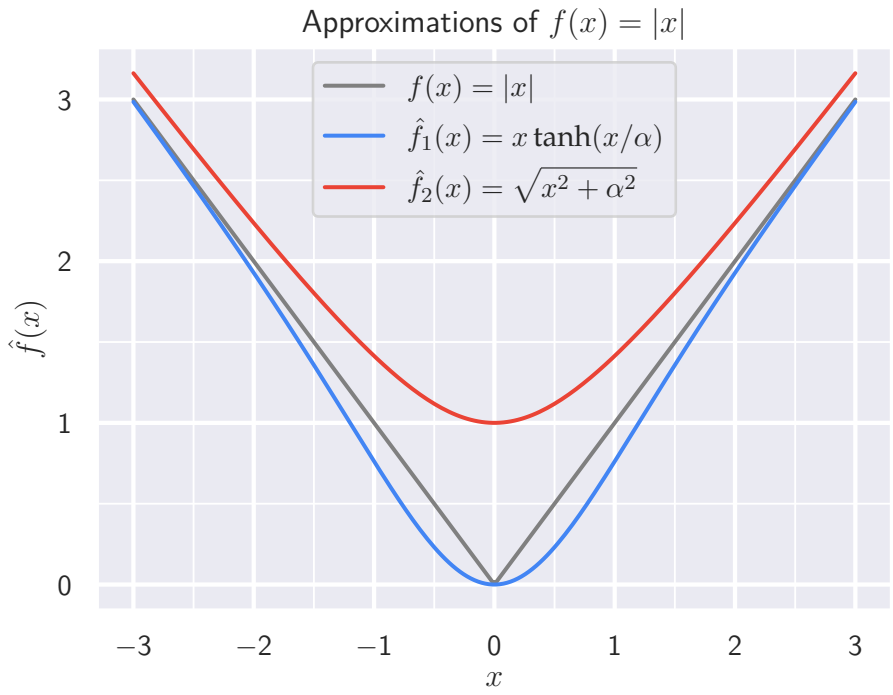

Figure 4-8: Two approximations to $f(x) = |x|$ as given in Eq. 4.17.

Because both of these approximations are C_1-continuous everywhere, minimization of these approximate functions is easily performed with AeroSandbox. We note that the second approximation, $\hat{f}_2(x) = \sqrt{x^2 + \alpha^2}$, is generally a superior choice, as it preserves convexity. This means that one can guarantee global optimality upon subsequent arbitrary convex transformations of the function.

Generalized Methods to Fix Discontinuities

Of course, $|x|$ is far from the only discontinuous function that one would want to optimize, and creating these continuous approximators for each individual function would be quite tedious. We can therefore present a few generalized methods to fix discontinuities that work on a wider array of problems.

Softmax One of the more common sources of C_1-discontinuity is the max() operator, which yields an element-wise maximum of its inputs. To resolve this, a convex operator called "softmax"[27] can be used to replace the max() operator. Softmax is defined as:

$$\max(x, y) \approx \text{softmax}(x, y) = \ln(e^x + e^y) \tag{4.18}$$

It can be generalized to allow an arbitrary number of inputs as well as a "hardness" parameter that details the amount of approximation relative to the max() operator. Both of these extensions are detailed further by Cook [7], and a visualization of this generalized softmax is given in Figure 4-9.

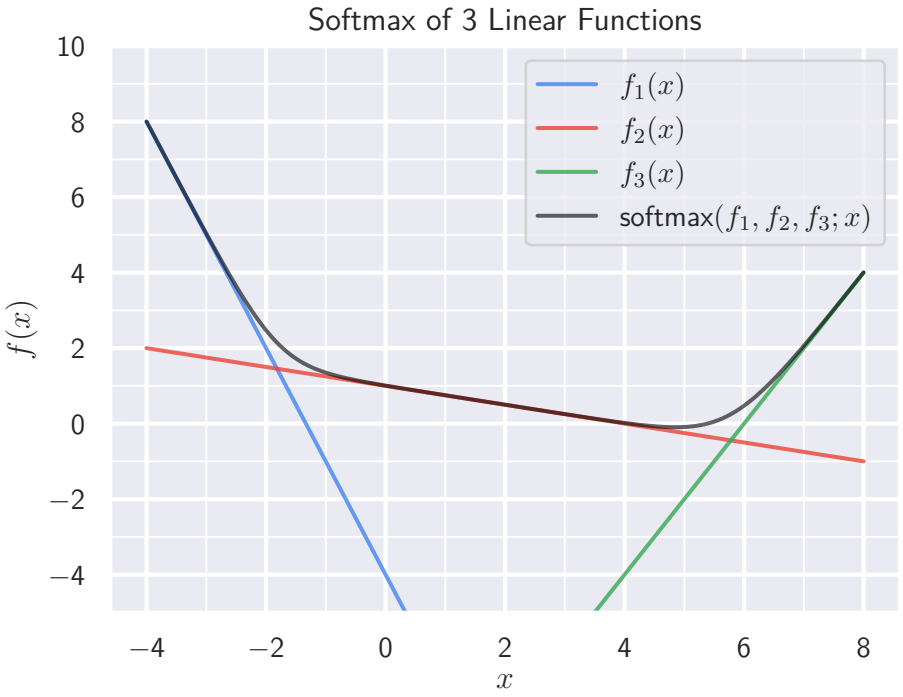

Figure 4-9: The Softmax function

Softmax requires careful numerical implementation that prevents floating-point overflow or underflow due to the exponentiation in its definition; this is implemented

[27] also known as "logsumexp" in some fields

in the AeroSandbox numerics stack[28].

Sigmoid Blending For general, nonconvex discontinuities, we can blend between the functions on either side of a piecewise discontinuity by using a sigmoid transition function. This "blend" operator[29] is effectively a smooth approximation to a conditional operator.

To define this blend operator, we first must define a sigmoid function $\sigma(x)$ that is scaled to have the following properties:

$$\begin{aligned} \lim_{x \to -\infty} \sigma(x) &= 0 \\ \sigma(0) &= \frac{1}{2} \\ \lim_{x \to \infty} \sigma(x) &= 1 \end{aligned}$$

One such function is:

$$\sigma(x) = \frac{\tanh(x) + 1}{2} \tag{4.19}$$

Then, we can define the blend operator as a sigmoid-weighted linear combination of the two sides of the discontinuity:

$$\text{blend}\big(f_1, f_2, s; x\big) = \sigma\big(s(x)\big) \cdot f_1(x) + \Big[1 - \sigma\big(s(x)\big)\Big] \cdot f_2(x) \tag{4.20}$$

where $f_1(x)$ = The function on one side of the discontinuity

$f_2(x)$ = The function on the other side of the discontinuity

$s(x)$ = A "switch" function that acts as the conditional statement, depending on its value. Hardness can be controlled by linear scaling of $s(x)$.

The blend operator does not preserve convexity as softmax does, but it has the benefit of yielding a general approximation for a much broader set of discontinuous problems. The blend operator is a generalization of the first approximation to $|x|$ given in Eq. 4.17, so its graphical characteristics can be examined in the $\hat{f}_1(x)$ trace

[28] implemented as `aerosandbox.numpy.softmax`

[29] implemented as `aerosandbox.numpy.blend`

of Figure 4-8.

4.6 Nonlinear Feasibility Problems and SAND Architectures

A general nonlinear optimization framework such as AeroSandbox has utility far beyond ordinary optimization problems, as many workhorse algorithms of scientific computing can be efficiently reframed as nonlinear optimization problems.

Here, we examine the common problem of solving an implicit system of governing nonlinear equations. This task is effectively synonymous with engineering analysis: nearly all analyses can be distilled into solving a system of nonlinear equations. In a general sense, this analysis problem can be written as a root-finding problem for the vector-valued function g:

$$\begin{aligned} &\vec{x} \in \mathbb{R}^n \\ &\vec{g} : \ \mathbb{R}^m \to \mathbb{R}^n \end{aligned}$$

$$\vec{g}(\vec{x}) = \vec{0} \implies \begin{cases} g_1(x_1, x_2, \dots, x_n) = 0 \\ g_2(x_1, x_2, \dots, x_n) = 0 \\ \dots \\ g_m(x_1, x_2, \dots, x_n) = 0 \end{cases} \tag{4.21}$$

This can be rewritten as an optimization problem with equality constraints and no objective:

$$\begin{aligned} \underset{\vec{x}}{\text{minimize}} \quad & 0 \\ \text{subject to} \quad & \vec{g}(\vec{x}) = \vec{0} \end{aligned} \tag{4.22}$$

This type of optimization problem is known as a *feasibility problem*, where the

goal is to find any feasible solution. This corresponding feasibility problem shown in Eq. 4.22 encodes the same problem information as the original nonlinear system, yet it generalizes the notion of a nonlinear solve to allow many powerful new features.

First, we consider that the original nonlinear problem given in Eq. 4.21 would likely be solved with a Newton iteration method, which tends to be the most efficient choice for large, strongly-coupled problems. A classical Newton's method requires several specific properties: $m = n$, and the Jacobian matrix must be invertible at every step of the iterative solve[30].

Assume these properties hold. Then, if the corresponding optimization problem in Eq. 4.22 is given to a modern 2$^{\text{nd}}$-order gradient-based optimizer (such as IPOPT via AeroSandbox), it will also be directly solved with a Newton system (after the Lagrangian is formed) to yield the same result as Eq. 4.21. Therefore, although Eq. 4.22 perhaps appears more complex than Eq. 4.21, computational performance on these two problems is essentially identical.

The power of the optimization approach shows when the problem is less well-behaved and these conditions for a classical Newton's method are not satisfied. In the case of an underconstrained system (generally $m < n$), the optimization frameworks allows a solution to be found, even though the solution is not unique. If desired, the solution can be made unique by adding an objective function that applies weak optimization pressure in a given direction; this essentially "selects" a desired solution from the set[31] of all feasible ones.

In the case of an overconstrained system ($m > n$), Eq. 4.22, the optimization framework allows the constraints to be relaxed in a variety of ways. The objective can be used to regularize the system as desired, including common examples such as:

[30] These requirements can be alleviated with modified Newton methods that use generalized inverses.

[31] generally uncountably infinite

L_1-minimization of $\vec{g}(\vec{x})$:

$$\begin{aligned} \min_{\vec{x},\vec{y}} \quad & \sum_{j=1}^{n} y_j \\ \text{s.t.} \quad & \vec{y} \geq \vec{g}(\vec{x}), \\ & \vec{y} \geq -\vec{g}(\vec{x}) \end{aligned}$$

L_2-minimization of $\vec{g}(\vec{x})$:

$$\min_{\vec{x}} \quad \sum_{i=1}^{m} g_i(\vec{x})^2$$

L_∞-minimization of $\vec{g}(\vec{x})$:

$$\begin{aligned} \min_{\vec{x},y} \quad & y \\ \text{s.t.} \quad & y \geq \vec{g}(\vec{x}), \\ & y \geq -\vec{g}(\vec{x}) \end{aligned}$$

Figure 4-10: Methods of regularizing overconstrained nonlinear systems using various norm metrics.

In both of these cases, an optimization formulation lets us regularize our ill-posed original nonlinear system into a well-posed optimization problem, with no loss in computational speed.

4.6.1 Simultaneous Analysis and Design (SAND)

However, perhaps the most powerful feature of moving nonlinear systems of equations into the constraints of an optimization problem is that it lets us arbitrarily combine many nonlinear analyses. And, because a design optimization problem can be thought of as an underconstrained analysis, we can once again use the objective function to "select" an optimal design from the set of feasible designs.

This technique is aptly known as *Simultaneous Analysis and Design* (SAND), as it essentially embeds the engineering analyses required for design optimization into the optimization problem itself: analyses and cross-discipline relationships are enforced implicitly via equality constraints.

The SAND paradigm, which was pioneered by Haftka [19], is an extremely efficient way to address the coupling problems of aerospace design optimization that were described in Section 2.3. To understand the efficiency benefit of SAND, we must consider its alternative: the more straightforward *nested* optimization architecture.

In a nested architecture, an optimizer is simply wrapped around an existing

analysis code. As the optimizer iterates, this analysis code (which, for generality, is assumed to be nonlinear) will perform subiterations at each iteration in order to satisfy its governing equations. This is quite wasteful, as computational power is spent satisfying governing equations to precise tolerances early in the optimization process when the design is far from optimal.

By contrast, a SAND architectures solves for both *optimality*[32] and *feasibility* (i.e. satisfaction of the governing equations) simultaneously. This is more theoretically sound, as it recognizes the observation that feasibility without optimality is uninteresting, and optimality without feasibility is uninteresting. Additionally, SAND is much more computationally efficient than nested optimization in practice, as it eliminates internal closure loops and subiteration.

Of course, the drawback of a SAND architecture is that it does not work on black-box models: the governing equations must be directly accessible to the optimizer in order to include them in the constraints. However, because this requirement is identical to the glass-box requirement of automatic differentiation, this does not present a new restriction in the case of a differentiable optimization environment like AeroSandbox.

4.7 Integrators: Solving ODEs with AeroSandbox

One category of governing equation that we often wish to solve using a SAND architecture is an ordinary differential equation (ODE). ODEs appear in many appear in any aircraft design optimization problem where dynamics or mission profile are important; as discussed in Section 2.1, this represents the vast majority of aerospace cases.

In the case of an ODE, the idea of using a SAND architecture essentially entails formulating an implicit integration scheme where relations between timesteps are posed as equality constraints.

[32] or more precisely, *dual feasibility*

4.7.1 Example: Falkner-Skan ODE

These principles can be demonstrated on an aerospace analysis example by solving the Falkner-Skan boundary layer equation. The Falkner-Skan boundary layer equation describes the velocity profile of viscous flow within a self-similar boundary layer, under the assumption that the edge velocity of the boundary layer follows some power law with respect to downstream distance. The equation, which is described more fully by Drela [15], can be written as:

Falkner-Skan Boundary Layer Problem

Solve for $F(\eta)$, given the governing ODE:

$$F''' + \frac{1+a}{2}(F \cdot F'') + a\left[1 - (F')^2\right] = 0 \tag{4.23}$$

with boundary conditions:

$$F(0) = 0$$

$$F'(0) = 0$$

$$F'(\infty) = 1, \text{ which we approximate as } F'(10) = 1$$

where:
- η = The nondimensionalized distance from the wall
- $(\,)'$ = A derivative with respect to η
- $F(\eta)$ = The nondimensionalized total mass flow rate between the wall and η
- $F'(\eta)$ = The nondimensionalized velocity profile; $F' = u/u_e$
- $F''(\eta)$ = The nondimensionalized shear profile
- a = A known constant that describes the edge velocity profile as $u_e \propto x^a$, with $a \geq -0.0904$ known[a]

[a] A separation singularity occurs below this value.

The Falkner-Skan equation is chosen because it shares many qualities typical of ODEs in engineering systems:

- It is nonlinear

- It is higher-order (third-order, specifically)
- It is a boundary value problem rather than an initial value problem, so explicit solution is difficult[33].

Solving the ODE

To solve this ODE within an optimization framework, we first make the observation that the Falkner-Skan equation (like all higher-order ODEs) can be split apart into a system of coupled first-order ODEs. Then, using a change of variables and algebraic manipulation, we can write the equivalent ODE system of three first-order equations:

$$\begin{aligned} F' &= U \\ U' &= S \\ S' &= -\frac{1+a}{2}FS - a(1-U^2) \end{aligned} \tag{4.24}$$

with boundary conditions:

$$\begin{aligned} F(0) &= 0 \\ U(0) &= 0 \\ S(10) &= 1 \end{aligned}$$

Suppose that we wish to solve this ODE for $a = 0.1$ using the AeroSandbox optimization framework. The AeroSandbox optimization stack has bespoke methods[34] which declare new variables as derivatives of existing variables and constrain existing variables. By default, these implement a direct trapezoidal collocation method for integration, following the principles described by Kelly [25]. The result is that this complicated ODE can be solved with sophisticated methods in less than 30 lines of code in AeroSandbox, as demonstrated in Listing 3.

We require an initial guess, and for that we guess a parabolic velocity profile

[33] Eq. 4.23 can be solved with explicit integration using a shooting method as described by Kelly [25], but is extremely unstable due to the high ODE order and nonlinearity of the Falkner-Skan equations.

[34] `Opti.derivative_of()` and `Opti.constrain_derivative()`

given by $U(\eta) = 1 - (1 - \eta/10)^2$; this is then integrated and differentiated to obtain guesses for F and S. This system can now be solved, with the resulting F, U, and S arrays are presented in Figure 4-11. We note that despite the fact that this is a third-order boundary value problem and therefore prone to divergence upon explicit integration, the implicit SAND approach shown in Listing 3 handles it fine, converging in only 4 iterations.

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

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

a = 0.1

eta = np.linspace(0, 10, 100)  # Discretize η ∈ [0, 10] with 100 points.

F = opti.variable(init_guess=eta + 10 / 3 * (1 - eta / 10) ** 3)

U = opti.derivative_of(F, with_respect_to=eta,
    derivative_init_guess=1 - (1 - eta / 10) ** 2
)
S = opti.derivative_of(U, with_respect_to=eta,
    derivative_init_guess=0.2 * (1 - eta / 10)
)

opti.constrain_derivative(S, with_respect_to=eta,
    derivative=-(1 + a) / 2 * F * S - a * (1 - U ** 2)
)

opti.subject_to([  # Implement boundary conditions.
    F[0] == 0,
    U[0] == 0,
    U[-1] == 1,
])

sol = opti.solve()
```

Listing 3: The AeroSandbox code to solve the Falkner-Skan system as in Eq. 4.7.1.

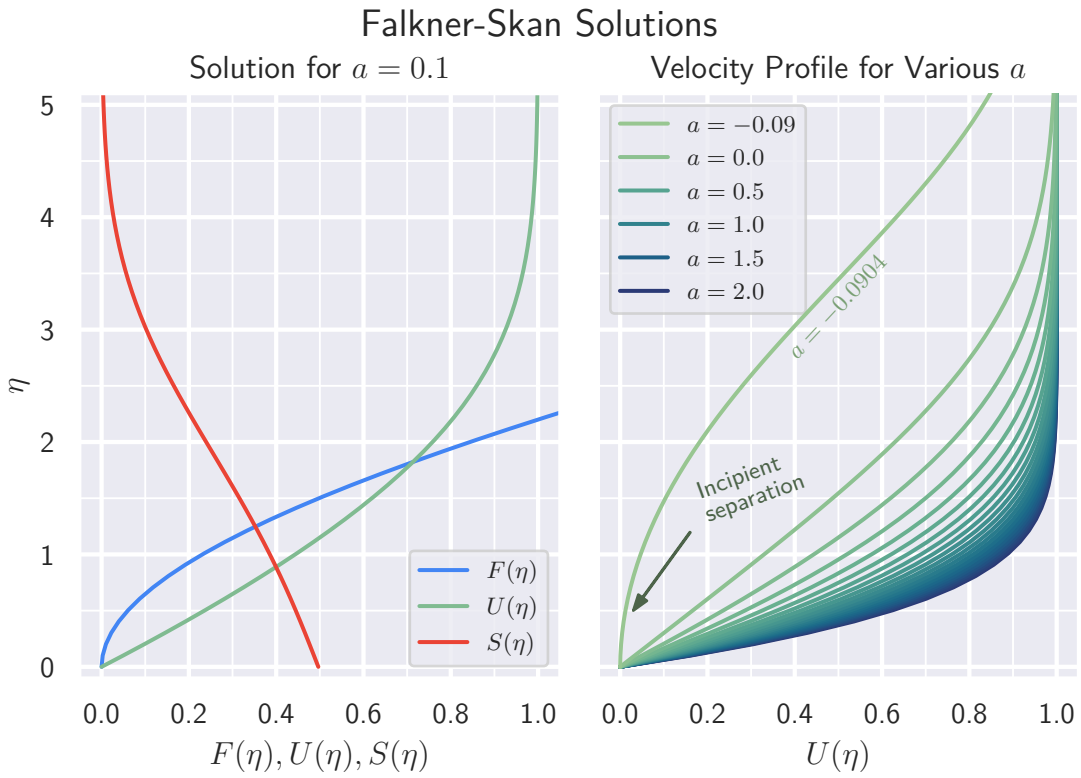

Figure 4-11: Solutions of the Falkner-Skan system in Eq. 4.7.1 from the code in Listing 3.

Inverse Analysis

However, the power of ODE analysis in a SAND architecture extends far beyond simply solving the ODE. Here, we demonstrate this by turning the problem around and performing inverse analysis.

Referring to Figure 4-11, we notice that an interesting phenomenon is occuring for the $a \approx -0.0904$ case. This is an incipient separation flow, which can be identified from the fact that the nondimensional wall shear $S(\eta) = \frac{\mathrm{d}U}{\mathrm{d}\eta}$ at the wall $(\eta = 0)$ is approaching zero.

However, the value of a for this incipient separation flow is only approximate, and it was found by trial and error. Suppose we now ask: at what exact value of a does the flow separate, as indicated by exactly zero wall shear $S(0) = 0$?

If this ODE had been solved with traditional methods, answering this question would be quite laborious - even in an implicit method, tracing a residual equation a manually would be very tedious. We could look to trial and error, but we run into a complication: for $a < -0.0904$, no solution exists, so trial and error is cumbersome. This is common in engineering models - often the forward problem is well-posed, but the inverse problem is not.

However, we can solve this inverse design problem extremely easily in a SAND framework such AeroSandbox, here with the addition of only two lines of code. To modify Listing 3 to solve this inverse problem:

1. We replace the `a = 0.1` line with a new variable definition:

```
a = opti.variable(init_guess=1)
```

2. We simply add a wall shear constraint:

```
opti.subject_to(S[0] == 0)
```

After these modifications, we can obtain the value of a at incipient separation to near machine precision without any trial and error or manual derivation of residual Jacobians. All of these numerics are abstracted by the AeroSandbox solver, which critically lets the user focus on the engineering problem rather than the numerical implementation.

This *inverse analysis* that is demonstrated here occurs all the time in engineering. For example: one might analyze an airfoil at an angle of attack $\alpha = 5^\circ$ and find a lift coefficient $C_L = 0.6$; this is the forward problem. If we then wish to compute the angle of attack that would yield $C_L = 0.5$, we must solve the inverse problem to "go backwards" through this analysis. Inverse analysis is often quite tedious to implement without the abstracted approach used here, but it is easily implemented in AeroSandbox.

Chapter 5

Modeling Tools

5.1 Geometry Stack

To be written...

5.2 Surrogate Modeling Tools

It is critical that a general-purpose engineering design optimization framework such as AeroSandbox allows the use of user-defined physics models. These models can generally be classified into one of three categories:

1. **Analytical models**, which are often derived theoretically and can be specified concisely in closed-form. For example, an analytical relation for aircraft drag, as given by Drela [11] is:

$$C_D(\text{Æ}R, S, \dots) = \frac{\text{CDA}_0}{S} + c_d(C_L, \text{Re}, \tau) + \frac{C_L^2}{\pi \text{Æ}R} \tag{5.1}$$

where: C_D = Drag coefficient, c_d = Profile drag coefficient, $\text{Æ}R$ = Aspect ratio, C_L = Lift coefficient, S = Wing area, Re = Reynolds number, CDA_0 = Fuselage drag area, τ = Taper ratio

2. **Expensive models** that are the result of laborious, black-box external computation. Examples of this might include:
    - A RANS CFD code, or other analysis that requires the solution of 3D nonlinear PDEs
    - Aerostructural analysis with explicit time-domain dynamics
    - An external code that computes engine performance with non-equilibrium gas dynamics
3. **Data-driven models**, where the underlying input to the model is not a set of equations but rather a dataset. Common sources of this data include wind tunnel runs, meteorological data, or experimental testing on prototype components.

Analytical models are trivially implemented into a differentiable optimization framework such as AeroSandbox using the techniques described in Chapter 4. However, the remaining two categories cannot generally be used in a straightforward way.

In the case of expensive models, there are two key problems that make direct implementation unattractive. First, black-box functions break the differentiability trace, so they must be re-coded from scratch in a differentiable numerics framework. Although this is usually possible, it can be tedious. Secondly, expensive models in a SAND framework may yield an optimization problem that takes many minutes or hours to solve, precluding the practice of *interactive design*.

Data-driven models also come with their own set of challenges. First, one must obtain data of sufficient quantity (it spans the input space of the model) and quality (the data has minimal noise). Secondly, one must form some kind of strategy to evaluate the model between known data points (interpolation) and beyond known data (extrapolation).

Both expensive models and data-driven models can be implemented into a differentiable optimization framework using *surrogate modeling*. Surrogate modeling is a collection of techniques that aims to replace an expensive, data-driven, or

otherwise unusable model with a differentiable, cheaply-evaluated model that approximates the original one.

There are two general approaches to surrogate modeling: *fitting* and *interpolation*. Fitting aims to replace an expensive model or dataset with an analytical expression. Interpolation forgoes the need for an explicit analytical expression, instead interpolating from known data points using a piecewise spline.

The present work provides computational tools for surrogate modeling using both of these approaches. In the following section, we detail both of these. We restrict our focus here to models of the form $f : \mathbb{R}^n \to \mathbb{R}^1$, because these are by far the most common types of models, and because vector-valued functions can be constructed by combining several scalar-valued functions.

5.2.1 Fitted Models

One approach to creating a surrogate model is curve fitting, or more formally, *regression*. Fitting is the process of deriving an analytical model that approximates a function from which samples have been drawn. An infinite number of fitted models could be regressed from a given dataset, and choosing which of these models is best is an optimization problem. Specifically, we can write the fitting problem as follows, following notation from [28]:

The Canonical Fitting Problem (Least-squares Regression)
We are given a dataset that consists of m entries. Each entry maps from $\vec{x} \in \mathbb{R}^n$ to $y \in \mathbb{R}^1$. We collectively refer to the inputs and outputs of the dataset as $\mathbf{X}$ and $\vec{y}$, respectively.

A model format is also provided, which includes some unknown parameters $\vec{\theta}$. We then denote the model outputs as:

$$\vec{y}_{\text{model}} = \text{model}\left(\mathbf{X}, \vec{\theta}\right) \tag{5.2}$$

The error of this model at each of the m data points is then:

$$\vec{e} = \vec{y}_{\text{model}} - \vec{y}$$

We then seek to find the optimal value of the model parameters $\vec{\theta}$ by solving the following optimization problem:

$$\vec{\theta}^* = \underset{\vec{\theta}}{\arg\min} \quad \left\|\vec{e}\right\|_2 \tag{5.3}$$

The vector norm in the objective Eq. 5.3 can be rewritten as $\left\|\vec{e}\right\|_2 = \sqrt{\sum_{i=1}^{m} e_i^2}$. Because the square root function is monotonic in the positive domain, it can be removed without changing the value of $\vec{\theta}^*$. This is convenient, as the objective function now tends to be more closely approximated by a quadratic[a], dramatically improving numerical performance. Therefore, the fitting problem can be expressed as:

$$\vec{\theta}^* = \underset{\vec{\theta}}{\arg\min} \quad \sum_{i=1}^{m} e_i^2 \tag{5.4}$$

[a] In the case of *linear* least-squares regression, the objective is now exactly quadratic and admits closed-form solution.

This forms the canonical fitting problem, also known as least-squares regression. Because Equation 5.3 is an optimization problem with entirely glass-box functions, it is efficiently differentiated and solved by AeroSandbox. This functionality is provided by the `asb.FittedModel` class, which acts both as the fitting solver (performed upon instantiation) and the callable model itself.

The generality of the model format in Eq. 5.2 is quite powerful, as the fitting routine presented here can use any composition of elementary operators as its model format. In addition, this optimization approach can fit piecewise functions and expressions that can only be tractably expressed in code (e.g. model formats with loops, complicated conditionals). The AeroSandbox `FittedModel` implementation can also be used to fit datasets with general multidimensional inputs. Furthermore, because the fitting optimization problem can utilize fast gradients via automatic

differentiation, fitting performance scales efficiently with parameter dimensionality[1].

Example: Wind Analysis

The power of this generality is demonstrated in Figure 5-1, where the AeroSandbox `FittedModel` routine is used to regress a model for peak[2] wind speeds at various points in the atmosphere above the continental United States in August[3]. This model holds great importance for high-altitude long-endurance (HALE) aircraft design: the primary failure mode of HALE aircraft in the past two decades has been in-flight aerostructural failure exacerbated by wind gusts. The underlying 2D dataset in this example was obtained via statistical analysis of the ERA5 Global Reanalysis meteorological dataset [21].

[1] analogous to the scaling seen in Figure 4-3
[2] Quantified as the 99th-percentile of wind speed over time
[3] Averaged over years 1979-2020

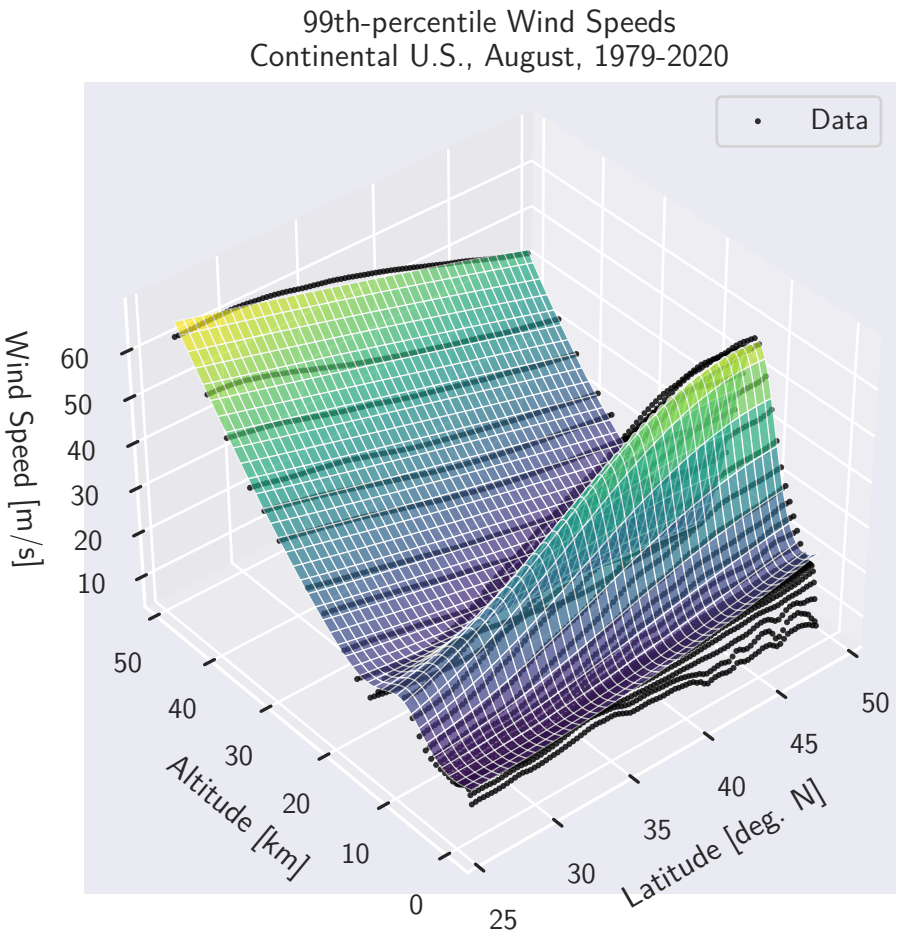

Figure 5-1: An demonstration of `asb.FittedModel`, where an 18-parameter analytical model is fitted to a multidimensional, aerospace-relevant example dataset.

There is a clear and strong nonlinearity present in this dataset, evidenced by the sharp rise in peak wind speeds near (15 km altitude, 50 deg. N latitude). This nonlinearity, which depicts the Arctic polar vortex of the jet stream, makes this a challenging dataset to fit. Here, an 18-parameter model consisting of a combination of polynomials and Gaussian-like terms was used to fit the data.

The resulting model from this fitting process is not only cheaply evaluated, but it is also end-to-end automatic differentiable. This means that it can be used as desired in the optimization framework described in Chapter 4. Fitting also tends to remove noise from the dataset, which makes optimization much more well behaved. This noise removal occurs because the fit is essentially a projection

of the dataset noise (generally relatively high-spatial-frequency[4]) onto the mode shapes associated with model linearization with respect to $\vec{\theta}$ (generally relatively low-spatial-frequency). Because of these desirable properties, curve fitting is a good tool for creating surrogate models from experimental or synthetic datasets.

The classical ordinary least-squares fitting problem described in Equation 5.3 can be extended in several interesting ways in order to obtain fits with more desirable characteristics. Several of these generalizations have been implemented into the AeroSandbox surrogate modeling toolkit via the `asb.FittedModel` class. These features are described in the following sections.

Generalization to Various L_p Norms

The fitting problem specified in Eq. 5.3 minimizes the L_2 norm of the error vector $\vec{e}$, a process known as least-squares fitting. This fitting problem can be generalized by instead minimizing various L_p norms of the error vector. In general, the L_p norm of the error vector can be expressed as:

$$\left\|\vec{e}\right\|_p = \left(\sum_{i=1}^{m} e_i^p\right)^{1/p}, \qquad p \in [1, \infty) \tag{5.5}$$

By analogy to Eq. 5.4, the fit optimization problem associated with this equation is often solved more easily after elimination of the (monotonic) root function.

Of particular interest are the L_1 and L_∞ norms, which can be expressed in special form derived from limit analysis of Eq. 5.5:

$$\left\|\vec{e}\right\|_1 = \sum_{i=1}^{m} |e_i| \qquad\qquad \left\|\vec{e}\right\|_\infty = \max\left(|e_1|, |e_2|, \dots, |e_m|\right)$$

These norms are of special interest for two reasons. First, they represent the extremes of the L_p norm family. Secondly, the fitting problem associated with them can be more efficiently expressed with a reasonable[5] number of constraints[6], as shown by

[4] In the common case of uncorrelated random error, the noise follows a *white noise* spectrum
[5] more precisely, a finite number of constraints that is of $\mathcal{O}(m)$
[6] and in particular, *linear* constraints in the case of linear regression

analogy to Figure 4-10.

The primary practical distinction between fits using these various norms is their response to outliers. This is demonstrated in Figure 5-2, where fits are made to a synthetic example dataset that contains an outlier. The L_1 fit largely eschews the influence of the outlier, essentially discarding the outlier as a *systematic* error rather than a *random* one. The L_∞ fit is the opposite, as it seeks to minimize the maximum deviation; essentially this treats the outlier as a random error that still conveys useful information about the underlying model.

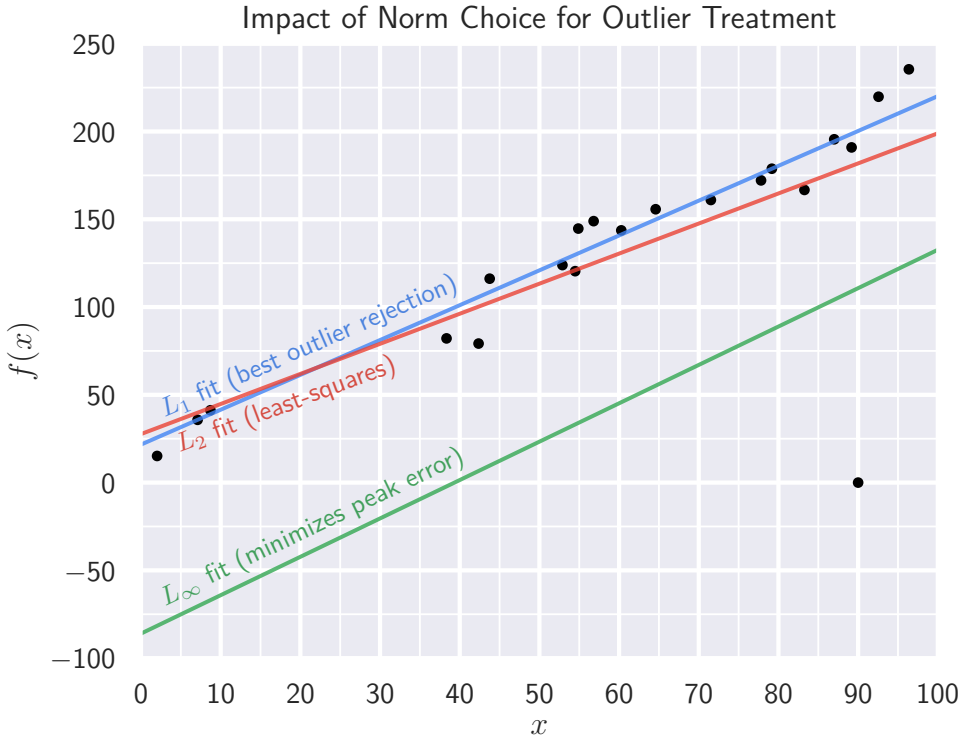

Figure 5-2: Fitting with various norms on an synthetic dataset with an outlier.

Neither approach is universally superior; they simply represent different prior beliefs about the likely source of error as a function of model deviation. Surrogate modeling from a synthetic dataset derived from high-fidelity computational simulation is likely best served by L_∞ fitting, as the noise in the dataset is generally assumed to be zero. The L_∞ fit will enforce the tightest possible bound on the deviation from the high-fidelity dataset.

On the other hand, an L_1 fit might be expected to produce superior results for an experimental dataset. This is because the systematic errors sometimes found in experiment can yield outliers that convey no useful information about the underlying physics. The example dataset in Figure 5-2, which simulates an experimental dataset with measurement dropout, is clearly better served by the robust L_1 fit.

Parameter and Model Bounds

Another useful feature of the AeroSandbox fitting submodule is the ability to easily perform constrained fitting. This can take two forms:

1. **Parameter bounds**: the vector of fit parameters $\vec{\theta}$ can be directly given bounds constraints, which can be used to stabilize the fit process on nonconvex problems.

2. **Model bounds**: the error vector $\vec{e}$ can be constrained such that the fitted model represents either an upper or lower bound on the dataset. This is quite useful in engineering practice, as it allows the creation of surrogate models that can be guaranteed to be conservative with respect to the original dataset. This reduces the likelihood that an optimization problem that includes a fitted model will result in an optimum that is infeasible according to the true underlying physics.

This second process of adding model bounds is demonstrated in the fits in Figure 5-3. Here, a drag polar for a SD7032 airfoil at $\text{Re} = 10^6$ is fit using `asb.FittedModel` and a quadratic model. This quadratic model is a common approximation for an airfoil's profile drag polar; for example, this approximation is seen in the QProp propeller design code by Drela [10]. Here, using an upper-bound fit to model profile drag means that downstream optimization using this model will be more robust to surrogate modeling error.

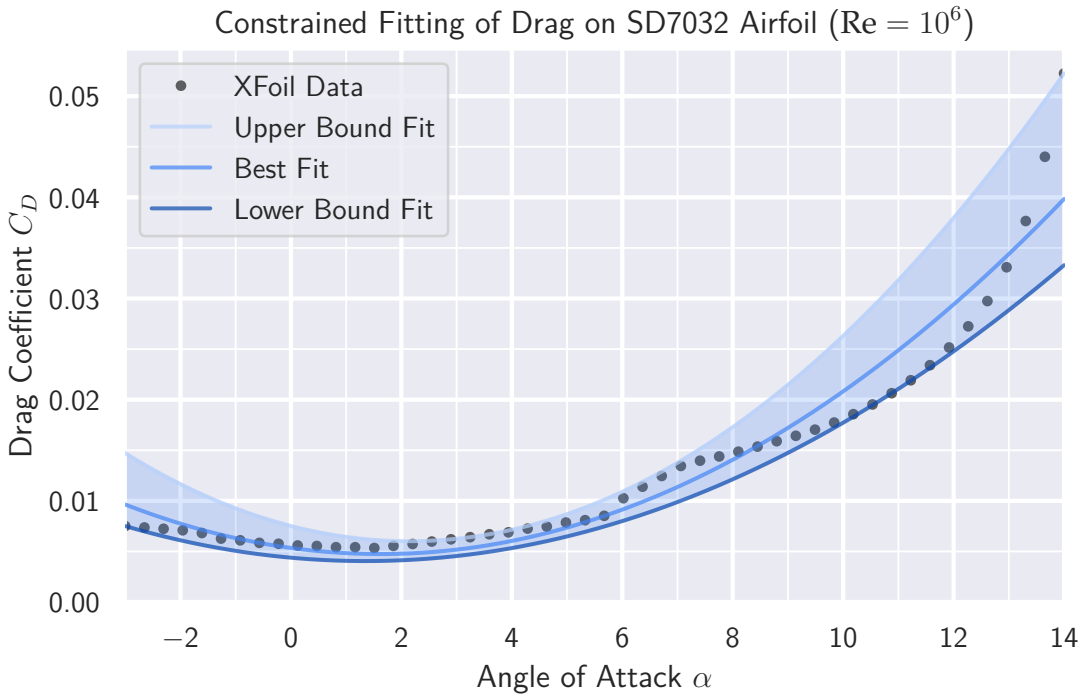

Figure 5-3: Robust fitting of an example drag polar with model bounds.

Log-transformed Errors

A final note here is that many outputs of engineering models are more physically relevant when considered in a log-transformed sense. In other words, the goodness of fit is a function of the relative (multiplicative) error rather than the absolute (additive) error.

A common example quantity that demonstrates this phenomenon is aerodynamic drag, which tends to approximately follow a power law with respect to Reynolds number. For illustration, we consider the case of the drag coefficient on a cylinder in crossflow. Experimental data plotted in Panton [35] (reproduced here in Fig. 5-4) found drag coefficients ranging from approximately 0.25 to 500, depending on Reynolds number. Indeed, in the $\mathrm{Re} \to 0$ (i.e. Stokes flow) limit, this drag coefficient exactly follows a power law and is unbounded[7].

[7]This is because the drag force D becomes linearly proportional to the freestream velocity U_∞ in the Stokes limit, rather than the usual $D \propto U_\infty^2$.

Using the AeroSandbox fitting module, a relatively parsimonious analytical model can be fit that accurately predicts (and extrapolates) cylinder drag for any Reynolds number. Surprisingly, the author has not found any other such universal model for cylinder drag in the literature.

Because of the logarithmic importance of drag coefficient, the fitting module `asb.FittedModel` is supplied with `put_residuals_in_logspace=True` at initialization. The resulting model, which minimizes log-transformed error with respect to the experimental dataset, is as follows:

Cylinder Drag Fitted Model

First, models for subcritical (i.e. below drag crisis) and supercritical drag are computed:

$$
\begin{aligned}
r &= \log_{10}(\text{Re}) \\
C_{D,\,\text{subcrit}} &= 10^{-0.6739r+1.0355} + 0.6325 + 0.1006r \\
\log_{10}(C_{D,\,\text{supercrit}}) &= -0.1200 - 0.04615 \ln\left[\exp\left(10 \cdot (6.7016 - r)\right) + 1\right]
\end{aligned}
$$

Then, these equations are blended together using a sigmoid:

$$
\begin{aligned}
\sigma(r) &= \frac{\tanh\left[12.597 \cdot (r - 5.547)\right] + 1}{2} \\
C_D &= \sigma(r) \cdot C_{D,\,\text{supercrit}} + \left[1 - \sigma(r)\right] \cdot C_{D,\,\text{subcrit}}
\end{aligned}
\tag{5.6}
$$

The fit of this model to the experimental data from [35] is shown in Figure 5-4.

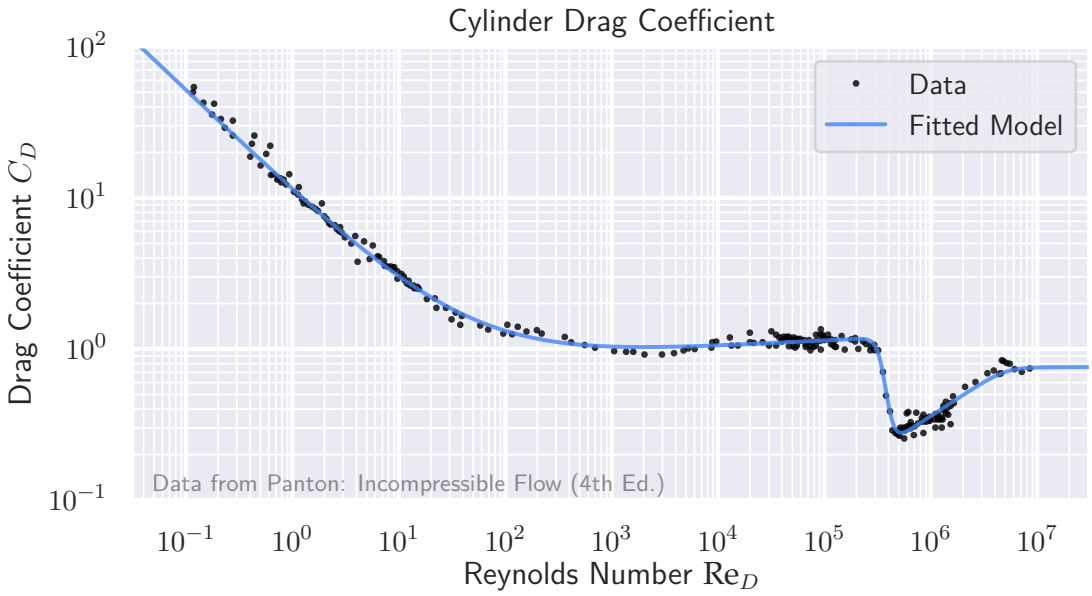

Figure 5-4: Analytical fitting of cylinder drag data from Panton [35]. Fit minimizes the L_1-norm of log-transformed error.

5.2.2 Interpolated Models

Another approach to creating surrogate models is interpolation. Interpolation forgoes the need for an analytic expression, instead representing the surrogate model in the form of a lookup table with rules for computing intermediate values. Interpolation is implemented in AeroSandbox via the syntax `asb.InterpolatedModel`, with inputs that are analogous to those for fitted models.

There are several challenges with interpolation that must be addressed in order to use an interpolated model in a differentiable optimization framework:

1. Interpolated models must be at least C_1-continuous, following the logic in Section 4.5.2. This means that many common interpolation techniques, such as linear interpolation, are not permissible. In AeroSandbox, this is solved by defaulting to a b-spline interpolation that consists of piecewise cubic polynomial patches; this is therefore a C_2-continuous representation and can be adequately treated with modern gradient-based optimization algorithms.

2. Interpolated models must extend to multidimensional datasets with an arbitrary number of inputs.

This combination of requirements is quite tricky to satisfy, and hence few underlying packages support this. For example, the SciPy library that forms the standard toolbox for scientific computing in Python only supports spline interpolation on 1D and 2D datasets [38]. Thankfully, the CasADi automatic differentiation library [3] includes routines that enable n-dimensional spline interpolation, so these have been wrapped for use in AeroSandbox.

These n-dimensional spline interpolators are demonstrated in Figure 5-5, where a synthetic dataset depicting lift of a SD7032 airfoil is turned into a surrogate model using the AeroSandbox interpolator library. Here, the C_2 continuity of the piecewise-cubic interpolation is clearly visible.

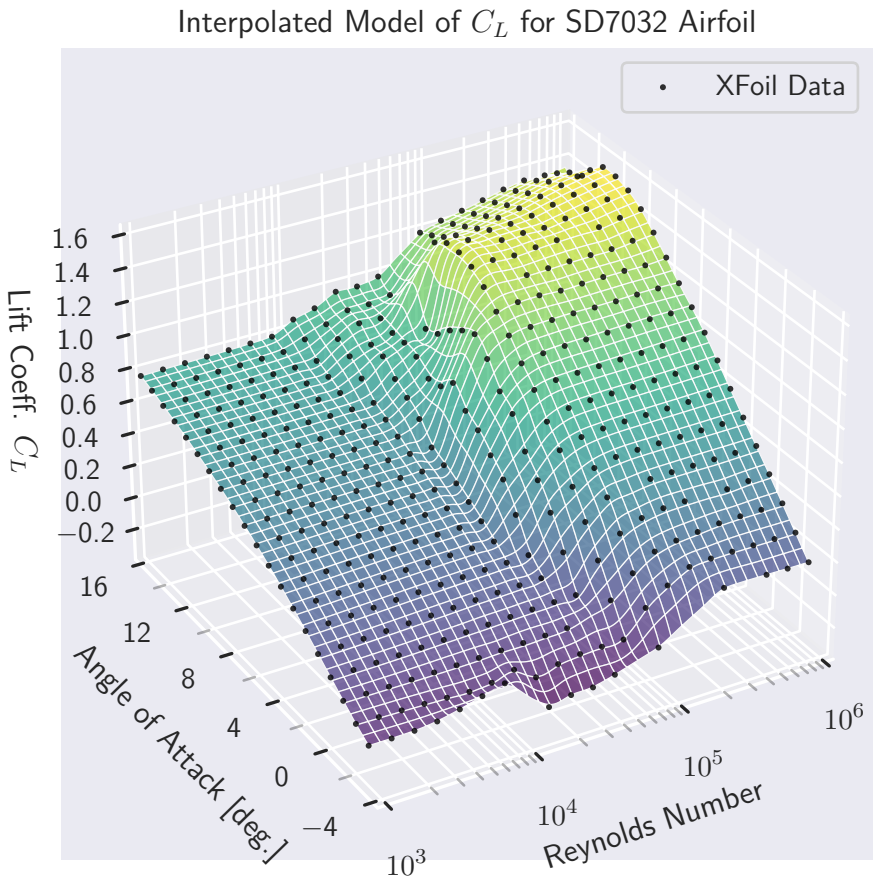

Figure 5-5: An interpolated model for airfoil C_L from a multidimensional dataset computed by XFoil [8].

This lookup-table approach to surrogate modeling is quite convenient to use when compared to the fitting approach described in Section 5.2.1. With a high-quality dataset, little-to-no engineering effort is need to create a surrogate model; by contrast, the fitting approach requires engineering intuition about the type of analytical model that best describes the data.

However, this interpolation approach is not without drawbacks. First, the b-spline interpolators implemented in AeroSandbox have no ability to reject noise - surrogate models will always pass exactly through input data points (as these form the knots of the spline). Splines also do not preserve monotonicity of the dataset:

on datasets with low signal-to-noise ratios[8], this can lead to wild oscillations as the interpolator attempts to model the random noise. This can even occur on datasets with moderate noise if the input space is not approprately sampled[9], a numerical problem known as Runge's phenomenon. Noisy datasets can also be effectively used if they are pre-processed by smoothing with several passes of a Laplacian (i.e. "heat equation") kernel. This process, sometimes referred to as the application of a *Gaussian blur*, yields more well-behaved interpolators at the cost of accuracy to the original dataset.

The lookup-table approach also necessitates the use of structured (i.e. gridded in n dimensions) data, rather than unstructured (i.e. "point cloud") data. This is because the interpolation consists of a collection of piecewise polynomials, and these polynomials must be linked back to specific knot points (drawn from the dataset) to determine their shape. With unstructured data, it is quite difficult to know which knot points should be used for a given polynomial.

Most synthetic datasets (e.g. those from expensive, high-fidelity computational analysis) tend to be structured, so this is not a problem. Conveniently, these datasets also tend to lend themselves well to interpolation as they generally have no random noise.

On the other hand, many experimental datasets tend to be unstructured, which prohibits the use of AeroSandbox interpopation as-is. Hence, fitting is generally preferred. Fitting also tends to be a superior choice for experimental datasets due to their better noise rejection.

In the event that interpolation is desired on an unstructured dataset, an approach that has been proven successful is to interpolate the unstructured dataset onto a structured grid using a radial basis function (RBF) interpolator, at which point the AeroSandbox spline interpolator in `asb.InterpolatedModel` can be used.

[8] In particular, this is an issue on some experimental datasets

[9] Often, typical linear full-factorial sampling is quite a poor choice; using Chebyshev nodes often produces far superior results for a given number of data points. A fuller description is given in Chapter 13 of [28].

Chapter 6

Discipline-Specific Models

6.1 Aerodynamics

6.1.1 3D Aerodynamics

6.1.2 2D Aerodynamics

6.2 Propulsion and Power Systems

6.3 Structures

6.4 Atmosphere and Wind

Chapter 7

Application: Firefly Micro-UAV

7.1 Requirements

7.2 Configuration

7.3 Design Code

7.4 Range Optimization Study

7.5 Understanding the Design Space

7.6 Multi-Objective Optimization and Pareto Efficiency

Chapter 8

Application: Dawn Solar UAV

8.1 Requirements

8.2 Configuration

8.3 Dawn Design Tool

8.4 Mass Budgets

8.5 Carpet Plots and Envelope Exploration

8.6 Dynamics

8.6.1 Cruise Dynamics

8.6.2 Ascent Dynamics

8.6.3 Design Sensitivities

Chapter 9

Conclusions

9.1 Summary of Contributions

9.1.1 Comparison to other Frameworks

GPKit

OpenMDAO

SUAVE

9.2 Future Work

Black-Box Functions and Finite Differencing

Investigations of Sparsity Exploitation Efficiency

Differentiation Frameworks and Numerical Backends

Assumption Tracking

N-Dimensional Interpolation of Unstructured Data

Higher-Order and Adaptive Integrators for Dynamics

Automatic Model Selection for Fitted Models

9.3 Final Notes

Appendix A

Installation Instructions and Basic Usage

AeroSandbox is very lightweight, and it can be installed on any machine running a modern operating system (Windows, Mac, or Linux) in just a few minutes using the following simple steps.

A.1 Installing Python

AeroSandbox is a Python package, so an installation of Python is required. AeroSandbox supports Python 3.7 and above, although the latest release is always recommended.

Python can be downloaded and installed from the `Python.org` website. However, is is recommended to instead install Python through the *Anaconda Distribution*, a distribution of Python that is popular in the scientific computing community and comes pre-packaged with many performance-optimized libraries [2]. The Anaconda Distribution is free and open-source under their "Individual Edition" and can be downloaded at `anaconda.com`.

When installing, it is recommended that you add the Anaconda Python executable (and all other executables in its `/bin/` binaries folder) to the system `PATH`. This can be done through a checkbox in the installer, or it can be performed manually

after installation.

Finally, correct installation can be verified by opening up a new terminal[1]. If Python has been installed correctly and added to the PATH, you will see a similar printout upon giving the command `python`:

```
C:\Users\User>python
Python 3.8.8 | packaged by conda-forge | (default, Feb 20 2021,
↪  15:50:08) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To return to the terminal, you can give the `quit()` command.

A.2 Installing AeroSandbox

Once Python has been installed, installation of AeroSandbox is very straightforward. Open up a new terminal window and give the following command:

```
pip install aerosandbox
```

Assuming the machine has a working internet connection, this will download AeroSandbox from the Python Package Index (PyPI) and install it. This concludes a normal installation.

A.3 Basic Usage

In order to use AeroSandbox, use a text editor or Python IDE[2] to create a Python script file[3]. Code the optimization problem that you would like to solve; an example is given in Listing 1. After saving the file, call the Python interpreter on this file by giving the following terminal command:

[1] On Windows, Command Prompt is one such terminal.
[2] PyCharm and VSCode being two popular choices
[3] `.py` file extension

```
python "path/to/my/file.py"
```

Alternatively, Python IDEs often have a ”Build” or ”Run File” command that can execute this code via the Python interpreter as well.

A.4 Developer Installation

For developers, the open-source AeroSandbox GitHub repository is available at `https://github.com/peterdsharpe/AeroSandbox`.

Any bugs and installation problems should also be reported via issue tickets at this GitHub repository.

A.5 Versioning

The version of AeroSandbox used in the code listings presented throughout this thesis is v3.x.x. Because of AeroSandbox's semantic versioning, compatibility is expected for all future AeroSandbox v3.x.x versions; however, this is not guaranteed. Please report any compatibility problems at the aforementioned GitHub repository.

Appendix B

Design Optimization Rules of Thumb

Here, we present several design optimization guidelines that have been collected across various studies. These are especially applicable to the conceptual design of engineering systems, but 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 more time-efficient to start low-fidelity and only increase fidelity as needed, rather than vice versa.
   (c) Identify sensitive models, requirements, and assumptions, and track the uncertainty associated with each of these. The vast majority of engineering time should be spent on refining these sensitive elements.
2. **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.

(b) In rare instances where high fidelity is truly required, surrogate modeling and reduced-order modeling is highly encouraged. 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.

3. **Do not blindly trust an optimizer.**

   (a) An optimizer only solves the problem that you give it. (And this is in the best case!) Often, it is easy to forget constraints that seem intuitively obvious.

   (b) If any flaw exists in a physics model, the optimizer will exploit it. Models should extrapolate sensibly and generally be parsimonious. On objective functions, adding quadratic regularization is an effective last resort.

   (c) Without special care, optimized designs are almost always fragile. An optimizer will tend to naturally drive to the edge of the feasible space. However, in nature[1], optima are usually not near extremes.

4. **When an incorrect result is returned, it's nearly always the "right solution to the wrong problem", rather than the "wrong solution to the right problem".**

   (a) If a strange solution, error, or indication of infeasiblity or unboundedness is reported when this was not expected, this often indicates an error in problem formulation. Common culprits are forgotten or unnecessary constraints, constraints that are unintentionally too loose or tight, duplicated constraints, etc.

   (b) If initial guesses or problem scales are off by many orders of magnitude, this can also cause convergence issues. Strong nonconvexities (e.g. a model interpolating noisy data) can also cause problems.

5. **Optimization is just one tool in the design toolbox.**

[1] Mother Nature being arguably the most successful optimizer

(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) Design is an interactive process, and a "push-button" design optimizer will not exist for the foreseeable future.

Appendix C

Addenda, Derivations, and Extended Code

C.1 Constrained Rosenbrock Problem

Here, we derive a closed-form solution to the constrained Rosenbrock problem, restated from Eq. 4.9 as:

$$\begin{aligned} \underset{x,\,y}{\text{minimize}} \quad & (1-x)^2 + 100(y-x^2)^2 \\ \text{subject to} \quad & x^2 + y^2 \leq 1 \end{aligned} \tag{C.1}$$

The objective function, which we denote $f(x, y)$ is shown in Figure C-1; the constraint limits the feasible region to the unit ball.

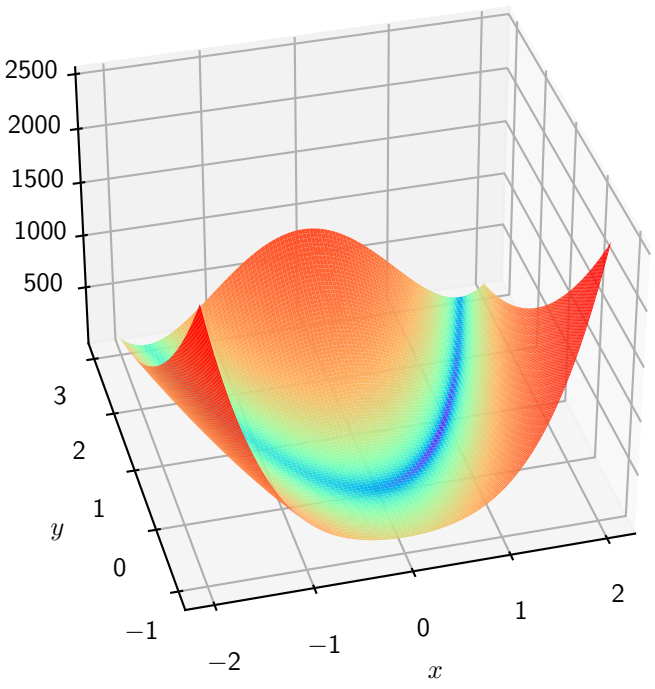

Figure C-1: Rosenbrock Function

We find the gradient of f to be:

$$\nabla f = \begin{bmatrix} \frac{\mathrm{d}f}{\mathrm{d}x} \\ \frac{\mathrm{d}f}{\mathrm{d}y} \end{bmatrix} = \begin{bmatrix} 400x(x^2 - y) + 2x - 2 \\ -200x^2 + 200y \end{bmatrix} \tag{C.2}$$

Ignoring the constraint to begin, we find the unconstrained critical points by setting $\nabla f = \vec{0}$. Thus, the $\frac{\mathrm{d}f}{\mathrm{d}y} = 0$ condition yields $x^2 = y$, which is substituted into the remaining equation to yield:

$$400x(y - y) + 2x - 2 = 0 \implies 2x - 2 = 0 \implies x = 1 \tag{C.3}$$

Given $x = 1$, the $\frac{\mathrm{d}f}{\mathrm{d}y} = 0$ conditions implies $y = 1$. Thus, $(x, y) = (1, 1)$ is the only critical point of the unconstrained problem.[1]

Thus, the only critical point for the unconstrained problem lies outside the

[1]Further analysis or graphical inspection reveals this to be the global minimum of the unconstrained problem.

feasible region. The complementary slackness optimality condition implies that criticality ($\nabla f = 0$ locally) is a requirement for optimality in the absence of an active constraint. Thus, we infer that the constrained optimum must lie on the constraint boundary and that the constraint have a nonzero associated dual variable.

We now prepare to formulate and directly solve optimality conditions including the constraint using the KKT conditions. We first rewrite the constraint in the form $g(x) \leq 0$, where here, $g(x) = x^2 + y^2 - 1$. The constraint gradient is then:

$$\nabla g = \begin{bmatrix} 2x \\ 2y \end{bmatrix} \tag{C.4}$$

Now, the optimality conditions can be formed. For the unknowns x, y, λ, we obtain a set of two equations from the stationarity requirement of the Lagrangian:

$$\nabla f + \lambda \nabla g(x) = \vec{0} \tag{C.5}$$

A third equation comes from the constraint itself, which we now know to be tight:

$$g(x) = 0 \tag{C.6}$$

This system of three equations does not readily admit an analytic solution, but a numerical solution can be easily obtained:

$$x = 0.7864, y = 0.6177, \lambda = 0.1215 \tag{C.7}$$

Finally, we can illustrate the poor scaling in this problem by evaluating the condition number of the Hessian at the optimum. The Hessian matrix of the objective function is found to be:

$$\mathbf{H} = \begin{bmatrix} \frac{\mathrm{d}^2 f}{\mathrm{d}x^2} & \frac{\mathrm{d}^2 f}{\mathrm{d}x\mathrm{d}y} \\ \frac{\mathrm{d}^2 f}{\mathrm{d}x\mathrm{d}y} & \frac{\mathrm{d}^2 f}{\mathrm{d}y^2} \end{bmatrix} = \begin{bmatrix} 1200x^2 - 400y + 2 & -400x \\ -400x & 200 \end{bmatrix} \tag{C.8}$$

Which, evaluated at the optimum of $(x, y) = (0.7864, 0.6177)$ yields:

$$\mathbf{H} \approx \begin{bmatrix} 497 & -315 \\ -315 & 200 \end{bmatrix} \tag{C.9}$$

After eigenvalue factorization, we evaluate the condition number to find:

$$\text{cond}(\mathbf{H}) \approx 1054 \tag{C.10}$$

C.2 Simple Wing

The Simple Wing problem described in Section 4.3.1 can be solved using the following AeroSandbox code:

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

### Constants
form_factor = 1.2  # form factor [-]
oswalds_efficiency = 0.95  # Oswald efficiency factor [-]
viscosity = 1.78e-5  # viscosity of air [kg/m/s]
density = 1.23  # density of air [kg/m^3]
airfoil_thickness_fraction = 0.12  # airfoil thickness to chord ratio [-]
ultimate_load_factor = 3.8  # ultimate load factor [-]
airspeed_takeoff = 22  # takeoff speed [m/s]
CL_max = 1.5  # max CL with flaps down [-]
wetted_area_ratio = 2.05  # wetted area ratio [-]
W_W_coeff1 = 8.71e-5  # Wing Weight Coefficient 1 [1/m]
W_W_coeff2 = 45.24  # Wing Weight Coefficient 2 [Pa]
drag_area_fuselage = 0.031  # fuselage drag area [m^2]
weight_fuselage = 4940.0  # aircraft weight excluding wing [N]

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

### Variables
aspect_ratio = opti.variable(init_guess=10)  # aspect ratio
wing_area = opti.variable(init_guess=10)  # total wing area [m^2]
airspeed = opti.variable(init_guess=100)  # cruising speed [m/s]
weight = opti.variable(init_guess=10000)  # total aircraft weight [N]
CL = opti.variable(init_guess=1)  # Lift coefficient of wing [-]

### Models
```

```
# Aerodynamics model
CD_fuselage = drag_area_fuselage / wing_area
Re = (density / viscosity) * airspeed * (wing_area / aspect_ratio) **
↪  0.5
Cf = 0.074 / Re ** 0.2
CD_profile = form_factor * Cf * wetted_area_ratio
CD_induced = CL ** 2 / (np.pi * aspect_ratio * oswalds_efficiency)
CD = CD_fuselage + CD_profile + CD_induced
dynamic_pressure = 0.5 * density * airspeed ** 2
drag = dynamic_pressure * wing_area * CD
lift_cruise = dynamic_pressure * wing_area * CL
lift_takeoff = 0.5 * density * wing_area * CL_max * airspeed_takeoff **
↪  2

# Wing weight model
weight_wing_structural = W_W_coeff1 * (
        ultimate_load_factor * aspect_ratio ** 1.5 *
        (weight_fuselage * weight * wing_area) ** 0.5
) / airfoil_thickness_fraction
weight_wing_surface = W_W_coeff2 * wing_area
weight_wing = weight_wing_surface + weight_wing_structural

### Constraints
opti.subject_to([
    weight <= lift_cruise,
    weight <= lift_takeoff,
    weight == weight_fuselage + weight_wing
])

### Objective
opti.minimize(drag)

sol = opti.solve()
```

C.3 Simple Aircraft (SimpleAC)

The Simple Aircraft (SimpleAC) problem described in Section 4.3.3 can be solved using the following AeroSandbox code:

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

opti = asb.Opti()
```

```
5
6   ### Env. constants
7   g = 9.81  # gravitational acceleration, m/s^2
8   mu = 1.775e-5  # viscosity of air, kg/m/s
9   rho = 1.23  # density of air, kg/m^3
10  rho_f = 817  # density of fuel, kg/m^3
11
12  ### Non-dimensional constants
13  C_Lmax = 1.6  # stall CL
14  e = 0.92  # Oswald's efficiency factor
15  k = 1.17  # form factor
16  N_ult = 3.3  # ultimate load factor
17  S_wetratio = 2.075  # wetted area ratio
18  tau = 0.12  # airfoil thickness to chord ratio
19  W_W_coeff1 = 2e-5  # wing weight coefficient 1
20  W_W_coeff2 = 60  # wing weight coefficient 2
21
22  ### Dimensional constants
23  Range = 1000e3  # aircraft range, m
24  TSFC = 0.6 / 3600  # thrust specific fuel consumption, 1/sec
25  V_min = 25  # takeoff speed, m/s
26  W_0 = 6250  # aircraft weight excluding wing, N
27
28  ### Free variables (same as SimPleAC, with extraneous variables removed)
29  AR = opti.variable(init_guess=10, log_transform=True)  # aspect ratio
30  S = opti.variable(init_guess=10, log_transform=True)  # total wing area,
    ↪  m^2
31  V = opti.variable(init_guess=100, log_transform=True)  # cruise speed,
    ↪  m/s
32  W = opti.variable(init_guess=10000, log_transform=True)  # total
    ↪  aircraft weight, N
33  C_L = opti.variable(init_guess=1, log_transform=True)  # lift
    ↪  coefficient
34  W_f = opti.variable(init_guess=3000, log_transform=True)  # fuel weight,
    ↪  N
35  V_f_fuse = opti.variable(init_guess=1, log_transform=True)  # fuel
    ↪  volume in the fuselage, m^3
36
37  ### Wing weight
38  W_w_surf = W_W_coeff2 * S
39  W_w_strc = W_W_coeff1 / tau * N_ult * AR ** 1.5 * np.sqrt(
40      (W_0 + V_f_fuse * g * rho_f) * W * S
41  )
42  W_w = W_w_surf + W_w_strc
43
44  ### Entire weight
45  opti.subject_to(W >= W_0 + W_w + W_f)
```

```
46
47  ### Lift equals weight constraint
48  opti.subject_to([
49      W_0 + W_w + 0.5 * W_f <= 0.5 * rho * S * C_L * V ** 2,
50      W <= 0.5 * rho * S * C_Lmax * V_min ** 2,
51  ])
52
53  ### Flight duration
54  T_flight = Range / V
55
56  ### Drag
57  Re = (rho / mu) * V * (S / AR) ** 0.5
58  C_f = 0.074 / Re ** 0.2
59
60  CDA0 = V_f_fuse / 10
61
62  C_D_fuse = CDA0 / S
63  C_D_wpar = k * C_f * S_wetratio
64  C_D_ind = C_L ** 2 / (np.pi * AR * e)
65  C_D = C_D_fuse + C_D_wpar + C_D_ind
66  D = 0.5 * rho * S * C_D * V ** 2
67
68  opti.subject_to(W_f >= TSFC * T_flight * D)
69
70  V_f = W_f / g / rho_f
71  V_f_wing = 0.03 * S ** 1.5 / AR ** 0.5 * tau
72
73  V_f_avail = V_f_wing + V_f_fuse
74
75  opti.subject_to(V_f_avail >= V_f)
76
77  opti.minimize(W_f)
78
79  sol = opti.solve()
```

C.4 Discontinuities at Non-Optimal Points

Continuing from Section 4.5.2, we note that discontinuity and non-differentiability is still admissible if it does not occur at the optimum. For example, we can pose the following problem, which has countably infinite non-continuous, non-differentiable points:

$$\underset{x}{\text{minimize}} \quad x \cdot \left\lfloor x + \frac{1}{2} \right\rfloor + \frac{1}{10}x^2 \tag{C.11}$$

where the objective function and its derivative are visualized in Figure C-2. This problem is easily solved as-written in AeroSandbox, because the region surrounding the optimum ($x = 0$, $f(x) = 0$) is locally C_1-continuous. Convergence is easily achieved even if the initial guess is one of these discontinuous points: with an initial guess of $x_0 = 10.5$, solution is achieved in just 6 iterations.

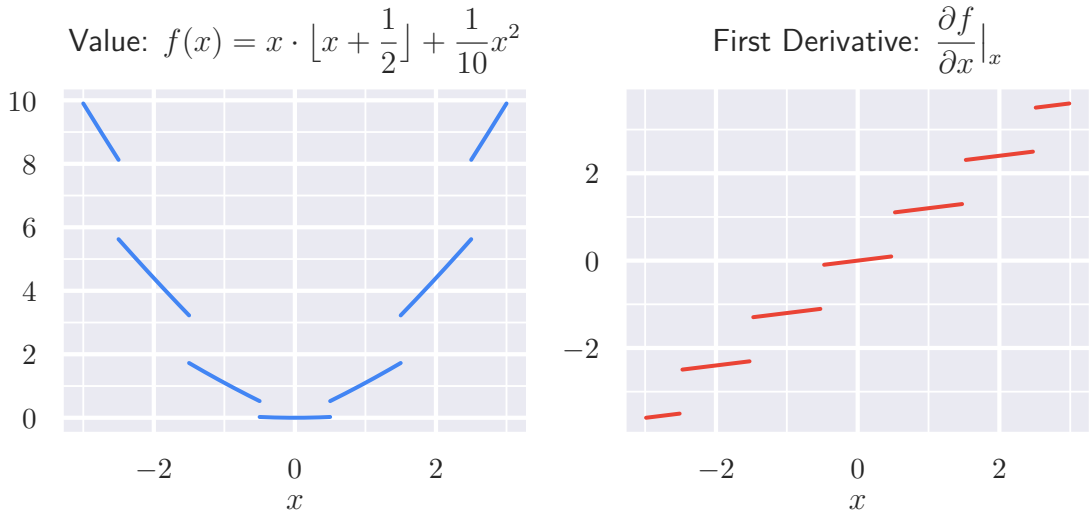

Figure C-2: Illustration of the function in Eq. C.11, which exhibits an infinite number of discontinuities in value and derivative.

Bibliography

[1] 787 airplane characteristics for airport planning, December 2015.

[2] Anaconda software distribution, 2020.

[3] Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1):1–36, 2019.

[4] John T. Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. 2nd edition, 2009.

[5] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization Problems*. Cambridge University Press, Cambridge, UK, 7th edition, 2004.

[6] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[7] John D Cook. Basic properties of the soft maximum. Technical report, 2011.

[8] Mark Drela. XFOIL: An Analysis and Design System for Low Reynolds Number Airfoils. *Low Reynolds Number Aerodynamics*, (54 , Berlin, Germany, Springer-Verlag, 1989):1–12, 1989.

[9] Mark Drela. Pros & Cons of Airfoil Optimization. In *Frontiers of Computational Fluid Dynamics 1998*. 1998.

[10] Mark Drela. QPROP Formulation. Technical report, Massachusetts Institute of Technology, Cambridge, MA, 2006.

[11] Mark Drela. Improved Performance Estimates for Optimization. Technical report, Massachusetts Institute of Technology, Cambridge, MA, 2009.

[12] Mark Drela. Design Drivers of Energy-Efficient Transport Aircraft. *SAE International Journal of Aerospace*, 4(2):602–618, 2011.

[13] Mark Drela. *Flight Vehicle Aerodynamics*. The MIT Press, Cambridge, MA, 2014.

[14] Mark Drela. TASOPT 2.15: Transport Aircraft System Optimization. Technical Description. Technical report, 2016.

[15] Mark Drela. *Aerodynamics of Viscous Fluids*. Massachusetts Institute of Technology, Cambridge, MA, draft edition, 2019.

[16] Thomas Durbin and CasADi Blog. On the importance of nlp scaling, Apr 2018.

[17] Louis Faury, Flavian Vasile, Clément Calauzènes, and Olivier Fercoq. Neural Generative Models for Global Optimization with Gradients. Technical report, 2018.

[18] Andreas Griewank. On Automatic Differentiation. Technical report, 1997.

[19] Raphael T Haftka. Simultaneous Analysis and Design. 23(7).

[20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[21] Hans Hersbach, Bill Bell, Paul Berrisford, Shoji Hirahara, András Horányi, Joaquín Muñoz-Sabater, Julien Nicolas, Carole Peubey, Raluca Radu, Dinand Schepers, Adrian Simmons, Cornel Soci, Saleh Abdalla, Xavier Abellan, Gianpaolo Balsamo, Peter Bechtold, Gionata Biavati, Jean Bidlot, Massimo Bonavita, Giovanna De Chiara, Per Dahlgren, Dick Dee, Michail Diamantakis, Rossana Dragani, Johannes Flemming, Richard Forbes, Manuel Fuentes, Alan Geer, Leo Haimberger, Sean Healy, Robin J. Hogan, Elías Hólm, Marta Janisková, Sarah Keeley, Patrick Laloyaux, Philippe Lopez, Cristina Lupu, Gabor Radnoti, Patricia de Rosnay, Iryna Rozum, Freja Vamborg, Sebastien Villaume, and Jean-Noël Thépaut. The era5 global reanalysis. *Quarterly Journal of the Royal Meteorological Society*, 146(730):1999–2049, 2020.

[22] Nick Higham. The top 10 algorithms in applied mathematics, Mar 2016.

[23] Warren Hoburg and Pieter Abbeel. Geometric Programming for Aircraft Design Optimization. *AIAA Journal*, 2014.

[24] Dipti Jasrasaria and Edward O Pyzer-Knapp. Dynamic Control of Explore/Exploit Trade-Off In Bayesian Optimization. Technical report, 2018.

[25] Matthew Kelly. An Introduction to Trajectory Optimization: How to Do Your Own Direct Collocation. *SIAM Review*, 59(4):849–904, 2017.

[26] Matthew P Kelly. Transcription Methods for Trajectory Optimization: A beginners tutorial. Technical report, 2015.

[27] Philippe Kirschen and Warren Hoburg. The power of log transformation: A comparison of geometric and signomial programming with general nonlinear programming techniques for aircraft design optimization. 2018.

[28] Mykel J. Kochenderfer and Tim A. Wheeler. *Algorithms for Optimization*. 2019.

[29] Schalk Kok and Carl Sandrock. Locating and Characterizing the Stationary Points of the Extended Rosenbrock Function. *Evolutionary Computation*, 17(3):437–453, 09 2009.

[30] Zhoujie Lyu, Zelu Xu, and Joaquim R R A Martins. Benchmarking Optimization Algorithms for Wing Aerodynamic Design Optimization.

[31] Joaquim R.R.A. Martins and Andrew Ning. *Engineering Design Optimization*. 2020.

[32] D A Masters, N J Taylor, T C S Rendall, C B Allen, and D J Poole. Geometric Comparison of Aerofoil Shape Parameterization Methods. 2016.

[33] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. 2006.

[34] Berk Ozturk. Conceptual Engineering Design and Optimization Methodologies using Geometric Programming. 2018.

[35] Ronald Lee Panton. *Incompressible Flow.* Wiley, 4th edition, 1984.

[36] Daniel P. Raymer. Aircraft Design: A Conceptual Approach (2nd Ed.).

[37] H. H. Rosenbrock. An Automatic Method for Finding the Greatest or Least Value of a Function. *The Computer Journal*, 3(3):175–184, 01 1960.

[38] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[39] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

[40] David H. Wolpert and William G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.