GPkit: a Human-Centered Approach to Convex Optimization in Engineering Design

Edward Burnell
Massachusetts Institute of Technology
Cambridge, MA, USA
eburn@mit.edu

Nicole B. Damen
University of Nebraska
Omaha, Nebraska, USA
ndamen@unomaha.edu

Warren Hoburg
National Aeronautics and Space Administration
Houston, TX, USA
whoburg@mit.edu

ABSTRACT
We present GPkit, a Python toolkit for Geometric and Signomial Programming that prioritizes explainability and incremental complexity. GPkit was designed through an ethnographic approach in the firms, classrooms, and research labs where it became part of the fabric of daily engineering work. Organizations have approached GPkit both in ways which centralize design work and in ways which distribute it, usecases which emerged from and inspired new toolkit features. This two-way flow between mathematical structure and practitioner knowledge resulted in several novel contributions to the formulation and interpretation of convex programs and to our understanding of early-stage engineering design. For example, dual solutions (often considered incidental) can be more valuable to a design process than the "optimal design" itself, and we present novel algorithms and design methods based on this insight.

Author Keywords
convex optimization; human-centered design; design models; geometric programming; modeling languages; toolkits

CCS Concepts
•**Human-centered computing** → User studies; *Usability testing;* **Collaborative interaction; User interface toolkits;** *Open source software;* Field studies; •**Theory of computation** → **Convex optimization;** •**Computing methodologies** → **Optimization algorithms;**

INTRODUCTION
Central to the work of many engineers is a "design model" that quantifies parameters of their designs and implements interactions amongst these parameters. Common types of design model include parameterized CAD assemblies which construct a shape from geometric constraints [35], spreadsheets which calculate performance [51, 42], and "design codes", small pieces of software which take in a desired performance and put out a design that achieves it [38].

CHI'20, April 25–30, 2020, Honolulu, HI, USA

© 2020 ACM. ISBN 978-1-4503-6708-0/20/04…$15.00

DOI: https://doi.org/10.1145/3313831.3376412

In engineering organizations design models often serve as loci for understanding *what* will be built, while encoding (and sometimes concealing) decisions on *why* [57, 35]. As such, design models are an important arena for intra-organizational design politics: it is often within and around these models that design participants' perspectives clash and coalesce [35, 6]. Any design model used by an organization has its outsiders and insiders, spectators and maintainers, and formal and informal power structures [35, 26].

Design models are subject to the agency and affordances of the "material" (e.g. Solidworks, Excel, FORTRAN) from which they are made [54, 34, 6, 39]. The influence of a design model's material is felt especially in a a design process' earliest stages, where the work is predominantly conceptual and lacks physical prototypes to reference [57, 21]. The effects of early-stage design model materials are typically examined through experimental outcomes [66, 67, 10], but some materials have not been studied in this way.

"Mathematical programs" are a type of design code, common in aerospace engineering, whose output achieves a desired performance while striving to minimize a chosen "cost" parameter [43]. Their research focuses on formulations, algorithms, and performance on benchmark models; published observations of effects on design and organizational outcomes are rare. This is doubly true for convex mathematical programs. Recent work (most notably by Boyd et al. and other contributors on open-source modeling toolkits [18, 13, 62] and solvers [15, 44]) has made convex programs more broadly accessible, but there are still few engineering organizations that use convex programs as a design model material.

This work sought to make convex programs more useful in engineering organizations and to study their effects as a material for early-stage design models. It did so by embracing a participatory human-centered framework focused on validating workers' knowledge [26] and using connections between that knowledge and the mathematics of optimization to develop GPkit, a toolkit for convex Geometric Programs. GPkit was used by many engineers and researchers during its development [68, 33, 46, 36, 16, 48, 27, 17, 49, 11, 28, 14, 1, 32, 60, 53, 64, 2], which resulted in features that are:

- technically and conceptually simple but with important user-experience benefits (e.g. using physical units to type-check symbolic equations),

- technically simple but conceptually complex (e.g. an object-oriented syntax for representing a set of "all potentially desirable airplane wings"),
- conceptually simple but technically complex (e.g. an algorithm for efficiently computing optimal performance tradeoffs with bounded error), and
- some both conceptually and technically complex (e.g. an algorithm for efficiently approximating a set of "all potentially desirable airplanes" with bounded error).

Organizations approached these features in ways both centralized (one firm-wide design model with a few developers) and distributed (design models developed independently by dozens of engineers from shared resources). This work's main contribution is our field observations of these outcomes alongside the toolkit features they emerged from and inspired.

BACKGROUND

This section contextualizes GPkit by comparing it to alternative software, design process concepts, and optimization techniques.

Usecases

Several in-depth comparisons between GPkit and alternative design model materials have been published by Ozturk, Burton, and Hoburg for an aerospace audience [48, 29, 11]. To summarize, in an aerospace design setting, GPkit's potential alternatives are programs written

1. by individual engineers to explore a conceptual design space (often in spreadsheets or scripting languages such as MATLAB or Python),
2. by subsystem teams to validate their decisions before a design review (often using a scripting language to wrap analysis codes implemented in C or FORTRAN), and
3. by system engineering teams to guide tradeoffs between subsystems (often using a library such as openMDAO [24] to connect black-box subsystem models).

GPkit has focused particularly on usecase (2), seeing the agency of design models in the rhetorical arena of a design review as important and underexplored.

Design Process

Frameworks for early stages of product and system design, such as Pahl and Beitz' [50] approach to engineering design and Ulrich, Eppinger and Yang's process for product design and development [63], are often based on the notion of an iterative and progressive design specification.

Following them we consider the formation of a design model as an iterative design process (Figure 1). First, the model is realized as a (Concept) in the (Modeler)'s mind, who formulates it as a (GPkit) model. Next, it is translated into the low-level representation needed by a numerical (Solver), and the convex optimization problem (or sequence of problems) is solved. GPkit then parses the returned solution and presents it back to the modeler. If the solution has any apparent errors then the modeler will seek to fix these errors in either their concept or

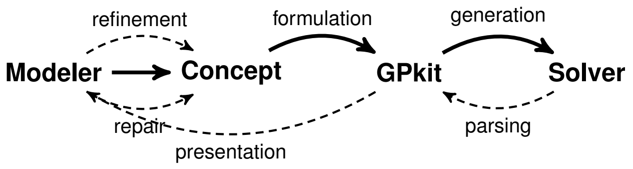


Figure 1. Schematic representation of the design modeling process. Dashed lines represent actions taken after a solution has been generated.

its formulation (the "repair" arrow). However, even without apparent errors the modeler is not finished, and will seek to refine their concept after having reflected upon its results (the "refinement" arrow) [59]. Either way the cycles continues for as long as the design model is used.

This iterative perspective emphasizes the importance of explainability [58] (presenting results with as much context and causality as possible) and incremental complexity (encouraging models to be made more complex one step at a time). Both principles are also important within an engineering organization, where repairs and refinements emerge through collaborative explanation and reformulation.

Optimization

Mathematical programs can be described as finding minima of a "cost" function $f(x)$ over a particular domain. For example, the minimum of $f(x) = x^2$ differs if the domain is "all real values of x" (Figure 2a) or "all real values of x greater than 0.5" (Figure 2b). This can be written generally as

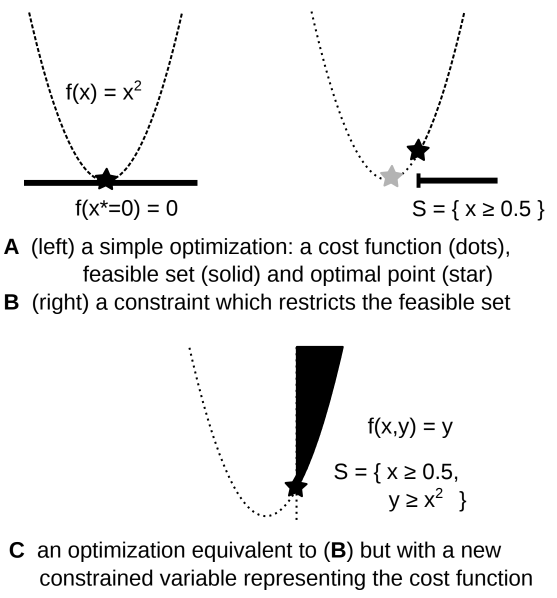
$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \mathrm{f}(x) \\ \text{subject to} \quad & x \in S \end{aligned} \tag{1}$$

where S is that optimization's "feasible set" [5]. S can be understood as the space of all potentially desirable answers: it is possible that any x in S might be an optimum, and the purpose of optimization is to return one which is. Each dimension of x is a "free variable" in the sense that it is not known before optimization, even though it may be heavily restricted by S.

This formulation affords both $f(x)$ and S equal importance, but it can be useful to move all complexity into the latter. For any optimization, a new free variable can be introduced as the cost (y in Figure 2c) and the original (potentially complex) cost function can be added to S as the inequality $f(x) \leq y$, forming a new feasible set $S^* = S \cap (f(x) \leq y)$. where the new "constraint" must also be met by all potentially desirable answers. This transformation (called the "epigraph problem form" [8]) allows us to consider the set of potentially desirable answers as the defining property of a mathematical program, fitting the engineering design intuition of "what a design space looks like depends on what you're going for".

Convex Optimization

An mathematical program is "convex" when S^* is convex, defined geometrically as "all points on any line between two points in S^* are themselves in S^*". If S^* is described by constraints, its convexity can also be derived from theirs: if each constraint represents a convex set, S^* is also convex. Certain forms of constraint have been established as convex, and much



A (left) a simple optimization: a cost function (dots), feasible set (solid) and optimal point (star)
B (right) a constraint which restricts the feasible set

C an optimization equivalent to (**B**) but with a new constrained variable representing the cost function

Figure 2. Visual representations of optimization.

research in convex optimization has been done on developing new forms or applying existing ones to engineering and operations problems [8].

Convexity can thus be considered a limitation on the types of constraints in a mathematical program. In return it provides benefits and guarantees for optimization algorithms. For example, it is straightforward to show that convex programs have only global optima. This is important because most algorithms for non-convex programs cannot make global claims about their solutions: instead of returning a point with the lowest possible cost, they often return a local optimum (an answer whose cost is not lowest in the entire set, just in a finite piece of it). Being able to guarantee that each design returned is a best-possible realization of their convex design model is immensely useful to those seeking to convince other design participants of their perspective.

Convexity also scales to large problems more readily than non-convex optimization. It is possible to solve convex optimization problems with thousands of free variables and constraints on a personal computer in seconds, while methods for non-convex optimization may struggle to solve in hours for merely dozens of free variables [8].

Strong Duality

Another property associated with convexity is strong Lagrange duality, by which a "primal program" can be transformed into a "dual program" whose optima correspond exactly. Dual programs optimize over free variables which each correspond to a primal constraint, and solving them gives the derivative of the optimal cost with respect to those constraints. That is, if the cost (as in Figure 2c) is equal to the variable y and every constraint in the primal is of the form $g_i(x) \leq c_i$ (where each c_i is a "fixed" variable not being optimized over), the dual solutions are equal to $\frac{dy}{dc_i}$. This interpretation of a dual solution's values as trade-offs between c_is and the overall cost has led them to be called "shadow prices" [8].

Geometric Programs

A Geometric Program is a type of convex program with strong Lagrange duality whose objectives and constraints are constructed of *posynomials*, which are expressions defined by

$$p(x) = \sum_{i=1}^{M} c_i \prod_{j=1}^{N} x_j^{a_{i,j}} \tag{2}$$

where each x_j is a free variable whose domain is real numbers greater than 0, each $a_{i,j}$ is a fixed real number, and each c is a fixed real number greater than 0. Geometric Programs are mathematical programs of the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & p_0(x_0, x_1 .., x_j) \\ \text{subject to} \quad & p_1(x_0, x_1 .., x_j) \leq 1, \\ & .., \\ & p_k(x_0, x_1 .., x_j) \leq 1 \end{aligned} \tag{3}$$

where $p_0, ..p_k$ are posynomials [7]. These posynomial constraints are not directly convex; rather, $\ln(p(e^z))$ is convex in z, and $x = e^z$ can be calculated after solving. Geometric Programs also be expressed in terms of exponential cone constraints [2].

Modeling Languages

For over 50 years Geometric Programs have been recognized as particularly suited to engineering equations [65]. The relatively recent introduction of optimization toolkits that do not require a background in numerical methods [23, 40] has made Geometric Programs accessible to a broader community. These toolkits, often called "modeling languages", abstract away the solvers behind convex optimization with syntaxes in which constraints can be written more "naturally". [18] Such a syntax reduces situations where the model received by the solver is not as the modeler intended [22], lowering the expertise barrier [23] for using convex optimization, allowing engineers to utilize the results of optimization researchers [62]. Unlike the assignment operator of a conventional programming language (such as $z = 0.5$) which directly assigns the results of a calculation to a variable, constraint specifications in a modeling language are represented only symbolically until the model is passed to a solver, at which point all free variables are simultaneously converged to an optimal point.

The Catastrophe of Delayed User Feedback

That free variables' values are not available in the code where they are declared or used but only after solving presents a user experience catastrophe [56]. When users make typos which render a constraint syntactically valid but substantively incorrect, they will not know of the error until the entire model is solved, and even then won't know which constraint caused it. Attempts to find the offending constraint via "printf debugging" [52] (placing print statements to isolate the error) falter because users have been abstracted away from (and may not have debugging experience with) the solver code in which those variables become numeric.

Field Site	Users	GPkit Design Models	Use Type
Class project with (under)graduate students	16	Electric “air taxi” service	*Mixed*
Aerospace R&D firm of 100-500 employees	12	Costing and sizing for airplane projects	*Distributed*
University researchers	10	Passenger jets, solar planes, other vehicles	*Distributed*
Industry-government-university initiative	10	Hybrid propulsion topologies	*Mixed*
Transportation startup of 150-250 employees	7	Pre-production system engineering	*Centralized*
Class project with (under)graduate students	3	Gas-powered high-altitude surveillance drone	*Centralized*

Table 1. Overview of field sites. The “Users” column is an approximation of active weekly users during the period of study.

Receiving a nonsensical “solution” after a simple typo and not being able to find its cause is the primary way we’ve seen optimization toolkits lose new users. Even for those patient enough to dig in and find the error it takes time to recover perceptions of the toolkit’s dependability. Like the static checks of a compiled programming language [4], any errors that can be raised during constraint specification, before solving, make repair iterations faster and more fluid.

Accessing Dual Solutions and Shadow Prices

Modeling languages for convex optimization typically make dual solutions accessible to users through either a vector of values or directly via the relevant constraint [23, 18], but surprisingly tend not to not suggest or encourage their use during model development.

From the perspective of the design process above, the dual solution can be incredibly useful. The relative importance of each constraint to a particular solution’s optimality is fully explained by the dual solution’s “shadow prices”. Constraints with a shadow price of zero could be removed without changing the solution at all, while those with surprisingly large or small shadow prices are likely erroneous, providing an opportunity for triage to speed repair iterations. In a “refinement” iteration, shadow prices can be interpreted as “model risk”, or the dependence of a design model’s conclusions on each of its assumptions. Interpreted in this way, a list of constraints ranked by the magnitude of their shadow price helps prioritize potential refinements.

In geometric programs, shadow prices can also be used to determine derivative of the cost with respect to each fixed variable (those which are not free to be changed during optimization) [29]. These can present even clearer feedback to a design participant than shadow prices: if a potentially complex design parameter such as a propeller’s efficiency has been modeled as a constant, but the cost is extremely dependent on this parameter, it may be worth developing a more complex model to refine the risk of that assumption being incorrect.

Disciplined Convex Programming (DCP)

Disciplined Convex Programming [23] is both a method for constructing certificates of a set’s convexity, and a syntax for formulating convex constraints. DCP works by creating a tree of operators with known conditions for convexity and checking if this tree can be formulated as convex. This allows non-convex constraints to raise errors during their creation, providing pre-solve feedback to users on the mathematical reasons a constraint is invalid. As discussed above, this kind of immediate feedback is scarce in a modeling language, making the “discipline” of Disciplined Convex Programming a powerful aid to a model’s development process.

METHODS

One of the best methods for gaining insight into user interactions is to observe users as they interact with systems of interest, but as noted by Olsen “simple usability testing is not adequate for evaluating complex systems” [45]. Exploring the consequences of convex design models would benefit most from expert users who are familiar enough with the toolkit to place a wide range of stringent demands on them [55]. However due to their novelty the vast majority of workers are novice users whose interactions will contain a mixture of genuine errors arising from intentional behavior, and false errors arising during familiarization. These learning behaviors of novice users are insightful for system design and development but muddy the waters when exploring consequences of convex design models for engineering organizations.

Additionally, while the agency [34] of a design model’s material is most apparent in people’s interactions with each other, i.e. organizational practices, [20], these are difficult to create in a more controlled environment. In a lab study participants are likely to be strangers, lack experience in collaborative design modeling, and have a limited period of interaction. Some participant groups in Breneman’s excellent study of collaborative satellite design show the scattered focus and poor outcomes that can occur in such contexts [3]. Similarly, lab experiments with convex optimization have shown with individual novice users that the underlying mathematics of a design model can affect design outcomes [10], but have struggled to show any change in this effect with pairs of novice users [37].

These considerations steered work on GPkit to proceed by recruiting users, working with them to understand the impact of GPkit on their design models, and then observing throughout continued engagement how this new design model material affected organizations and designs. Observations that informed GPkit were carried out across various research groups, classrooms, and industrial firms (see Table 1).

With users at each of these sites, the first author took field notes through in-person and digital observations [19], conducted informal usability tests, and worked with them to form shared understandings of their practice. This was done to gain insight into what they wanted to express when they made design models, the ways in which they expressed those with each material, and why those were their methods and goals.

GPkit’s development process also provided insight through material dialogues between the first author and users. Toolkit features often began as a way of asking a user “is this (syntax, algorithm) what you meant when you said that about your work?". In using that feature the user questioned often found an expressivity different from what they or the first author had expected, and communicated this in how they used and reflected on it [59]. This created a space in which each could share knowledge and build concepts for design work and convex programs that were interdependent, requiring the previous conceptions of each to change simultaneously [26]. Features which emerged from such spaces often felt spontaneous, each party feeling it wasn't really *their* idea, and were often the features most obviously responsible for organizational change.

RESULTS

Organizational Outcomes

All field sites (see Table 1) used GPkit to validate decisions presented in design reviews. In one case this was its exclusive use: two researchers were funded to construct a GPkit model by a firm hoping to convincingly validate its existing design. Field sites were split between using it in ways which centralized design work, in ways which distributed it, or both. The differences between sites reveal how GPkit may change engineers' interactions with their design models and each other, particularly by increasing the amount of direct feedback on modeling decisions.

Distributed Usecase

At the aerospace research and development firm, GPkit was primarily contrasted with an in-house MATLAB toolkit. Most design models made with either were developed by an individual modifying copies of others' code to represent their particular design prompt, testing their ideas for design opportunities, and then validating their conclusions in a series of team meetings. The firm trained dozens of conceptual design engineers to develop GPkit models and interactive demonstrations in hopes of creating a horizontal culture of sharing models and using them across new projects. They also developed a toolchain for interactive visualizations of GPkit models in presentations, with one designer describing his primary usecase as "leading an audience through novel design spaces". Through these presentations GPkit's introduction has increased group discussion of detailed design model decisions. Overall GPkit complements the MATLAB toolkit, which has a great deal of organizational trust but is less amenable than GPkit to extremely novel design spaces and interactive use during a presentation.

The increased discussion of detailed design model decisions was echoed when GPkit was introduced to academic contexts, where design presentations are typically larger, longer, and less frequent than at the aerospace firm. Academic use of GPkit has been also been distributed and propagated in similar ways, though with a wider variety of toolkits in competition or complement.

Centralized Usecase

GPkit has also lent itself to more centralized approaches, as shown by a transportation startup's use. There a small "systems engineering" team developed a GPkit design model, then established biweekly meetings with each subsystem team to build consensus on a particular subsystem's representation in that model. Updates to the system model and its current results results were then disseminated across teams on a monthly basis to build organizational consensus. The firm developed an internal website to interactively run the system model and encourage all design participants to explore its design space. Development of this system design model helped bridged a communication gap between higher-level managers and subsystem engineers that had been startling at our first site visit. At that time, subsystem engineers had said they felt unheard when they suggested practical but less glamorous or novel technologies, while managers said that too many engineers didn't realize the necessity of riskier technologies for their product category. System engineers, making models for the managers but eating lunch with the subsystem engineers, felt caught in between. The structure of the GPkit design model's development assured subsystem engineers that they could express the manufacturing and maintenance costs of risky technologies, while also reassuring managers that they could express the market and sales costs of conventional ones.

The first use of GPkit in an aerospace design classroom also centralized the work of design modeling. Several students took on the role of system engineers and built consensus with their peers to deviate from the requested solar-power architecture to a gas-powered one (see *Observation D* below for more details). Some of these students then reused their GPkit design code for manufacturing in the next semester's "build" class, through which the gas-powered surveillance drone proposed the previous semester was built and successfully flown (video at youtu.be/HMu3x5WxpeM). Teaching staff expressed surprise that the final build adhered so closely to the GPkit model's weight estimates.

Mixed Usecases

Other field sites mixed the centralized and decentralized usecases. In the design class after the gas-powered drone, GPkit adoption started with distributed use but became fairly centralized by the end of the semester. In the government-industry-academia initiative, some participants collaborated on a centralized design model while others developed individual models. During a mid-cycle progress review presentation, government officials expressed a preference for distributed models' dual-solution-based scaling laws over centralized models' complex comparisons of propulsor topologies.

Toolkit Features and Observations

This section discusses exemplary GPkit features and observations accompanying their development.

Observation A: Variable Declaration Comments

Comments that users added to their code were used to gain insight into their understanding of the design model. One repeat observation was of the comments added to variable declarations to provide additional information about that variable, for example "`S = Variable() # [ft**2] wing surface area`".

Feature A: Variable Metadata

While some modeling frameworks do not capture a variable's name as a code object [13], GPkit brought these variable metadata directly into the code: "`S = Variable('S',` `units='ft**2', label='wing surface area')`". Then, "paving the desire paths"[41], we used this information wherever possible.

Adding variable names and labels allowed GPkit to provide a default solution presentation that could be understood without referencing the original code (note S, its units, and its label in Figure 3), helping users more quickly see errors and giving them something which could be shown immediately to other design participants.

Incorporating variables' units and the corresponding unit algebra enabled errors to be raised whenever users attempted to add variables or create a constraint whose units were incompatible. As previously working models stopped solving, we heard some complaints. But unit checking provided immediate error feedback on constraints, and even the last holdouts came to actively appreciate it when it saved them from typos. Units also reduced metrication errors, removing the risk of changing a variable's unit. Units were still fairly new when the gas-powered drone mentioned above was proposed, and students spent an hour checking that the automatic unit conversions were being done correctly as they prepared to present to their client; later users rarely thought about such concerns.

Observation B: Modeling Gravity
During development of a (different) solar plane, gravity was modeled as a fixed variable g (instead of a united constant), and at "+4" was the most sensitive fixed variable. As a bit of a joke (and to get it to stop showing up in fixed-variable sensitivity tables), researchers modeled the slight difference in gravity at high altitude; to their surprise this lead to changes in their design.

Feature B: Sensitivity Tables
As part of the default presentation of results GPkit summarizes the dual solution (returned by the solver automatically), showing the "sensitivities" of fixed variables and constraints (note A and the fuel-burn constraints in Figure 3). As with the shadow prices of strong Lagrange duality, sensitivities correspond to $\frac{d\log(y)}{d\log(c_i)}$, where y is the cost and c_i is a fixed variable whose value is known before solving. This means that sensitivities correspond to a local power-law fit: in the situation above, the cost was locally proportional to g^4, so decreasing gravity by $\varepsilon\%$ decreased the cost by exactly $(4 \times \varepsilon)\%$ (for sufficiently small epsilon; sensitivities are only exactly true exactly at a solution). By showing these sensitivities to users, GPkit helps them repair erroneously large or small fixed-variable dependencies (which might not be clear in the primal solution) to prioritize refinements of their design model.

Observation C: Plotting Along
Solution tables are the atomic result of convex programs. The features mentioned above increase their usefulness for presenting to other design participants, but numerical plots showing multiple solutions at once are still more commonly used for that purpose, in part because of their rhetorical weight [47, 31]. The vast majority of user generated solutions are made specifically for plots comparing trade-offs between performance parameters, and most such visualizations use a conservative number of manually specified points on a rectangular grid in

```
Cost
----
 1.091 [lbf]

Free Variables
--------------
      | Aircraft
    W : 144.1                    [lbf]   weight

      | Aircraft.Wing
    S : 44.14                    [ft**2] surface area
    W : 44.14                    [lbf]   weight
    c : 1.279                    [ft]    mean chord

      | Mission.FlightSegment.AircraftP
Wburn : [ 0.274     0.272    ]  [lbf]   fuel burn
Wfuel : [ 1.09      0.272    ]  [lbf]   fuel weight

      | Mission.FlightSegment.AircraftP.WingAero
    D : [ 2.74      2.72     ]  [lbf]   drag force

Sensitivities
-------------
    | Aircraft.Fuselage
  W : +0.97  weight

    | Aircraft.Wing
  A : -0.67  aspect ratio
rho : +0.43  areal density

Tightest Constraints
--------------------
       | Aircraft
  +1.4 : .W >= .Fuselage.W + .Wing.W

       | Mission
    +1 : Wfuel[0] >= Wfuel[1] + Wburn[0]
 +0.75 : Wfuel[1] >= Wfuel[2] + Wburn[1]
  +0.5 : Wfuel[2] >= Wfuel[3] + Wburn[2]

       | Aircraft.Wing
 +0.43 : .W >= S*.rho
```

Figure 3. Example of a results table in GPkit.

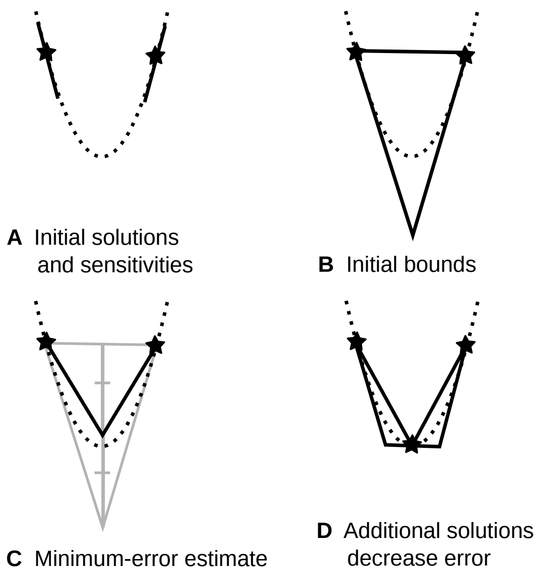

A Initial solutions and sensitivities
B Initial bounds
C Minimum-error estimate
D Additional solutions decrease error

Figure 4. Visual explanation of the Autosweep algorithm.

an attempt to avoid inaccuracy and perceptible discontinuities between solution points.

Feature C: Autosweep

For plots in which the independent axes are fixed variables and the only dependent axis is cost we can solve for it with optimal efficiency through the use of dual solutions. Because of convexity, variable sensitivities at corners of the grid define hyperplanes which lower-bound the cost inside, and the hyperplanes constructible from those corners upper bound it. Any solved points are exact, so these upper and lower bounds intersect at those corners. Figure 4 illustrates this in two dimensions with one independent fixed variable as the horizontal axis and the dotted parabola representing the (unknown by the algorithm) optimal cost. Averaging upper and lower bounds results in an estimate whose inaccuracy is equal to half the vertical distance between upper and lower bounds. If this isn't accurate enough, the estimate can be optimally improved by solving at points of maximum vertical distance, iterating until the desired accuracy is achieved. Note that any straight or planar portions of the cost function are completely described by just their boundary points. This algorithm needs very few solutions to describe predictable portions of the design space. Between its improved solution placement and the ability for users to directly specify their desired accuracy this algorithm is faster and more precise than naive gridding. Because of its iterative nature it can also be used as an online algorithm for real-time design space exploration.

Observation D: Shifting Costs and Constraints

As mentioned in *Centralized Usecase* above, GPkit was used in an aerospace design classroom where the "client" requested a solar-powered surveillance drone that could loiter for as long as possible at high altitude above a target location. Students debated this specification, realizing the airplanes their model returned were large enough to present transportation and deployment difficulties. Could it make sense to consider architectures other than solar power? After some consternation at going against the client's instructions, they developed a model for propulsion by a gasoline engine and changed their cost function, solving instead for the lightest gas-powered plane that could loiter for at least 6 days.

To the surprise of teaching staff an optimized gas-powered architecture was both capable of 6 days endurance and much more amenable to manufacturing. The underlying design model stayed mostly the same, changing only its cost function and propulsion and fuel models, which helped the client understand the tradeoffs and fully buy in to the new architecture.

Feature D: Modularity and Convexity

What came out of this observation was not just GPkit syntax but an appreciation for the synergy between convex modeling and engineering practice.

By asking users about how they used cost functions, we learned that their value was not in returning the "best result", but rather in getting a better understanding of the set of potentially desirable designs by collapsing it along a particular axis. We observed engineers optimizing for multiple performance parameters, each generating a "paragon" optimal by that metric [49]. Comparison of these paragons gave a sense of possibilities for their design [48]. Sometimes even a singular and quantifiable goal (such as the expected profit of a product), was thanklessly complex to represent, leading engineers to a preference for discussing paragons. While many mathematical programs' structure or method of solving is built around the cost function, in convex design models we have often found that *all* performance parameters are viable cost functions. As a result, after the events above occurred we standardized setting cost functions as the last pre-solve step.

We also found that because constraints can represent equations describing a design's function as well as those describing its form, constraints and entire models (e.g. an aircraft's wing spar) are often reusable across projects.

Observation E: Where Constraints "Belong"

We repeatedly heard GPkit users refer to where constraints "belonged". This was done explicitly in "#TODO" comments apologizing for a constraint's misplacement, in "#NOTE" comments justifying its presence, and in design discussions ("does the fuel burn constraint belong here [with the fuel model] or there [with the airplane drag constraints]?"). It was also done implicitly, with "belonging" represented by the clustering of constraints into named categories. Constraints were generally described as "belonging" either to categories arising from a hierarchy of the design's physical layout (`wing`, `fuselage`) or from the disciplinary hierarchy (`aerodynamics_eqs`, `structural_eqs`) common in non-convex aerospace design codes.

Feature E.1: Named Object-Oriented Constraint Sets

We found that such categories could be well represented as a tree of custom objects; for example, an Airplane instance containing Wing and Fuselage instances. Although most users had not written object-oriented code prior to GPkit, adoption of this feature was surprisingly rapid; the object-oriented

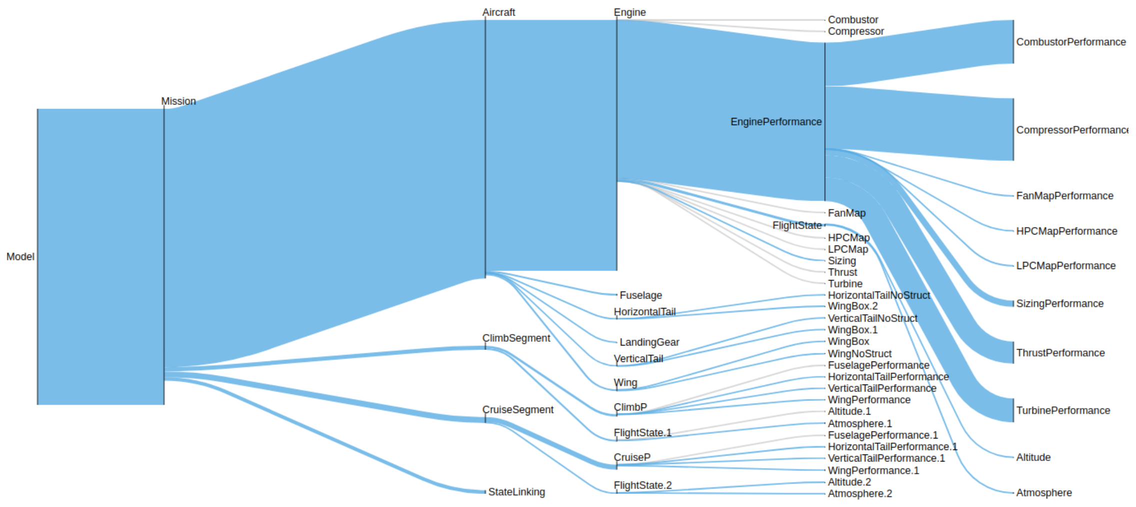


Figure 5. Sankey diagram displaying a model's constraint tree (left is towards the root) and the sensitivity of each named constraint set object (thicker is more sensitive). Discontinuous increases in thickness at a particular object indicate the sensitivity of the unnamed constraints inside it.

framework seemed to fit their experience and intuition, and introduction of a hierarchy lowered the number of variables available in each constraint's and made the interlinking of subsystems more repairable and explainable. With this new layer of abstraction, engineers felt more comfortable repairing, refining and explaining design models spanning multiple files, and began creating "stub models" with fixed variables and no constraints to be refined later if the model was sensitive to one of those variables [48].

Feature E.2: Constraint Set Boundedness Verification

These objects (Wing, Fuselage) provided another way to check constraint's correctness before solving, by asking users to specify which variables they expected to be upper or lower unbounded by instances of that object class. For example, a WingStructure instance might be expected to not put a lower bound on b (representing the wing's length), because a vanishingly short wing structure would be perfect in its weightlessness. However WingStructure ought to upper bound b, because a too-long wing would bend and snap. If a WingStructure instance imposed a lower bound or lacked an upper bound on b, GPkit would warn the user and suggest they either repair their model or refine their expectations for WingStructure. Because variables in a model are typically both upper and lower bounded (if they're not the model is generally infeasible), we explained these bounds checks to users by relating them to a jigsaw puzzle where objects' upper-bounded "tabs" and lower-bounded "notches" fit together to form a design model without any unmatched tabs or notches.

Observation F: "An Engine With a Stick Attached"

A researcher in the industry-government-university initiative came to us with a frustrated certainty that the jet engine component of their design model was "too brittle": "The plane is being entirely designed around the engine! [...] changing the plane model doesn't change the optimal engine at all; changing the optimal engine model completely changes the plane. It's like an engine with a stick attached." Part of their frustration was that, while *they* had become certain of this by solving for various points, they did not know how to convince their colleague who was developing the engine model.

Feature F: Sankey Diagrams of the Dual Solution

By summing all constraint sensitivities in a named constraint set we calculate the effect changing those variables simultaneously would have on cost. The diagram in Figure 5 shows both the hierarchy of a model (see *Feature E.1*) and the relative sensitivities of each named constraint set. For the engineer who'd come to us, seeing this exact diagram felt like a vindication; here was clear indication of their design model's sensitivity to the engine to the exclusion of the rest of the airplane. After further finding that the basic lift constraint in the Airplane object was responsible for most of the rest of its sensitivity, their initial heated description of their design model as "an engine with a [lift-providing] stick attached" seemed quite apt.

We also developed Sankey diagrams for variables, showing the portion of a variable's total sensitivity coming from each constraint. Free variables always have zero sensitivity (if they'd be better at a higher or lower value, that would by definition be their optimal value), but seeing how positive and negative sensitivities *across* constraints sum to that zero explains which constraints are responsible for that variable's value. This is particularly helpful if a free variable's optimal value is either unexpected (repair cycle) or part of a design debate (refinement cycle).

Observation G: Shared Tooling

Early in the gas-powered-drone class project, the model found a curious way of flying; accelerating as it increased in altitude, but unexpectedly keeping the same speed even as it spent fuel and became lighter. The culprit turned out to be one variable, the Reynolds number (a non-dimensional characterization of aerodynamics), which had been incorrectly specified as a scalar rather than a vector variable. The model was thus, correctly, trying to find an optimal single Reynolds number it could maintain across all modes of flight, making the plane's airspeed a function only of its altitude. Years later, the idea of this plane striving to achieve a perfectly consistent Reynolds number is still considered extremely comedic by some of the original teaching staff.

Feature G: Vectorization Environments

To make such mistakes less likely, we built upon Tony Tao's work [61] exploring simultaneous optimization of a fleet of airplanes. Instead of designing one plane for multiple flight states (e.g. climbing, cruising, landing), he modeled one set of manufacturing tools shared by multiple airplanes, each with multiple flightstates, thus adding another dimension to a typical "multi-point" design problem. In GPkit we introduced vectorization environments to add a dimension to variables in any subtree of the constraint hierarchy. This made adding such abstractions straightforward; what had been one airplane and flight state could, with a few lines of code, become multiple flight states which shared an airplane, and with another few lines multiple planes with shared tooling. As with object-oriented constraint sets, the introduction of this method encouraged users to make significantly larger models. Vectorization made adding new modes of flight, new missions, and new structural loading cases much simpler, though it required care in model construction to ensure variables were partitioned to permit the desired vectorization. On one project, the Wing model had to be split into WingStructure (which stayed in Airplane) and WingAero (which went into a new AirplanePerformance object) so that a single Airplane could have a vectorized AirplanePerformance for each of its flight states.

Observation H: Flight Envelopes

While the concept of a feasible set originates in optimization, we have found it quite intuitive to engineers. A graduate student using GPkit once explained to us that they did not wish to specify an objective function because they did not want to solve for a particular point. Rather, they wanted to solve for all designs which fit their requirements. Instead of building a design model to ask "at what altitude and speed does a single-bladed helicopter fly *best*?", they wanted to ask "at what altitudes and speeds can a single-bladed helicopter be flown *at all*?".

Feature H: Design Space Approximation

Because of this request, GPkit has an efficient design space approximation algorithm, similar to that for autosweep. By selecting a simplex in the given dimensionality (a triangle in 2D; see Figure 6), we can solve from each corner for the nearest edge of the feasible set. Because the feasible set must be convex, the simplex formed by these feasible points is an inner approximation of the feasible set, while the area contained by the perpendicular tangents of those points is an outer one. As with autosweep, averaging the two forms an estimate with minimum inaccuracy, reducable as desired by additional solves at the most inaccurate points. If solving a point outwards returns a certificate of dual infeasibility the design space is unbounded in that direction, and because of convexity that direction will be unbounded for all points.

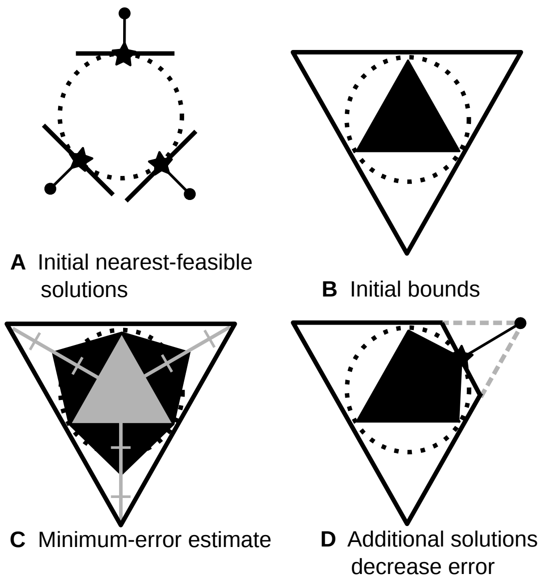

A Initial nearest-feasible solutions
B Initial bounds
C Minimum-error estimate
D Additional solutions decrease error

Figure 6. Visual representation of design space approximation.

DISCUSSION

In this section we will explore how this work fits within the HCI literature, but first we should discuss its limitations [12]. Convex programs are fundamentally limited as a material for design models, and this is a limited study of a particular toolkit in a few field sites. While our methods produced a wealth of information and insights that could otherwise not have been obtained they are sensitive to selection biases, and our ethnographic methods were not rigorously planned but often emerged from necessity or opportunity. To alleviate issues of physical distance the we made an effort to provide up-to-date documentation and discussions online, but not everyone was equally able or willing to communicate their experiences back to the first author, which marginalized them in the development process. Further many of the above features (unit analysis, variable metadata, sensitivity analysis) are present in other programming languages or engineering design toolkits, though as far as we know they are novel to convex optimization toolkits.

GPkit as a Toolkit

Toolkits have been defined by Greenberg as providing a "vocabulary and set of building blocks" which "give people a 'language' to think about these new interfaces, which in turn allows them to concentrate on creative designs." [25]. Ledo et al. [12] extended this to define toolkits as "generative platforms designed to create new interactive artifacts, provide easy access to complex algorithms, enable fast prototyping of software and hardware interfaces, or enable creative exploration of design spaces". We have done several of these with GPkit, giving engineers the ability to rapidly build and improve interactive explorations of convex design spaces. GPkit is thus an "artifact contribution", where "new knowledge is embedded in and manifested by artifacts and the supporting materials that describe them", as demonstrated via the case studies and observations above [12].

GPkit as a User Interface System

This work also follows in the footsteps of other user interface systems [45]. GPkit falls under each of the three limitations of usability experiments, being a tool for expert users doing non-standard tasks over a time period of months or years. As with previous papers using this framework [30] we consider GPkit in Olsen's framework for User Interface Systems [45]: it introduces a new task to a preexisting group of users (engineering designers) in a preexisting situation (early stage design), but (unlike in Olsen), GPkit also seeks to change that situation, changing the ways in which computers are used in these organizations as part of the its development.

Value Added by UI Systems Architecture

GPkit sets up "paths of least resistance" [45, 9] for explaining design codes; this results in models that can be more amenable to iterative improvement than alternatives, allowing users to derive related solutions with reduced development viscosity and build on common infrastructure. Convex programs have been inaccessible to most engineering organizations, but by making them accessible, GPkit helps increase the scale of their design models from dozens of free variables to thousands. GPkit thus lowers the threshold and raises the ceiling for design codes. The "Moving Targets" present in any use-motivated work [9] have been a fuel for the development of GPkit; we have encouraged our users to change their goals and expectations as they use it so they might adopt previously untenable organizational practices.

CONCLUSION

Much can be gained by integrating user experience and the mathematics of convexity. Our original goal was to understand and develop convex optimization as a material for engineering design models, so we adopted an explicitly human-centered framework and attempted to align practitioners' knowledge of their work with the way they used GPkit. We found convex design models to be powerful tools that can contain the hopes and concerns of many design participants. In this paper we have presented several of the novel algorithms (such as design space approximation and autosweep) and visualizations (such as Sankey sensitivity diagrams) that resulted from this approach. We have also developed insights into the user experience of design models and the methods by which users iteratively repair and refine them. The utility and ease of use of GPkit has led it to be woven into the fabric of several engineering organizations, changing their practices of engineering design. By drawing from a breadth of research areas and practitioner experiences, it has also opened up new avenues for HCI research in design models, convex optimization, and organizational outcomes.

ACKNOWLEDGEMENTS

This study was supported in part by an award by the MIT-Sensetime Fund and the National Science Foundation under grant #1854833. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funders.

EB: Without the creative ferment of the Convex Engineering Group this work could never have happened; cheers to Cody Archer, Arthur Brown, Michael Burton, Lochie Ferrier, Philippe Kirschen, Berk Ozturk, and Ali Saab. Shouts out to David Anderson, who started with me on this research direction while we were undergraduates, and to Maria Yang, who gave me a place from which to start again. To Ambika Kamath, whose reminder that "There's room for solidity and for ambition" kept me going. Thanks also to those who handled early drafts, some already listed but including also Dr. Hong-en Chen and Katy Gero, whose questions and critiques carried us over the finish line. A particularly special thanks goes to the members of my family (Nevin, Michael, and Nevin) who with patient exasperation tolerated my lack of foresight as I finished this paper's first submission amidst Yellowstone campsites.

Finally, we would like to acknowledge our kindly critical reviewers, whose efforts in understanding our oddball first submission and suggesting improvements made a stark difference in its quality.

REFERENCES

[1] Sara Achour and Martin Rinard. 2018. Time Dilation and Contraction for Programmable Analog Devices with Jaunt. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 229–242.

[2] Akshay Agrawal, Steven Diamond, and Stephen Boyd. 2019. Disciplined geometric programming. *Optimization Letters* (2019), 1–16.

[3] Jesse Austin-Breneman, Tomonori Honda, and Maria C Yang. 2012. A study of student design team behaviors in complex system design. *Journal of Mechanical Design* 134, 12 (2012), 124504.

[4] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008), 22–29.

[5] Brian Beavis and Ian Dobbs. 1990. *Optimisation and stability theory for economic analysis*. Cambridge university press.

[6] Daniel Beunza and David Stark. 2004. Tools of the trade: the socio-technology of arbitrage in a Wall Street trading room. *Industrial and corporate change* 13, 2 (2004), 369–400.

[7] Stephen Boyd, Seung-Jean Kim, Lieven Vandenberghe, and Arash Hassibi. 2007. A tutorial on geometric programming. *Optimization and Engineering* (2007).

[8] Stephen Boyd and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press, New York, NY, USA.

[9] Randy Pausch Brad Myers, Scott E Hudson. 2000. Past, present, and future of user interface software tools. 7 (2000), 3–28. Issue 1. DOI: http://dx.doi.org/10.1145/344949.344959

[10] Edward Burnell, Michael Stern, Ana Flooks, and Maria C Yang. 2017. Integrating Design and Optimization Tools: A Designer Centered Study. In *ASME 2017 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers Digital Collection.

[11] Michael J Burton and Warren W Hoburg. 2017. Solar-Electric and Gas Powered, Long-Endurance UAV Sizing via Geometric Programming. In *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. 4147.

[12] Jo Vermeulen Nicolai Marquardt Lora Oehlberg David Ledo, Steven Houben and Saul Greenberg. 2018. Evaluation strategies for HCI toolkit research. In *ACM SIGCHI Conference on Human Factors in Computing Systems*. ACM, 18.

[13] Steven Diamond and Stephen Boyd. 2016. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research* 17, 83 (2016), 1–5.

[14] Luis Diez, George-Pantelimon Popescu, and Ramón Agüero. 2016. A geometric programming solution for the mutual-interference model in hetnets. *IEEE Communications Letters* 20, 9 (2016), 1876–1879.

[15] Alexander Domahidi, Eric Chu, and Stephen Boyd. 2013. ECOS: An SOCP solver for embedded systems. In *2013 European Control Conference (ECC)*. IEEE, 3071–3076.

[16] Aidan Dowdle and Marija Ilić. 2017. Interconnected state-space modeling for turboelectric aircraft. In *2017 North American Power Symposium (NAPS)*. IEEE, 1–6.

[17] Aidan P Dowdle, David K Hall, and Jeffrey H Lang. 2018. Electric Propulsion Architecture Assessment via Signomial Programming. In *2018 AIAA/IEEE Electric Aircraft Technologies Symposium (EATS)*. IEEE, 1–21.

[18] Iain Dunning, Joey Huchette, and Miles Lubin. 2015. JuMP: A modeling language for mathematical optimization. *arXiv preprint arXiv:1508.01982* (2015).

[19] Robert M Emerson, Rachel I Fretz, and Linda L Shaw. 2011. *Writing ethnographic fieldnotes*. University of Chicago Press.

[20] Martha S Feldman and Wanda J Orlikowski. 2011. Theorizing practice and practicing theory. *Organization science* 22, 5 (2011), 1240–1253.

[21] Sebastian K Fixson and Tucker J Marion. 2012. Back-loading: A potential side effect of employing digital design tools in new product development. *Journal of Product Innovation Management* 29 (2012), 140–156.

[22] Robert Fourer, David M Gay, and Brian W Kernighan. 1987. *AMPL: A mathematical programming language*. Citeseer.

[23] Michael Grant, Stephen Boyd, and Yinyu Ye. 2006. Disciplined convex programming. In *Global optimization*. Springer, 155–210.

[24] Justin Gray, Kenneth Moore, and Bret Naylor. 2010. OpenMDAO: An open source framework for multidisciplinary analysis and optimization. In *13th AIAA/ISSMO Multidisciplinary Analysis Optimization Conference*. 9101.

[25] S. Greenberg. 2007. Toolkits and interface creativity. *Multimedia Tools and Applications)* 32 (2007), 139 –159. Issue 2. DOI: http://dx.doi.org/10.1007/s11042-006-0062-y

[26] Davydd J Greenwood and Morten Levin. 2006. *Introduction to action research: Social research for social change*. SAGE publications.

[27] David K Hall, Aidan Dowdle, Jonas Gonzalez, Lauren Trollinger, and William Thalheimer. 2018. Assessment of a Boundary Layer Ingesting Turboelectric Aircraft Configuration using Signomial Programming. In *2018 Aviation Technology, Integration, and Operations Conference*. 3973.

[28] Monowar Hasan, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B Bobba. 2018. A design-space exploration for allocating security tasks in multicore real-time systems. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 225–230.

[29] Warren Woodrow Hoburg. 2013. *Aircraft design optimization as a geometric program*. Ph.D. Dissertation. University of California, Berkeley.

[30] & Marquardt N. Houben, S. 2015. Watchconnect: A toolkit for prototyping smartwatch-centric cross-device applications. In *ACM Conference on Human Factors in Computing Systems*. ACM, 1247–1256.

[31] Sarah Kaplan. 2011. Strategy and PowerPoint: An inquiry into the epistemic culture and machinery of strategy making. *Organization Science* 22, 2 (2011), 320–346.

[32] Pasha Khosravi, Yitao Liang, YooJung Choi, and Guy Van den Broeck. 2019. What to Expect of Classifiers? Reasoning about Logistic Regression with Missing Features. *arXiv preprint arXiv:1903.01620* (2019).

[33] Philippe G Kirschen, Edward Burnell, and Warren Hoburg. 2016. Signomial programming models for aircraft design. In *54th AIAA Aerospace Sciences Meeting*. 2003.

[34] Bruno Latour. 1992. Where are the missing masses, the sociology of mundane artefacts. *Bijerker, WE & Law, J.(1992). Eds., Shaping Technology/Building Society: Studies in Sociotechnical Change. MIT Press, Cambridge* (1992), 255–258.

[35] Paul M Leonardi. 2011. When flexible routines meet flexible technologies: Affordance, constraint, and the imbrication of human and material agencies. *MIS quarterly* 35, 1 (2011), 147–167.

[36] Ramon Leyva. 2016. Optimal sizing of Cuk converters via Geometric Programming. In *IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2480–2485.

[37] Eunice Lin. 2017. *Collaboration in design optimization*. Master's thesis. Massachusetts Institute of Technology.

[38] Gloria Mark. 2002. Extreme collaboration. *Commun. ACM* 45, 6 (2002), 89–93.

[39] Nolwenn Maudet, Germán Leiva, Michel Beaudouin-Lafon, and Wendy Mackay. 2017. Design Breakdowns: Designer-Developer Gaps in Representing and Interpreting Interactive Systems. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*. ACM, 630–641.

[40] M Mutacipc, K Koh, S Kim, and S Boyd. 2008. A matlab toolbox for geometric programming. (2008).

[41] Carl Myhill. 2004. Commercial success by looking for desire lines. In *Asia-Pacific Conference on Computer Human Interaction*. Springer, 293–304.

[42] Bonnie A Nardi and James R Miller. 1991. Twinkling lights and nested loops: distributed problem solving and spreadsheet development. *International Journal of Man-Machine Studies* 34, 2 (1991), 161–184.

[43] Jorge Nocedal and Stephen J Wright. 2006. *Numerical optimization*. Springer Science+ Business Media.

[44] Brendan O'Donoghue, Eric Chu, Neal Parikh, and Stephen Boyd. 2016. Conic optimization via operator splitting and homogeneous self-dual embedding. *Journal of Optimization Theory and Applications* 169, 3 (2016), 1042–1068.

[45] Dan R. Olsen. 2007. Evaluating User Interface Systems Research. *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology* (2007), 251–258. DOI: http://dx.doi.org/10.1145/1294211.1294256

[46] Max MJ Opgenoord, Brian S Cohen, and Warren W Hoburg. 2017. Comparison of Algorithms for Including Equality Constraints in Signomial Programming. *Aerospace Computational Design Lab., Massachusetts Inst. of Technology, TR-17-1, Cambridge, MA* (2017).

[47] Thomas Østerlie, Petter G Almklov, and Vidar Hepsø. 2012. Dual materiality and knowing in petroleum production. *Information and organization* 22, 2 (2012), 85–105.

[48] Berk Öztürk. 2018. *Conceptual engineering design and optimization methodologies using geometric programming*. Master's thesis. Massachusetts Institute of Technology.

[49] Berk Ozturk and Ali Saab. 2019. Optimal Aircraft Design Deicions under Uncertainty via Robust Signomial Programming. In *AIAA Aviation 2019 Forum*. 3351.

[50] Gerhard Pahl and Wolfgang Beitz. 2013. *Engineering design: a systematic approach*. Springer Science & Business Media.

[51] Kevin LG Parkin, Joel C Sercel, Michael J Liu, and Daniel P Thunnissen. 2003. Icemaker™: an excel-based environment for collaborative design. (2003).

[52] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2017. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* 25, 1 (2017), 83–110.

[53] Ryan Pilgrim. 2017. Source Coding Optimization for Distributed Average Consensus. *arXiv preprint arXiv:1710.01816* (2017).

[54] Alex Preda. 2006. Socio-technical agency in financial markets: The case of the stock ticker. *Social Studies of Science* 36, 5 (2006), 753–782.

[55] Prashanth Rajivan, Pablo Moriano, Timothy Kelley, and L Jean Camp. 2017. Factors in an end user security expertise instrument. *Information & Computer Security* 25, 2 (2017), 190–205.

[56] A Rizzo, O Parlangeli, E Marchigiani, and S Bagnara. 1996. The management of human errors in user-centered design. *ACM SIGCHI Bulletin* 28, 3 (1996), 114–118.

[57] BF Robertson and DF Radcliffe. 2009. Impact of CAD tools on creative problem solving in engineering design. *Computer-Aided Design* 41, 3 (2009), 136–146.

[58] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. 2017. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *arXiv preprint arXiv:1708.08296* (2017).

[59] D Schon. 1983. *The reflective practitioner. How professionals think in action*. Temple Smith, London.

[60] Junnan Shan, Mario R Casu, Jordi Cortadella, Luciano Lavagno, and Mihai T Lazarescu. 2019. Exact and Heuristic Allocation of Multi-kernel Applications to Multi-FPGA Platforms. In *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 3.

[61] Tony Shuo Tao. 2018. *Design, optimization, and performance of an adaptable aircraft manufacturing architecture*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[62] Madeleine Udell, Karanveer Mohan, David Zeng, Jenny Hong, Steven Diamond, and Stephen Boyd. 2014. Convex optimization in Julia. In *High Performance Technical Computing in Dynamic Languages (HPTCDL), 2014 First Workshop for*. IEEE, 18–28.

[63] Eppinger Steven D Ulrich, Karl T and Maria C Yang. 2011. Product Design and Development. (2011).

[64] Bo Wu, Murat Cubuktepe, Suda Bharadwaj, and Ufuk Topcu. 2019. Reward-Based Deception with Cognitive Bias. *arXiv preprint arXiv:1904.11454* (2019).

[65] Chin Chang Wu. 1976. *Design and modeling of solar sea power plants by geometric programming*. Carnegie-Mellon Univ., Pittsburgh, PA (USA).

[66] Maria C Yang. 2005. A study of prototypes, design activity, and design outcome. *Design Studies* 26, 6 (2005), 649–669.

[67] Maria C Yang. 2009. Observations on concept generation and sketching in engineering design. *Research in Engineering Design* 20, 1 (2009), 1–11.

[68] Martin A York, Warren W Hoburg, and Mark Drela. 2017. Turbofan engine sizing and tradeoff analysis via signomial programming. *Journal of Aircraft* 55, 3 (2017), 988–1003.