

How Can Large Language Models Help Humans in Design and Manufacturing? Part 1: Elements of the LLM- Enabled Computational Design and Manufacturing Pipeline

**Liane Makatura¹ Michael Foshey¹ Bohan Wang¹ Felix Hähnlein²
Pingchuan Ma¹ Bolei Deng¹ Megan Tjandrasuwita¹ Andrew Spielberg³
Crystal Owens¹ Peter Yichen Chen¹ Allan Zhao¹ Amy Zhu²
Wil Norton¹ Edward Gu¹ Joshua Jacob¹ Yifei Li¹ Adriana Schulz²
Wojciech Matusik¹**

¹Massachusetts Institute of Technology, Cambridge, Massachusetts, United States of America,

²University of Washington, Seattle, Washington, United States of America,

³Harvard University, Cambridge, Massachusetts, United States of America

The MIT Press

Published on: Dec 23, 2024

DOI: <https://doi.org/10.1162/99608f92.cc80fe30>

License: [Creative Commons Attribution 4.0 International License \(CC-BY 4.0\)](#)

ABSTRACT

The advancement of large language models (LLMs), including GPT-4, provides exciting new opportunities for generative design. We investigate the application of this tool across the entire design and manufacturing workflow. Specifically, we scrutinize the utility of LLMs in tasks such as: converting a text-based prompt into a design specification, transforming a design into manufacturing instructions, producing a design space and design variations, computing the performance of a design, and searching for designs predicated on performance. Through a series of examples, we highlight both the benefits and the limitations of the current LLMs. By exposing these limitations, we aspire to catalyze the continued improvement and progression of these models.

“How Can Large Language Models Help Humans in Design and Manufacturing?” is a two-part article. Part 2, “Synthesizing an End-To-End LLM-Enabled Design and Manufacturing Workflow” can be read [here](#).

Keywords: large language models, GPT-4, computational design, computational fabrication, CAD, CAM, design for manufacturing

Media Summary

Large language models (LLMs) and other generative, artificial intelligence (AI) technologies need not be confined to creating virtual content; they can help automate the creation of physical content as well. We take a magnifying lens specifically to GPT-4’s ability to automate aspects of the (computational) manufacturing pipeline, including designing products and iterating on those product designs through dialogue, evaluation, and optimization, as well as preparing those designs for manufacturing. We demonstrate GPT-4’s ability to enable an end-to-end design and manufacturing workflow by using it to create two physical demos: a cabinet and a quadcopter.

In each section, we pose investigative questions about GPT-4’s capabilities to aid design and manufacturing, and study them through detailed experiments. In addition to the cabinet and quadcopter, design domains studied include other types of furniture and robots as well as kitchen items, toys, 3D graphics, and more.

At each stage of the design workflow we find challenges and opportunities, though some major themes persist. While LLMs are versatile and widely knowledgeable about the entire manufacturing pipeline and many target design domains, they currently tend to make mistakes and can require human teaming on hard problems. In particular, LLMs, being language based, currently tend to have a lack of spatial reasoning capabilities, causing them to struggle with geometric complexity in design. Further, LLMs such as GPT-4 do not scale gracefully to complex problems, especially as they struggle to reason about long conversations. Finally, LLMs are only

fluent in topics about which there is data available at scale, and this does not extend to more specific or emerging topics. Despite these limitations, they facilitate rapid iteration, and, when used judiciously, can still solve hard tasks when a user aids GPT-4 in decomposing problems at the module level.

We see the current limitations as surmountable. LLMs are only now becoming a mainstream technology, meaning that they are relatively nascent from an applications development perspective. Methods that incorporate spatial reasoning, hierarchically decompose complex tasks, and recursively perform error-checking and error-correction seem tractable. Such methods will make design and manufacturing workflows more automated and streamlined. Further, providing native access to mature domain-specific tools will give LLMs access to the same computing technologies as their humans counterparts.

LLMs bridge the gap between conceptual design and technical design; we show that they can also bridge the gap between virtual and physical, providing a concrete path to a new manufacturing paradigm.

1. Introduction

Advances in computational design and manufacturing (CDaM) have already permeated and transformed numerous industries, including aerospace, architecture, electronics, dentistry, and digital media, among others. Nevertheless, the full potential of the CDaM workflow is still limited by a number of barriers, such as the extensive domain-specific knowledge that is often required to use CDaM software packages or integrate CDaM solutions into existing workflows. Generative AI tools such as large language models (LLMs) have the potential to remove these barriers, by automating or semiautomating each step of the CDaM process, providing means by which to combine external software tools with middleware, and by providing an intuitive, unified, user-friendly interface by which users can customize the pipeline for their goals. However, to date, generative AI and LLMs have seldom been applied to hard manufacturing problems. In this study, we show how these tools can (and cannot) currently be used to develop new design and manufacturing workflows.

Our analysis examines the standard CDaM workflow to identify opportunities for LLM-driven automation or acceleration. Specifically, we break the CDaM workflow into five phases, and then assess whether and how the efficiency and quality of each phase could be improved by integrating LLMs. The components under investigation include (1) generating a design, (2) constructing a design space and design variations, (3) preparing designs for manufacturing, (4) evaluating a design's performance, and (5) discovering high-performing designs based on a given performance metric and design space.

Although it is feasible to create specialized LLMs for design and manufacturing, we demonstrate the opportunities offered by generic, pretrained models. To this end, we conduct all of our experiments using GPT-4 ([OpenAI, 2023](#)), ¹ a state-of-the-art general purpose LLM. Our GPT-4-augmented CDaM workflows demonstrate how LLMs could be used to simplify and expedite the design and production of complex objects. Our analysis also showcases how LLMs can leverage existing solvers, algorithms, tools, and visualizers to

synthesize an integrated workflow. Finally, our work demonstrates current limitations of GPT-4 in the context of design and manufacturing, which naturally suggests a series of potential improvements for future LLMs and LLM-augmented workflows.

In order to ground the discussion in a concrete goal, we highlight two running design examples throughout the main text: a cabinet, which examines GPT-4's ability to reason about static (but functional) items that can be built with traditional manufacturing processes, and a quadcopter, which examines GPT-4's ability to reason about the design and manufacturing of cyber-physical, dynamical systems. We visit these examples in most sections, and synthesize them into final, working products by the end of the manuscript. For the sake of brevity, extensive additional experiments are summarized in the main text and shown in increased detail in the appendix.

Before continuing to the overview, we provide a brief note about the goals and scope of this article, in order to set reader expectations. While much of this article is dedicated to understanding what GPT-4 excels and struggles with (or, what it can and cannot do) at the time of writing, it is difficult to prove a negative. This article should be read as a) a way of understanding, at a high level, what is easy and what is difficult for LLMs in CDaM, b) an analysis to provide recommendations of how to use LLMs effectively in CDaM (distilling effectual workflows), and c) an attempt to understand the opportunities for improving and extending LLMs, discovering what will and will not be possible in the near and far future. Negative results in this article do not necessarily mean that something is not possible with an LLM or even GPT-4, but rather, that we struggled with a task after considerable effort. Other workflows may exist that we have not yet examined. It is intractable to perform a systematic exploration and evaluation of GPT's entire generation space. We have thus focused on strategic, principled, and when possible, broad case studies. We believe that this work is representative of the current state of LLMs and hope it is useful for the audience.

We have structured this work into two parts. While the first part investigates the current potential of LLMs within the five phases of the CDaM workflow, the second part presents the two end-to-end examples. Readers interested in individual aspects of the CDaM workflow should look to Part 1 to understand how LLMs can be useful in each phase of design and manufacturing. Readers interested in how LLMs can be useful in end-to-end workflows and in understanding the underlying challenges in making such systems will find interest in Part 2. At the end of Part 2, we further summarize our findings and recommendations for the combined parts.

Vision. As with any nascent technology, LLMs have both strengths and gaps in their capability. Throughout this article (including at the end of each section), we will highlight where LLMs are strong and useful, where they are weak, and where they provide some mixed, possibly unclear benefit. We highlight where users can presently adapt workflows to incorporate LLMs and to extract maximal value of LLMs. We envision such technology will grow in capability over time, especially in areas of present weakness. In the context of CDaM, these notably include complex geometric reasoning, hidden assumptions, overconfidence, hallucinations, short memories, and overfit answers. As this technology improves, users will benefit from further integration of

CDaM software with LLMs. However, language is only one possible input modality, and integrating generative AI with other forms of input (gestures, images, 3D models, etc.) will provide ways for traditional workflows to gradually evolve into AI-augmented design software.

2. Introduction to Part 1

In this part, we conduct experiments to study how text-based interactions allow us to 1) generate a design, 2) construct a design space and design variations, 3) prepare designs for manufacturing, 4) evaluate a design's performance, and 5) discover high-performing designs based on a given performance metric and design space. We note that these five steps are rarely performed in isolation. Manufacturing imposes hard constraints on what kind of design can be realized, design aesthetics can prime or be driven by performance considerations, and so on. In this part, we have artificially isolated these steps to focus our experiments on single aspects of the process; end-to-end workflows are the dedicated focus of Part 2.

Vision. Throughout the computational design workflow, designs are represented by various data structures, for example, hierarchical representations, dependency structures, boundary representations, feature lists, and so on. These representations are of critical importance for downstream tasks, especially in considering both a design optimization process and in considering design spaces that are suitable for manufacturing. Unfortunately, design representations are often selected early on. Much of this part (including the subsequent two sections) discusses explorations of design spaces. Unfortunately, as will also be seen, LLMs presently suffer from an inability to reason about long or highly detailed procedures, limiting the expressiveness of designs across modalities (geometry, materials, on-board controls, and so on).

Given the critical importance of design representations and fidelity, we envision future generative CDaM software to incorporate the explorative and wide-knowledge-base strengths of LLMs, but also to explicitly reason about design representations behind the scenes. Future generative systems should process user input and autonomously determine an (abstracted, under-the-hood) design representation. As mission specifications (constraints, goals, contexts, variables) change, the design system should on-the-fly modify/convert these representations appropriately. A system should only burden the user with reasoning about these representations when explicitly queried; that is, when a user requests some direct control over them. Further, a generative design system should automatically order the steps of a workflow for a user. For example, if the inverse design of a component or subcomponent is desired, all relevant representations, evaluation methods, and optimization methods should be determined before any parameterization, simulator, or solver is instantiated. If a design is sufficiently complex, a design system should recursively and hierarchically partition the CDaM workflow (including its base representations) as needed.

In cases where direct modifications to, say, geometry are deemed necessary by a user, a system should provide strong high-level handles with which to interact naturally with geometry. In other words, a user should be able

to directly modify features (such as the profile of an object of revolution) without needing to explicitly reason about CAD abstractions (such as 2D views and revolution axes).

Ultimately, we envision that LLMs will not replace existing workflows, but that they will complement them and help the user get a better understanding of both the constraints imposed by the programmatic design and the normative ground of the design software ([Li et al., 2023](#)).

3. Background

To contextualize our work, we briefly describe the state-of-the-art for generative LLMs, as well as recent breakthrough methods for various aspects of CDaM. We note that, given the pace and breadth of both fields, this list is by no means comprehensive.

3.1. LLMs for Generative Modeling

LLMs have garnered significant interest in the research community and beyond, as a result of both their already demonstrated generative capabilities and their seemingly unbounded promise. Although these models are recognized primarily for their influence on text generation ([Radford et al., 2019](#)), their reach has been extended to impact various other domains, including image generation ([Ramesh et al., 2021](#)), music generation ([Dhariwal et al., 2020](#)), motion generation ([Jiang et al., 2023](#)), code generation ([Chen et al., 2021](#)), 3D model creation ([Liu et al., 2023](#)), and robotic control ([Mirchandani et al., 2023](#)). Notable foundation models include OpenAI’s GPT series, ranging from GPT-2 to the more recent GPT-4 ([OpenAI, 2023](#)). These models have showcased progressive improvements in fluency, coherence, and generalization capabilities. Meta AI’s CM3Leon model has further extended the reach of large models by demonstrating proficiency in both text and image synthesis ([Yu et al., 2023](#)). The Falcon LLM ([Penedo et al., 2023](#)), trained exclusively on properly filtered and deduplicated web data, has exhibited comparable performance to models trained on meticulously curated data sets. These models have been utilized in conjunction with reinforcement learning from human feedback (RLHF) to improve the quality of the generated content ([Quyang et al., 2022](#)). This is done by incorporating human feedback into the training process, where humans rate the quality of the generated outputs and provide examples of ideal outputs for a given input ([Christiano et al., 2017](#)). In parallel, domain-specific LLMs have also been trained for performance within a specific subject area. For example, ProtGPT2 specializes in predicting protein folding structures ([Ferruz et al., 2022](#)), while Codex has been specifically tailored to understand and generate code ([Chen et al., 2021](#)). In this work, we investigate the generative capabilities of generic, pretrained LLMs within CDaM.

4. Part 1: Related Work

4.1. Computational Design and Manufacturing

The CDaM workflow is often decomposed into a series of steps including (1) representing a design, (2) representing and exploring a design space, (3) preparing a design for manufacturing, (4) computing the

performance of a design, and (5) finding a design with optimal performance. For each phase, we provide a brief overview of the relevant work, with a focus on aspects that offer the best opportunities for LLM integration.

4.1.1. Design Representations

The cornerstone of computational design is the capacity to digitally represent and manipulate the salient aspects of a given design—such as geometry, articulated joints, material composition, and so on. There are many ways to represent such aspects, but we focus on design representations that are compact, understandable, and editable. For example, modern computer-aided design (CAD) systems represent a shape as a sequence of operations such as 2D sketches, extrusions, and Boolean operations ([Willis et al., 2021](#)). These can be represented as compact programs written in domain-specific languages (DSLs) such as OnShape’s FeatureScript (["FeatureScript Introduction," n.d.](#)). Designs can also be represented compactly as a graph ([Prusinkiewicz & Lindenmayer, 1990](#); [Zhang et al., 2018](#)), in which the nodes typically represent individual components, while edges represent multicomponent interactions. Such graphs have been used to efficiently and hierarchically represent CAD models ([T. Du et al., 2018](#)), robots ([Zhao et al., 2020](#)), metamaterials ([Makatura et al., 2023](#)), architecture ([Müller et al., 2006](#)), and chemical molecules ([Guo et al., 2022](#)). To represent even more complex designs—such as a quadcopter with a physical design and a software controller—multiple DSLs may be used simultaneously. For example, the copter’s physical design may be encoded using CAD, while its software is coded using a control-specific DSL.

4.1.2. Design Space Representations

A design space represents an entire family of designs—rather than a single instantiation—which allows for design exploration, customization, and performance-driven design optimization. One of the most popular design space representations is parametric design, in which a few exposed parameters are used to control a design. This is commonly used in CAD systems, where, for example, a bookshelf may be parametrized by its height, width, depth, and number of shelves. Another popular option is formal languages such as L-systems ([Rozenberg & Salomaa, 1980](#)) or shape-grammars ([Özkar & Stiny, 2009](#); [Stiny, 1980](#)), which generate design variations by manipulating a set of terminal and nonterminal symbols according to given rewrite rules. Formal languages have been used in domains such as architecture ([Müller et al., 2006](#)), robotics ([Zhao et al., 2020](#)), and chemistry ([Guo et al., 2022](#)).

4.1.3. Design for Manufacturing

Design for manufacturing (DFM) is a planning process used to generate designs that can be fabricated with maximal efficiency and minimal cost. One prominent aspect of this is computer-aided manufacturing (CAM), which transforms a digital design into a viable fabrication plan for some manufacturing process, such as 3D printing, 3- or 5-axis computer numerical control (CNC) milling, or sheet-metal stretching. CAM also extends to multiprocess representations such as STEP-NC, which abstracts away from machine-specific G-code in

favor of tool-type-specific machining operations that are interpretable on different hardware. Because all of these fabrication plans can also be described as a program in some DSL, CAM can be interpreted as a *translation* from a design DSL to a manufacturing-oriented DSL. DFM also includes many other aspects, such as selecting an appropriate manufacturing method, optimizing manufacturing process parameters ([Erps et al., 2021](#)), sourcing parts and materials, or modifying a design in light of manufacturing constraints ([Koo et al., 2017](#)).

4.1.4. Performance Prediction

Before manufacturing a design, engineers typically want to understand its predicted performance. For example, automobile engineers may wish to evaluate and iteratively refine a candidate design’s efficiency, safety, and aesthetics. To do this, engineers frequently make use of numerical simulation methods such as general purpose finite element analysis (FEA) ([T. Du et al., 2021](#)) or more domain-specific approaches for, for example, acoustics ([O’Brien et al., 2002](#)), robotics ([Erez et al., 2015](#)), and electromagnetism ([Sullivan, 2013](#)). Commercial CAD systems, for example, [Autodesk \(n.d.\)](#) and [Dassault Systèmes \(n.d.\)](#) integrate simulation into their ecosystem. Since engineers are primarily interested in the performance of the design’s manufactured counterpart, it is crucial to minimize the gap between an object’s performance in simulation versus reality.

4.1.5. Performance Optimization

Given a design space and a way to predict performance, it is natural to seek designs that perform best with respect to a particular metric. Although this search could be performed via manual trial and error, it is more efficient and effective to use automated exploration tools. One process known as *design optimization* or *inverse design* can automatically search (or optimize) over a given design space to find a design that exhibits some target performance ([Ma et al., 2021](#)). Inverse design has already been applied to many problem domains. For example, a parametric design space can be searched for designs that have the best value of a simulated metric ([Xu et al., 2021](#)). Topology optimization has been applied to problems such as minimum compliance. In addition, designs can be optimized for metrics such as weight, cost, and manufacturing time.

4.1.6. Human-Centric and Automated Strategies for Solving Problems With LLMs

In many cases, solving hard problems (especially inverse problems, which involve many subcomponents) require modularization into subproblems that can be more readily solved and easily iterated on. This strategy further dovetails with emerging work that provides automatic solutions for problem modularization ([Richards, n.d.](#)), boosting ([Ni & Buehler, 2024](#)), and error-correction ([Y. Du et al., 2023](#)).

5. Overview

The fundamental aim of this study is to conduct an in-depth exploration of the opportunities and challenges of applying contemporary LLMs within the landscape of the CDaM workflow described in [Section 4.1](#). Driven by

this objective, we propose a thorough and wide-ranging exploration that is independent of any predefined or proposed framework.

To apply LLMs coherently across such diverse tasks, we leverage the insight that all building blocks in the CDaM workflow (design, design spaces, manufacturing instructions, and performance metrics) can be represented by compact programs. Thus, at a high level, every phase of the CDaM workflow can be seen as a translation layer between an input domain-specific language and an output DSL. The fact that LLMs excel at such symbolic manipulations suggests that LLMs have the potential to address these tasks, while simultaneously leveraging and improving upon our traditional solutions.

To achieve comprehensive coverage and uncover the different facets of LLM-assisted CDaM, we have undertaken an extensive suite of experiments, incorporating a broad variety of design representations, manufacturing processes, and performance metrics. These are detailed further in [Section 5.2](#).

5.1. Methodology

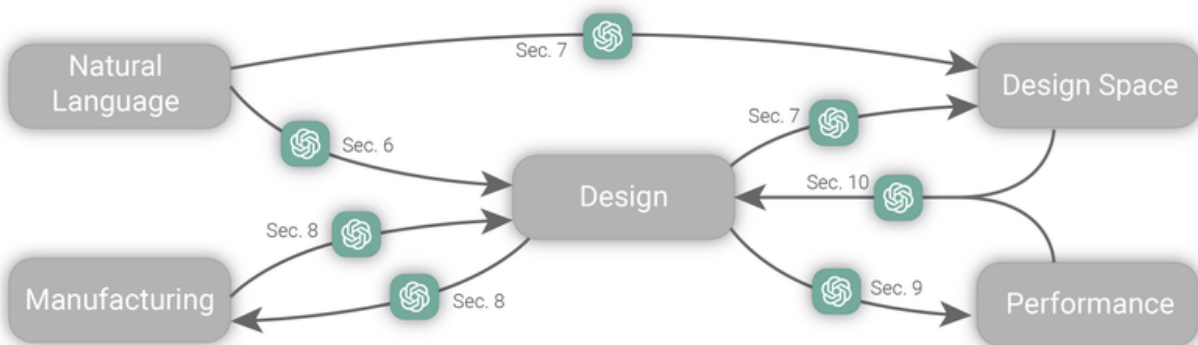


Figure 1. Opportunities for large language model (LLM) integration within the computational design and manufacturing workflow. Each technical section of our article covers opportunities for LLM integration in one of the tasks depicted above: natural language to design, natural language/design to design space, bi-directional design for manufacturing, design to performance, and design optimization (from performance and design space to an optimized design).

Our methodology is crafted to provide a comprehensive inspection of the opportunities for and efficacy of various interfaces between GPT-4 and the CDaM workflow. We investigate each of the five stages of the design and manufacturing pipeline individually. As illustrated in [Figure 1](#), these stages include: design generation from natural language ([Section 6](#)), design space generation (from language and examples) ([Section 7](#)), design for manufacturing ([Section 8](#)), performance prediction (aka computer aided engineering, or CAE) ([Section 9](#)), and design optimization aka inverse design ([Section 10](#)).

In each of these stages, we pose fundamental questions about ways in which GPT-4 may offer some benefit, and then conduct a series of experiments to answer these questions. For each query, we investigate aspects such as (1) strategies for engineering effective prompts, (2) strategies for integrating human feedback, expertise, or

preferences into the LLM-assisted design process, and (3) tasks that GPT-4 can accomplish natively versus tasks that are better completed by asking GPT-4 to leverage external tools.

After a detailed examination of each stage, we sought to understand the implications of incorporating GPT-4 within an end-to-end CDaM process. To this end, we designed and fabricated two practical examples (a cabinet and a quadcopter) with GPT-4's support. The end-to-end design process for each example is detailed in [Part 2, Section 3](#).

Beyond these individual questions, our comprehensive investigation has exposed several key insights about GPT-4's general capabilities and limitations with respect to CDaM. We have also observed a group of properties that we term 'dualisms' because they may manifest either as an opportunity or a drawback, depending on the situation. Our findings are summarized in [Table 1](#), with a full description in [Part 2, Section 5](#). To emphasize the pervasive nature of these properties, we also use these labels as a framework for our discussions and takeaways at the end of each section. Specifically, we draw on each section's findings and examples in order to illustrate the manifestation and impact of various properties in [Table 1](#) throughout the CDaM workflow.

Table 1. GPT-4's key properties for CDaM. To facilitate discussion of GPT-4's applicability for design and manufacturing (Des.&Mfg.), we have identified nine key observations about GPT-4 that persist across several aspects of the CDaM workflow. This includes three powerful capabilities, four limitations, and two dualisms (so named because they may manifest either as an opportunity or a drawback, depending on the context). We use these observations to frame our discussions about GPT-4's suitability for each stage of the CDaM workflow.

Category	Code	Title	Summary
Capabilities	C.1	Extensive Knowledge Base in Des.&Mfg.	GPT-4 has a broad knowledge of design and mfg. considerations
	C.2	Iteration Support	GPT-4 attempts (and often succeeds) to iterate and rectify errors when prompted
	C.3	Modularity Support	GPT-4 can reuse or adapt previous/provided designs or solutions

Limitations	L.1	Reasoning Challenges	GPT-4 struggles with spatial reasoning, analytical reasoning, and computations
	L.2	Correctness and Verification	
	L.3	Scalability	GPT-4 produces inaccurate results or justifications for its solutions
	L.4	Iterative Editing	GPT-4 struggles to respect multiple requests concurrently GPT-4 forgets/introduces errors when modifying previously-generated designs
Dualisms	D.1	Context Information	GPT-4’s performance depends on the amount of context provided
	D.2	Unprompted Responses	GPT-4 makes inferences/suggestions beyond what is specified in the prompt

5.2. Scope of Evaluation

To conduct a holistic survey of GPT-4-assisted CDaM, our experiments span a number of different design domains ([Section 5.2.1](#)), performance metrics ([Section 5.2.2](#)), and manufacturing methods ([Section 5.2.3](#)). Here, we briefly describe each domain of interest, along with the specific challenges they pose and the sort of representative, transferable insight we hope to glean by studying each domain in connection with LLMs.

5.2.1. Target Design Domains

Our experiments are concentrated in three main design domains, including 2D vector graphic design, 3D parametric modeling, and articulated robotics problems.

Vector graphics use a series of text-based commands to represent paths and areas that form a given design. Vector image formats are an important part of CDaM, as they can be used as both a design specification and a manufacturing specification for, for example, laser cutters. Despite their simplicity, vector graphics can represent a wide range of 2D and 3D objects, such as artistic engravings or flat-pack furniture. We examine LLMs’ capacity to generate two popular vector formats: scalable vector graphics (SVG) and drawing exchange format (DXF). These formats present several challenges: they contain boilerplate formatting that GPT-4 may struggle to reproduce; they often require GPT-4 to reason about the layout of individual pieces on the canvas;

and finally, they test GPT-4’s ability to decompose higher dimensional designs into 2D. Thus, vector graphics will test GPT-4’s spatial reasoning and ability to respect highly constrained syntax, either on its own or with the use of external libraries.

Parametric modeling languages generate 3D geometry through a sequence of constructive instructions. The term ‘parametric modeling’ reflects how each constructive operator exposes a set of parameters, such as the radius of a circle. We explore two distinct approaches that are powerful, widely used, and well-documented online. The first is rooted in classic constructive solid geometry (CSG), which constructs shapes by successively deploying Boolean operations (union, intersection, subtraction) over basic shapes or primitives (such as cuboids, spheres, cylinders, and so forth) that can undergo transformations such as translations, rotations, and scaling. The CSG approach is intended to test the *global* spatial reasoning capacity of GPT-4, as every CSG operation/transformation occurs with respect to a shared coordinate space. The second representation relies on the contemporary B-rep format used by modern CAD systems. Here, geometry is built through a sequence of operations like sketching, extruding, and filleting. Each operation in this context is parametric and uses references to previously created geometry to, for example, select a plane for a sketch design or select a sketch for an extrusion. Sketch-based CAD will test GPT-4’s ability to effectively switch between and reason over multiple relative, local coordinate frames.

Robotics offers a particularly rich design domain, as GPT-4 must coordinate a set of *articulated* and *actuated* geometries to form complex objects such as open chain robot arms, wheeled robots, copters/UAVs, and robot grippers. Robotics representations must describe not only the high-level geometry of each part, but also their properties and relationships—including the joints between parts, the degrees of freedom that those joints exhibit, and dynamics information such as the inertia of a given part. Several existing formats support these tasks, but we primarily use the XML-based language known as the universal robot description format (URDF). We also investigate the use of a more general graph-based robot representation. These formats test GPT-4’s ability to simultaneously reason about multiple aspects of design, such as static geometric bodies and dynamic articulation constraints.

Our choices for parametric CAD and robot description files, for which we have chosen OpenSCAD/OpenJSCAD and URDF throughout much of this article, are constrained by a lack of robust alternatives. Methods for parameterizing CAD files are often implemented by high-level CAD software and particular geometry kernels, and lack a universal interchange format. While specific CAD programs can output files that record parameterizations, universal training data is in low supply, and thus it is a difficult generation target for LLMs. Open(J)SCAD provides a native programming-language-based parametric CSG-based CAD format with a robust community and examples, thus making it a good choice for our experiments. Meanwhile, URDF is one of two major open robot representation formats, with the other being the XML format of MuJoCo ([Todorov et al., 2012](#)). However, MuJoCo’s representation commingles robot design, environment, and

simulator, and is thus less modular and more unnecessarily complex for our purposes. URDF, meanwhile, encodes a minimal but feature-complete representation of robot morphology.

5.2.2. Target Performance Domains

Diverse performance domains within engineering design require evaluation of aspects such as structural and material properties, mechanical integrity, geometry-based functionality, materials use, electromechanical integration, and subjective features. The results of such evaluation allow us to (dis)qualify a design for use and to use the evaluation to further understand and improve the design. Using GPT-4, we focus on assessing mechanical and structural properties through generating first-order analysis equations for input designs of standard objects like chairs, cabinets, and a quadcopter, which test the ability of GPT-4 to sufficiently understand a given input design in text form or through a DSL and to evaluate criteria for functionality and failure. Mechanical properties assessed include weight, size, load capacity, storage capacity, and stability. Analysis of electromechanical functionality include battery life and quadcopter travel distance. Further use of GPT-4 aims to streamline the computationally intensive process of finite element analysis (FEA), a crucial tool for understanding structural behavior in detail under various conditions, and we apply this to the case of a load on a set of chairs.

In addition to these technical aspects, our investigation extends into the subjective domains of sustainability and aesthetics, which cannot be strictly quantified. The inherent complexity and qualitative nature of these areas present unique challenges in evaluation. While it is well-known that computational systems can compute quantitative features, machine learning systems are becoming more sophisticated in artistic domains, and so we seek to leverage the capacity of LLMs for lexical analysis to aid more holistically in the more ambiguous realms of the design process and to find its limits. For example, could an LLM reasonably address whether a piece of furniture of a given size is ‘large,’ or if a shoe of a given design is ‘comfortable,’ or can it only handle classically quantifiable features? Can it even help us to reason more objectively about what aspects delineate these properties? To this end, we test evaluation of subjective domains and use GPT-4 to generate a scoring system and functions for quantifying the sustainability of a chair, the classification of chairs based on categories of aesthetic influence, and the appropriate distribution of a set of chairs into a set of rooms in a house, among other examples.

We further combine these performance metric evaluations with the principles of design optimization. Design optimization entails finding performant designs according to user- (or LLM) specified metrics and subject to a design space, as well as selecting appropriate methods of optimization. In this case, given a design/decision space for an object, we use GPT-4 to generate and implement methods to computationally improve or optimize qualifying designs to satisfy designated performance goals. This methodical approach evaluates whether LLMs can apply constructive logic for design enhancement and innovation.

5.2.3. Target Manufacturing Domains

Leveraging language models like GPT-4 in the design for manufacturing context can yield more consistent and scalable decision-making, potentially augmenting human expertise and reducing our reliance on CAD software usage. Potential applications of GPT-4 include the selection of optimal manufacturing techniques, suggestion of design modifications that would enable easier production, identification of potential suppliers, and creation of manufacturing instructions. The approach is aimed to alleviate many of the bottlenecks caused by the designers' lack of knowledge and experience in DFM.

In a set of experiments, we have explored GPT-4's capabilities across various tasks. First, GPT-4 was used to identify the optimal manufacturing process for a given part, considering factors such as part geometry, material, production volume, and tolerance requirements. Next, GPT-4 was tasked with optimizing a component design for CNC machining. Given the geometry of the component, GPT-4 identified potential manufacturing difficulties and modified the design to address these. We also leveraged GPT-4's extensive data set knowledge to identify parts needed for manufacturing.

In addition to these, GPT-4 was used to create manufacturing instructions for both additive and subtractive design processes. Additive design can be challenging due to the need for spatial reasoning, precision, and meticulous planning, and often requires many iterations. We have explored the generation of fabrication instructions using subtractive manufacturing techniques for a cabinet design. We also investigated GPT-4's potential in generating machine-readable instructions for robot assembly tasks and converting those into human-readable standard operating procedures. This allowed for effective communication and collaboration between robots and human operators.

6. Generating Designs From Natural Language Specifications

For our first line of inquiry, we explore the extent to which GPT-4 is able to generate designs across a variety of domains. Even within the specific context of manufacturable design, the concept of a 'design' is quite broad, and exists at many scales. For example, we may want to specify a single self-contained part, or a sizable hierarchical assembly containing several levels of sub-assemblies and/or other individual component modules. Such assemblies may be completely customized/self-contained, with all parts designed simultaneously, or they may be hybrid designs that integrate existing, premanufactured elements such as brackets or motors. In many cases, our target design tasks also include dynamic considerations such as assembly mating or articulated joints.

Although these complex tasks may initially seem out-of-scope for lexical models such as LLMs, there are many modeling and design paradigms that can be expressed in terms of potentially LLM-compatible language. To guide our exploration of GPT-4's ability to interface with each of these models, we pose the following questions:

- **Q1** Can GPT-4 generate a meaningful design when provided with a high-level description of the goal and a given modeling language?
- **Q2** To what extent is the user able to control the designs created by GPT-4? Is GPT-4 able to interpret and respect user-defined constraints, such as spatial relationships between objects or integration of standard prefabricated parts?
- **Q3** Is GPT-4 able to incorporate high-level abstractions used by human designers, such as modular (de)composition?

6.1. Simple, Self-Contained Designs from High-Level Input (Q1)

6.1.1. CSG With OpenJSCAD

To explore a full-fledged approach for LLM-aided CSG, we test GPT-4’s ability to generate meaningful designs using the open source Javascript-based constructive solid geometry (CSG) library, OpenJSCAD (["JSCAD User Guide," n.d.](#)). OpenJSCAD has extensive documentation available online, and we found that GPT-4 natively possesses a good grasp of the API, its components, and the required code structure. In particular, it understood that it needed to import each function from the corresponding modules, and that it needed to define and export a function named `main`. For our experiments, we provided GPT-4 with access to the full application programming interface (API), and generally allowed it to select the appropriate primitives and operations without user interference.

To test GPT-4’s design abilities, we ask it to design a simple cabinet with one shelf, as shown in [Figure 2](#). GPT-4 reliably selects and instantiates the required primitives, along with intuitive naming conventions and structure within the OpenJSCAD code. GPT-4’s initial orientation of the parts was also generally reasonable, but the specific positioning of each part was often incorrect. Despite multiple attempts, GPT-4 was unable to generate any fully correct cabinet in a single shot, with no subsequent user intervention.

Moreover, GPT-4 frequently produced highly disparate results from one run to the next. Even when using an identical prompt on fresh chat environments, GPT-4’s responses varied widely in terms of their overall code structure, design accuracy, and the specific errors or oversights made. [Figure 3](#) shows one example of a drastically different design process, even when seeded with the same initial prompt as [Figure 2](#).

Throughout our experiments, we found that GPT-4 encountered a few common pitfalls when generating designs in OpenJSCAD. Occasionally, GPT-4 made small syntactic errors such as generating incorrect boilerplate, importing functions from incorrect modules, or making ‘typos’ in API calls—for example, trying to import from the `boolean` module rather than the correct `booleans` module, or calling the `cube()` function with parameters that were intended to generate a `cuboid()`. In an attempt to avoid these pitfalls, we created a small list of ‘hints’/‘reminders’ for best practices when working with OpenJSCAD; this short list was always passed in alongside our initial prompt. See [Appendix A.1](#) for a full listing of these reminders. Although these reminders seemed to help mitigate these issues, we were unable to eradicate them entirely. However, GPT-4

can easily correct the majority of these issues when they were pointed out by the user. Often, the process of correcting the issue through prompts and responses was faster than actually adjusting the code manually, making LLMs a useful design partner.

One pervasive issue that seemed more difficult to correct was the fact that GPT-4 struggled to position the primitives in 3D space. In particular, GPT-4 frequently seemed to forget that OpenSCAD positions elements relative to the *center* of a given primitive, rather than an external point on the primitive (for example, the lower left corner). GPT-4's arrangements were frequently incorrect due to this issue. When GPT-4 is reminded of this convention, it does generally alter the design, but it is not always able to correct the issue. If sufficiently many rounds of local edits prove unable to address the alignment issues, we found that it was generally more effective to direct GPT-4 to disregard all existing measurements, and re-derive the elements' positions from scratch (see [Figure 3](#)).

Overall, we find that GPT-4 is able to generate reasonable OpenSCAD models from high-level input. However, the design specifications that emerge on the first attempt are rarely fully correct, so users should expect to engage in some amount of corrective feedback or iteration in order to attain the desired result.

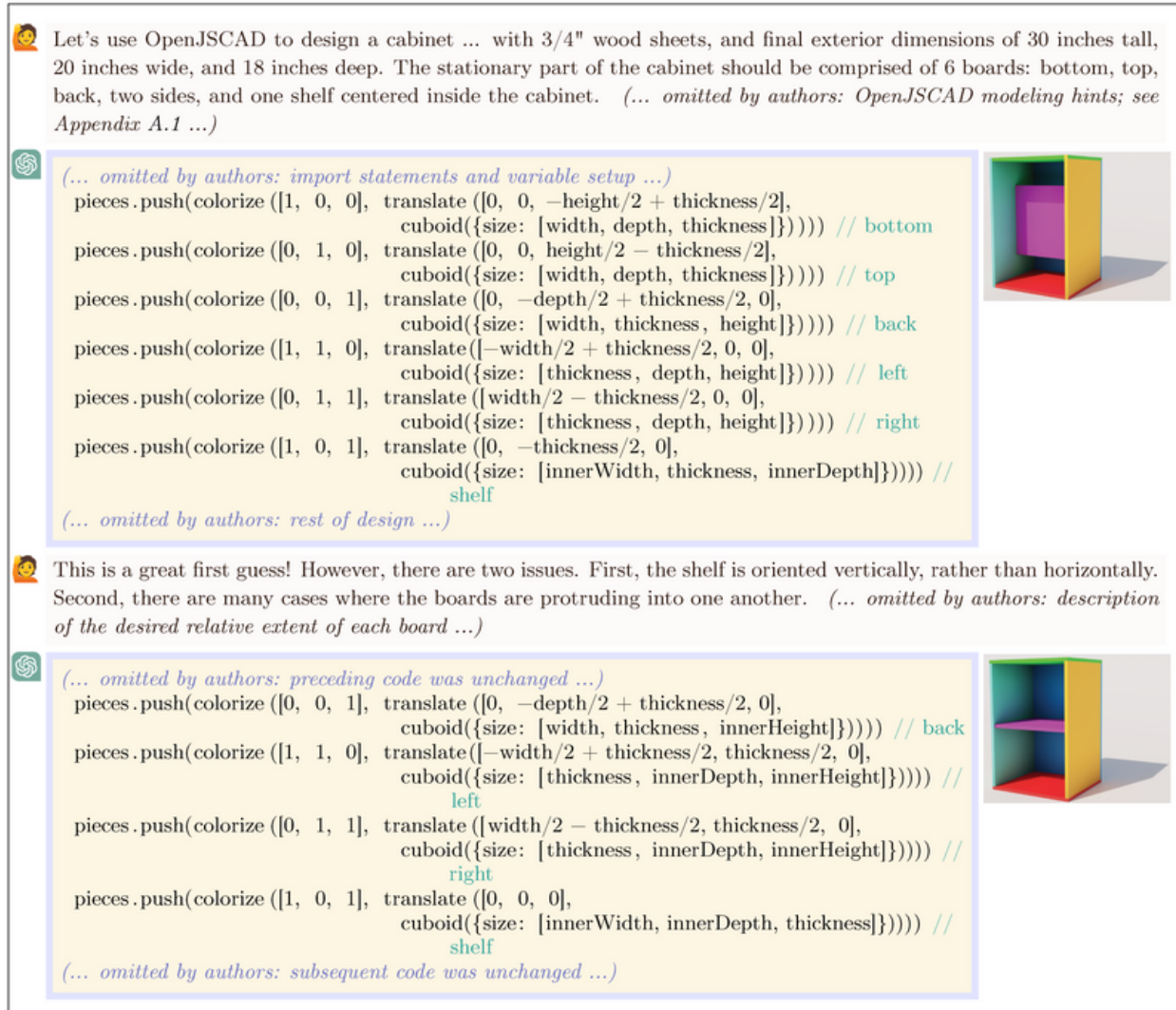


Figure 2. GPT-4's cabinet design process. Simple cabinet design, from initial prompt to the final fabricable result, in which all elements have the correct orientation and no components overlap. GPT-4 reached the final result in eight messages (four each of prompt/response).

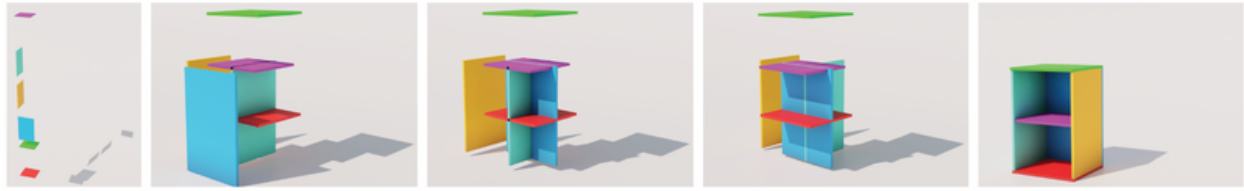


Figure 3. Alternate cabinet design process by GPT-4. From left to right, these images show a second example of GPT-4’s design process for the simple cabinet, beginning from the same prompt as [Figure 2](#). This example highlights the widely varying paths that GPT-4 may take to construct a design. GPT-4 reached the final result after 16 messages (eight each of prompt/response). Up until the fourth displayed result, the user prompts’ phrasing focused on improving upon the previous result. However, after failing to make progress, we asked GPT-4 to disregard the previous attempts and recompute the elements’ positions from scratch; this quickly resulted in a valid cabinet, as shown in the final image.

6.1.2. Summary Discussion

GPT-4 is capable of generating designs based on high-level text input, even across a wide variety of representations and problem domains. We note that several of GPT-4’s capabilities and limitations remain consistent independent of the representation. For example, in all cases (see [Appendix B](#)), GPT-4 is able to generate sensible, well-structured code with semantically meaningful variables and comments. Moreover, independent of the representation or the problem domain, GPT-4 consistently shows superior performance with respect to the high-level, *discrete* elements of a problem (for example, identifying the correct type and quantity of each primitive/operation) as opposed to the lower level continuous parameter assignments (e.g., correctly positioning the primitives relative to one another). A more detailed discussion of capabilities, limitations, and opportunities will follow in [Section 6.4](#). For now, we rely on the similarities between various representations to justify a reduced scope for our future experiments. In particular, moving forward, we study each question with respect to only a subset of the design representations and domains introduced above.

6.2. Interpreting and Respecting User Control (Q2)

The above examples demonstrate GPT-4’s ability to generate a design based on very high-level semantic input. However, we also wanted to test its ability to generate designs that adhere to a specific user-given intent. This section also tests whether GPT-4 is able to overcome its own potential biases induced by the training data, in order to generate something that truly adheres to a user’s specified constraints—whether or not those constraints match the ‘common’ form of a given design target. In particular, we choose to study whether GPT-4 is able to (1) understand and respect semantically meaningful spatial constraints, and (2) incorporate specific prefabricated elements into a design.

Here, we present investigations with both our running cabinet and quadcopter examples, and provide supplementary experiments in [Appendix B](#).

6.2.1. Spatial Constraints

Through the general experiments above, GPT-4 has already shown some capacity to respect high-level spatial constraints, such as a design element’s absolute size or its position relative to another element of the design. GPT-4’s compliance with such requests was frequently flawed at the outset, but the results were generally workable after some amount of interactive feedback. This section aims to explore the types of constraints GPT-4 is able to natively understand, and how we might best interact with GPT-4 in order to improve the chance of successful compliance with such constraints.



Figure 4. Building a cabinet with a door. GPT-4’s attempt to build a cabinet similar to that from [Section 6.1.1](#), with the addition of a simple door (orange) that has a handle (dark grey) on the right-hand side. GPT-4 quickly fixes the position of the cabinet’s primary pieces (e.g., the yellow and cyan side panels), but it struggles to correct the door. GPT-4 must be iteratively prompted to fix the door orientation, the relative door placement, and the handle’s placement and protrusion into the door. GPT-4 is able to arrive at a suitable design after several iterations of user feedback.



Figure 5. Adding a door to an existing cabinet. We provide GPT-4 with the initial cabinet design from [Section 6.1.1](#) (semitransparent blue), then ask it to add a door (orange) with a handle on the right-hand side. Despite beginning from a largely complete model, GPT-4 still has difficulty placing the door and handle correctly.

As an initial experiment, we explored whether GPT-4 is able to construct a version of the previous cabinet design that includes a door and a handle (see [Figure 4](#)). We started from a fresh chat, and provided GPT-4 with a prompt similar to the one described in [Section 6.1.1](#), asking for a cabinet to be built from scratch. However, this time, we also request a door at the front of the cabinet, with a handle on the right-hand side of its outward-facing face. As shown in [Figure 4](#), GPT-4 initially struggled to position several of the cabinet elements—particularly the side panels and the door. Although GPT-4 corrected the position of the side boards immediately, GPT-4 continued to have trouble placing the door, as it was oriented incorrectly relative to the rest of the design. When reminded that the door should be oriented vertically, GPT-4 was able to comply with

the request, but the corrected position was still not fully suitable, as the door coincided with the cabinet's side panel. After another reminder that the door should reside at the front of the cabinet, with the handle on the right so it could be attached with hinges on the left, GPT-4 was able to place the door correctly. However, the handle remained ill-positioned because it was located on the left-hand side, and was protruding into the door panel. After two additional prompts, GPT-4 was able to correct the position to the left-hand side. To correct the protrusion issues, GPT-4 needed three more prompts. During these iterations, GPT-4 moved the handle fully to the *inside* of the door; it needed explicit reminder that the handle should be placed on the *outside* of the door.

With a fresh GPT-4 session, we also tried providing the previous OpenJSCAD specification of the cabinet as part of our input prompt, then asking GPT-4 to modify the existing design such that it contained a door and a handle, as before. Despite the different starting points, GPT-4 followed a similar trajectory, as shown in [Figure 5](#): the door was initially aligned incorrectly, because it coincided with one of the side panels; after one prompt, GPT-4 was able to correct the door placement. However, despite GPT-4's explicit assertion that `the handle is also placed on the right side of the door's exterior face`, the handle remained on the left. Finally, after another prompt, GPT-4 was able to correct the handle position such that it was on the right rather than the left.

The way in which GPT-4 dealt with the underspecified handle request also proved interesting. In [Figure 4](#), GPT-4 opted for an additional cuboid that would be unioned into the final design. By contrast, in [Figure 5](#), GPT-4 opted to create the handle by subtracting a small cuboid from the door panel. In still other examples, GPT-4 refused to add the handle, and instead offered the following disclaimer: `Note that the handle for the door is not included in this script, as its size, shape, and position would depend on additional details not provided. This would likely require additional modules, such as cylinder from @jscad/primitives, and might be added as an eighth component in the main function.`

These interactions provide a promising basis for interactive user control of the design, but the process is somewhat tedious at the moment, as GPT-4 requires very explicit instructions about the design or correction intent. The addition of highly detailed user constraints also seems to confuse GPT-4 to an extent, as it seems to 'forget' the larger context of the design in the process, so it must be frequently reminded.

6.2.2. Incorporating Prefabricated Elements

It is also common to design an object around specific premanufactured elements, such as hinges, brackets, or motors. We explore the possibility of using GPT-4 to source the parts in [Section 8.3](#)—at that time, we explore whether GPT-4 can identify the required part categories, provide options, and/or select a set of options that are compatible with one another and the intended overall design.

For now, we assume that the user has a specific (set of) part(s) in mind that they would like to incorporate into their design. Then we investigate whether, given these components, GPT-4 is able to (1) build a reasonable

proxy of this design, then (2) effectively use it as a module within a larger assembly.

6.2.2.1. Cabinet With Standard Hardware

To make the cabinet design more stable, a designer may wish to include extra support brackets to work with. Many prefabricated variations of these brackets exist, and they are inexpensive and readily available. Given this, it does not make sense to design or manufacture these parts via GPT-4. Rather, we'd like to incorporate instances of a pre-fabricated version. To do this, GPT-4 must first build a proxy of the part, place the proxies throughout the design appropriately, and adjust the remaining elements of the design to accommodate these components.

For our first experiment, we chose to incorporate the Prime-Line 1/4 in. Nickel-Plated Shelf Support Pegs from Home Depot into our design. We provided GPT-4 with a URL to this part's listing on the Home Depot website, which contained a text description of the item and the schematic diagram pictured in [Figure 6](#) (left). We then asked GPT-4 to build a simple geometric proxy that we could incorporate into our design as a placeholder. As shown in [Figure 6](#) (right, top), GPT-4 was able to infer and generate the appropriate primitives (one cylinder for the peg and two cuboids for the L bracket). However, it was not able to correctly scale, orient, or position the elements. In an effort to test GPT-4's understanding of the structure, we asked it to describe the structure in its own words. Although it gave a reasonable description of the bracket, there was little improvement in the result when it was asked to improve the script accordingly. Thus, even with several iterations of user feedback, GPT-4 was unable to construct this shape from high-level third-party (URL) or user input.

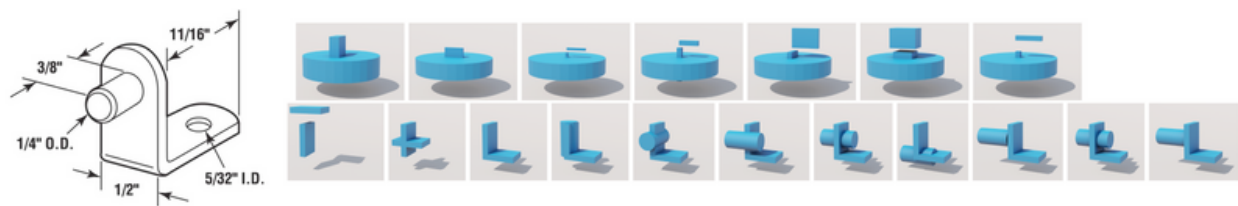


Figure 6. GPT-4's Attempts to create a proxy for an L-bracket. **Left:** Image of the desired prefabricated part, to which GPT-4 was provided a link. **Right, Top:** GPT-4's attempt to design a proxy based on the knowledge it gleaned from the provided product webpage, with iterative high-level user feedback. Although GPT-4 identified the primary structures (two cuboids for the L and a cylinder for the peg), it was unable to arrive at a proper design in this manner. **Right, Bottom:** GPT-4's process for designing a proxy for the part from scratch with explicit user guidance about the structure and its dimensions.

Ultimately, we had to provide GPT-4 with an explicit description of the structure that we wanted. Moreover, we found that even with an explicit description, GPT-4 was unable to generate the correct shape when provided with all directions at once. Instead, we had to create the shape in an iterative fashion, beginning with the L bracket and then adding in the peg, as shown in [Figure 6](#) (right, bottom). Eventually, it was able to generate the structure and consolidate the instructions into a high-level module called `createBracketWithPeg`, as desired.

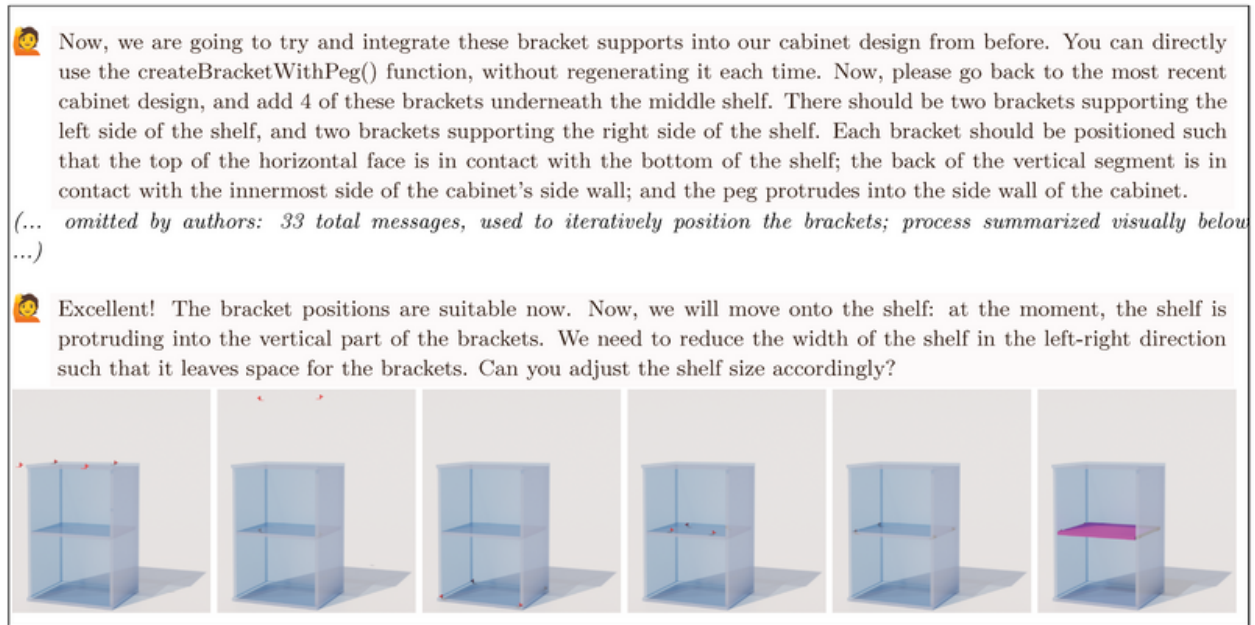


Figure 7. Process for integrating L-brackets (red) into an existing cabinet design (semitransparent blue) using GPT-4. It takes 34 messages to position the brackets appropriately (17 each of prompt/response), but once this is done, GPT-4 is able to efficiently generate a modified shelf (pink) to accommodate the placed brackets (six messages; three each of prompt/response).

We then provided the module `createBracketWithPeg` as an input to GPT-4, and asked it to incorporate these structures into the design, as detailed in [Figure 7](#). In particular, we asked for four brackets under each shelf, with the pegs protruding into the cabinet's side walls, the back face of the bracket's vertical leg in contact with (but not protruding into) the side wall, and the top face of the bracket's horizontal leg in contact with (but not protruding into) the bottom face of the shelf. We initially tried to complete this experiment in a single continuous chat that (1) designed the cabinet, (2) designed the L-bracket, and then (3) incorporated the brackets into the cabinet. However, we found that after the extended discussion regarding the L-bracket design, GPT-4 seemed to have completely forgotten its cabinet specification. Despite multiple prompts, it was unable to recover the previous design. Instead, we directly provided GPT-4 with the L-bracket module and its prior cabinet design, and then asked for a modification. This approach was far more successful. Overall, we found that GPT-4 was able to instantiate the correct number of brackets, but it struggled to rotate and position them appropriately. After several user prompts, GPT-4 was able to successfully place the brackets in their locations. Finally, we asked GPT-4 to adjust the shelf in order to (1) not protrude into the brackets, and (2) incorporate some additional allowance so the shelf could easily fit between the supporting brackets in a physical assembly. GPT-4 was able to complete these requests without issue.

Overall, although GPT-4 initially struggled to build a proxy of the prefabricated part we had in mind, GPT-4 seemed quite capable of incorporating the completed proxy into a given design, as desired.

6.2.2.2. Quadcopter

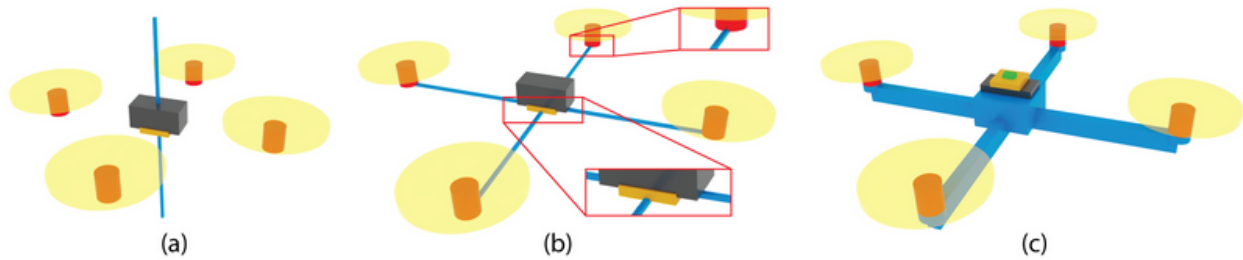


Figure 8. A quadcopter designed with the aid of GPT-4. The motors are colored in red. The propellers are in yellow. The battery is in dark grey. The frame is in blue. The dark yellow box is the controller and the green box is the receiver.

Designing a quadcopter involves integrating prebuilt elements like the motor, propeller, and battery. Detailed sourcing of these parts will be addressed in a later section ([Section 8.3](#)). Once these components are sourced, the frame must be designed to accommodate their dimensions. We will explore how GPT-4 can assist with this task.

However, enabling GPT-4 to accurately represent these parts is not straightforward. To simplify the task, parts are represented as either a box of dimensions $w \times h \times d$ or a cylinder with radius r and height h . GPT-4 can handle these representations well, as demonstrated in [Appendix B.2](#). Rather than having a single function that creates a primitive and translates it as in [Appendix B.2](#), we introduce three functions for ease of design: `createBox(w, h, d)`, `createCylinder(r, h)`, and `place(item, x, y, z, a)`. The first two functions generate a box or a cylinder at origin (0,0,0), while the third rotates and moves the item to desired coordinates.

Subsequently, we task GPT-4 with creating a design that integrates these parts using only the above functions. The primary element GPT-4 must design is the frame, which should hold the selected components. Initially, GPT-4 produced a correct textual design, but struggled with the geometric representation, similar to [Appendix B.2](#). It understood the quadcopter structure, but had issues with part positioning and orientation which can be seen in [Figure 8\(a\)](#). In particular, GPT-4 could not properly orient the frame and intersect parts appropriately in its first attempts. By guiding GPT-4 in correcting these issues, we achieved a near-correct quadcopter design ([Figure 8\(b\)](#)).

The initial frame design was not practical because it was directly attached to the motor cylinder and insufficient to hold components like the battery, controller, and signal receiver. To address this, we asked GPT-4 to incrementally implement specific solutions, such as adding a cylinder base under each motor and a box body to reinforce the frame bars and house remaining parts. After minor adjustments, we arrived at a valid design, which then undergoes further testing in a simulator or real-world conditions ([Figure 8\(c\)](#)).

Throughout the design process, GPT-4 demonstrated proficiency in textual design analysis but struggled with mathematical and physical concepts such as collision and structural integrity. Thus, human guidance remains

crucial in these areas.

6.3. Incorporating Abstractions such as Modular/Hierarchical Designs (Q3)

As we have seen from previous examples, GPT-4 is inclined to use some abstractions like variables by default. It is also clear that GPT-4 is well suited to the use of modular or hierarchical design, as in the case of the prefabricated L-brackets that it was able to instantiate several copies of, and distribute throughout a design. However, there are often instances where a user might want to impose their own specific modules—for example, a certain hierarchical grouping may facilitate easier debugging or cleaner code.

To test GPT-4’s abilities in this area, we revisit the cabinet example, and try to modify it such that it contains multiple shelves. Because we have already incorporated prefabricated brackets, this modification is nontrivial, as GPT-4 must instantiate and position the appropriate number of shelves *and* all associated support brackets. We began by directly asking GPT-4 to make this modification on top of the existing code, by generating two evenly spaced shelves within the cabinet instead of one. GPT-4 correctly identifies the elements that must be duplicated, and it instantiates the correct number of them. However, it is unable to correctly adjust the position of each module; after the initial request, neither the shelves nor the brackets were in reasonable locations. It took four additional user prompts to correct the relative positions of these components. After this correction, GPT-4 did seem able to generalize its logic directly to generate cabinets with a varying number of shelves. However, the code itself is fairly convoluted.

To avoid these issues, it may be more natural to consider a shelf with its appropriate supporting brackets as a single module. This way, the entire ‘subassembly’ could be instantiated and positioned as a unit on future calls. We asked GPT-4 to implement this plan, by requesting the creation of a module named `supportedShelves()`, which instantiates and appropriately positions a shelf and its associated support brackets within the design. Then, we asked GPT-4 to refactor the original script such that it used the new module to generate a cabinet with two evenly spaced shelves. The initial response had a minor compilation error, a shelf tolerance issue, and a bracket alignment issue, as before, but each of these issues were immediately corrected after a single user prompt.

Overall, the approaches resulting from both experiments seem equally effective and flexible once they have been fine-tuned. Thus, we conclude that GPT-4 is able to effectively create and use modules, whether they are explicit (e.g., in the form of a function, as in the second experiment) or implicit (e.g., in the form of a for-loop, as in the first experiment). However, it seems as if the explicit module made it slightly easier for GPT-4 to reason about a challenging alignment problem. Moreover, it is useful to know that users can effectively request this kind of hierarchical refactoring, as most human programmers/designers would generally find it easier to reason over a function in this scenario.

6.4. Discussion

In this section, we elaborate on the key capabilities (C), limitations (L), and dualisms (D) previously outlined, particularly as they relate to the domain of generating designs from natural language specifications.

C.1 Extensive Knowledge Base in Design and Manufacturing: Within the language-specified design space, GPT-4 exhibited proficiency in supporting high-level structure and discrete composition. For instance, GPT-4 consistently generated the correct primitives (type and quantity) for a given task, regardless of the specific design language it was using. GPT-4 also demonstrated a capacity for interpreting and auto-completing underspecified prompts, as in the case of the CSG table example (see [Appendix B.2](#)), where GPT-4 inferred and provided reasonable values for a set of missing parameters. Finally, GPT-4 generated readable, explainable, and maintainable code that contained descriptive variable names and comments, along with appropriate modularity and other high-level structural elements.

C.2 Iteration Support: Even when GPT-4 did not immediately arrive at a suitable design solution, it often succeeded in rectifying errors after a reasonably small number of user interactions. For example, it was able to successfully adjust the placement of the cabinet handle after a handful of additional prompts. The ability to engage in iterative design is also very helpful when building up complex structures such as the wheeled robot from [Appendix B.5](#) or the L-bracket proxy discussed in [Section 6.2.2.1](#), because users can start with a simple prompt, then iteratively increase the complexity to arrive at a suitable result.

C.3 Modularity Support: GPT-4 effectively incorporates modules and hierarchical structures, using natural language as a powerful tool for conceptualization and orientation.

L.1 Reasoning Challenges: Spatial reasoning posed a significant challenge for GPT-4. Well-crafted domain-specific languages may be able to mitigate this issue. We noted specific difficulties with constructive solid geometry due to the computational requirements for object placement. Sketch and extrude languages that utilize reference points can minimize this challenge to an extent, as they offload the computation to reference resolution. This approach is effective for simpler designs but falters when managing complex sequences of transformations. As discussed in the sketch-based car example from [Appendix B.4](#), we found that DSLs that balance the benefits of reference-based language with global positioning information may be more effective.

GPT-4's lack of spatial awareness also created difficulties with constraint handling, such as when GPT-4 was asked to ensure that elements were non-overlapping. We found that iterative refinements and careful prompting often provided a workaround for these issues. For example, GPT-4 typically failed to respect 'non-overlapping' constraints, but it generally responded well to the instruction that some element should be "in contact with (but not protruding into)" another element.

L.2 Correctness and Verification: GPT-4 is not able to reliably verify its own output, and it frequently makes contradictory claims. For example, when asked to place a handle on the right side of the cabinet structure in

[Section 6.2.1](#), GPT-4 frequently placed the handle on the left-hand side of the cabinet, then immediately declared its design a success by (erroneously) affirming that the handle was on the right, as requested. This seems to suggest that external verification tools may be helpful, particularly in cases where the contradictions are less obvious.

L.3 Scalability: GPT-4’s success seems to decline as the number of simultaneous requests increases. For example, it is best to issue one to two constraints or correct one to two issues at a time, rather than trying to issue several constraints or correct several issues at once. Similarly, GPT-4 encountered challenges when interpreting high-level information to build proxies for more complex designs all at once; instead, the models must be built iteratively, with gradually increasing complexity. This iterative modeling was most effective when the user provides explicit instructions about both the aspects that should change, as well as the aspects that should remain unaltered (either because they are already correct, or because they will be addressed later). Despite GPT-4’s initial difficulty creating complex models, GPT-4 is able to effectively use and combine existing modules to create more intricate models.

L.4 Iterative Editing: As discussed in [Section 6.2.2](#), GPT-4 seems to exhibit limited memory and attention span. In particular, it often ‘forgets’ things from previous messages. We address this by occasionally reminding GPT-4 of its previous input/output, either by asking it to summarize a previous interaction/finding, or by explicitly including a prior result as a starting point in our prompt.

D.2 Unprompted Responses: GPT-4 is frequently able to recognize and address underspecified problem statements. For example, in the CSG table specification ([Appendix B.2](#)), GPT-4 correctly inferred the need to assign a tabletop thickness value. Similarly, when augmenting the cabinet with a door and a handle in [Section 6.2.1](#), GPT-4 responded with several distinct approaches for handle design. This can be powerful, as it may alert the user to parameters or variations that may otherwise have gone overlooked; then, users have an explicit opportunity to consider and refine the specification accordingly. Moreover, it allows users to undertake a design process and begin receiving feedback without first needing to craft a perfect specification or prompt. However, if GPT-4 confidently hallucinates a particular solution to an underspecified aspect of a design problem—rather than explicitly prompting the user to consider a range of options—it may limit and/or bias their exploration in unexpected ways.

7. Design Spaces via Natural Language and Extrapolation

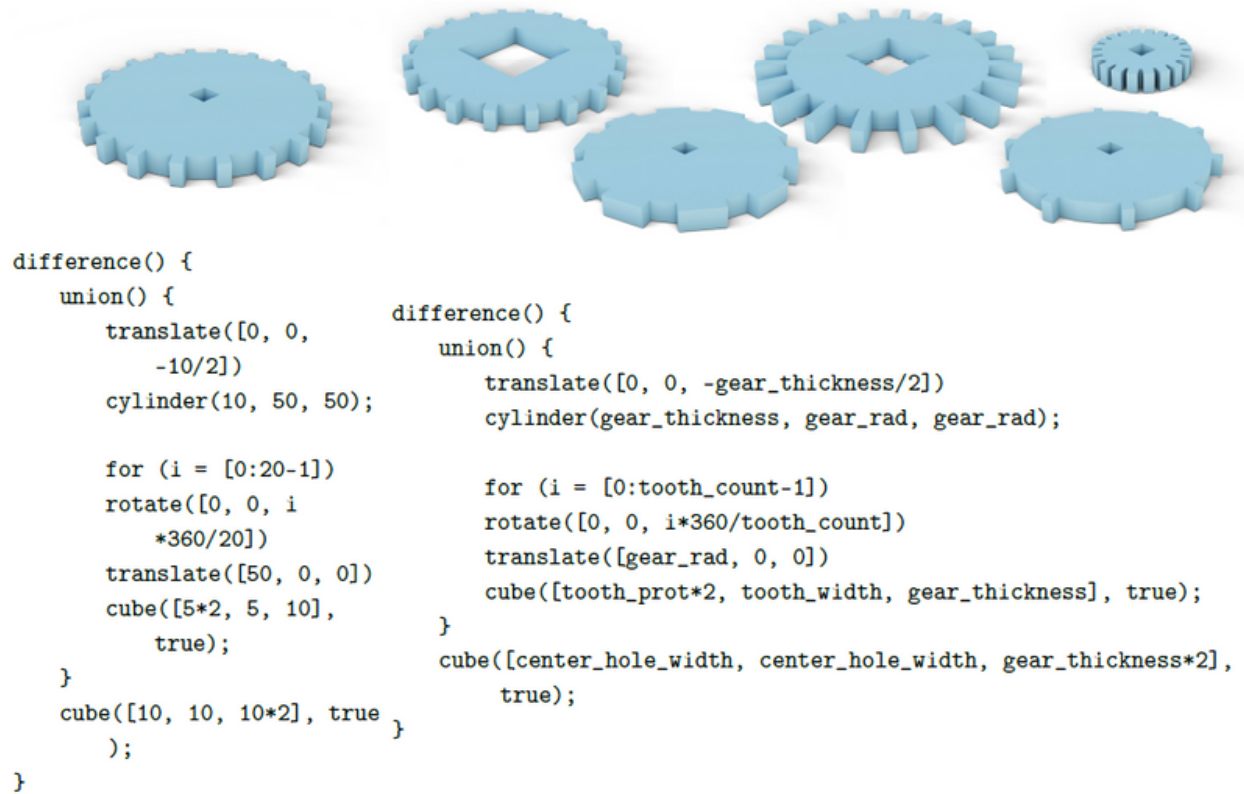


Figure 9. Gear design space. The same gear (**Top left**) can be constructed with different design representations. Here, we have an OpenSCAD design (**Bottom left**) and a parametric OpenSCAD design (**Bottom right**) where the degrees of freedom (**Top right**) have been exposed via design variables.

A design is a sequence of construction operations which take input values and which modify the current state of the design. These input values can directly be represented as numbers. For example in [Figure 9](#) (left), the design of a 3D gear is constructed by directly using 3D coordinates and dimensions. While this representation has the merit of being direct, without any references to previous code, it does not expose the degrees of freedom of a design. To modify the thickness of the gear, we have to modify several input values at once to obtain the desired 3D model. The introduction of *design parameters* in [Figure 9](#) (right) makes this change easier by modifying a single variable, namely `gear_thickness`. We call this representation a *parametric design*. Note that design parameters can be continuous or discrete, for example, `gear_thickness` or `tooth_count`, respectively.

To explore different design variations, either manually or automatically, having a parametric design is not enough. We still need to know which specific values we can assign to the design parameters. For this, we introduce lower and upper *bounds* for each design parameter. Each design parameter can take any value within

its specific bounds. Together, a parametric design and parameter bounds define a *design space*, which is the set of all possible design variations.

Design spaces are an import tool to understand what a design can accommodate for. This is important for both the manual and automatic optimization of designs. With this in mind, we want to investigate the following questions:

- **Q1** Can GPT-4 create a design space from text?
- **Q2** Can GPT-4 create a design space from an existing design?
- **Q3** Can GPT-4 create a design space from multiple designs?
- **Q4** Can GPT-4 explore a given design space?

For each of these questions, we want to find out what is currently possible and what seems to be beyond its capabilities.

7.1. Generating a Design Space From Text (Q1)

In [Section 6](#), we showed that GPT-4 is capable of generating designs. The next step toward generating a design space is to test if it can also generate *parametric* designs. Here, we summarize the findings of our experiments, and defer readers to [Appendix C.1.1](#) for detailed support of our claims. To enforce the generation of parametric designs in our prompts, we ask it to `explicitly use high-level design parameters` and to use `as few variables as possible`. It should be noted that GPT-4 often introduces variables to improve readability by itself, without explicitly being asked to do so. However, we found that including this in our prompts *always* resulted in parametric designs.

We also notice that when asking for a simple design and asking for a parametric design of the same object, there are generally fewer mistakes in the reuse of certain dimensions.

Bounds

Length = [3,5]

Height = [1,2]

Width = [2,3]

Radius = [0.3,0.6]

Constraints

Length < *Height*

Width < *Length*

Radius < *Height*

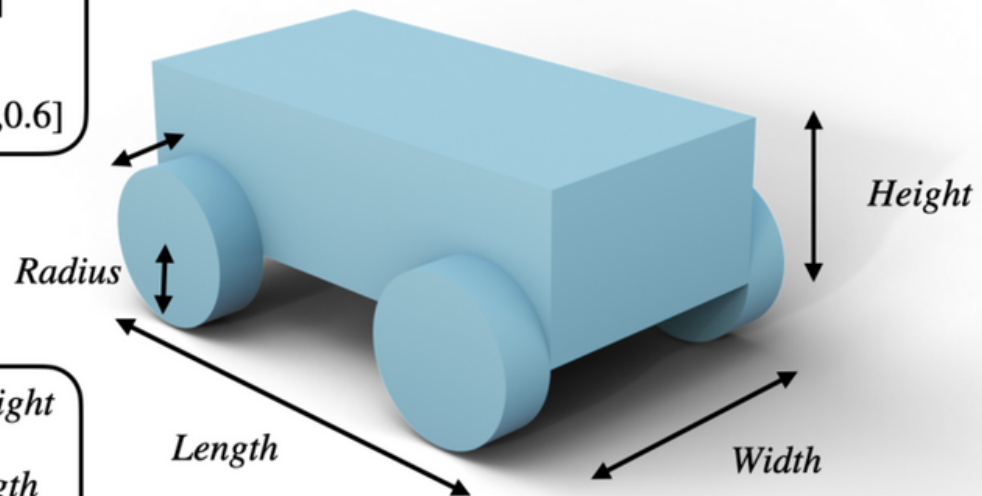


Figure 10. Car parameter bounds. GPT-4 generates semantically based parameter bounds and constraints based on this simplified car design.

To generate a design space, we need parameter bounds. When asked for lower and upper bounds for parameters, GPT-4 proposes bounds that are based on typical proportions of the designed object. This implies that the scale is often arbitrary but that bounds are semantically reasonable relative to each other. For example, when asked to design a parametric car with exposed parameter bounds, GPT-4 returns lower and upper bounds and arguments for these bounds in terms of inequalities, see [Figure 10](#). According to GPT-4, the width of the car body should be less than the length but larger than the height and the radius for the cylindrical wheels should be less than the height of the car's body so the wheels don't exceed the height of the body. These constraints between design parameters can also be queried in the form of actual inequalities, which is useful for downstream optimization when combined with parameter bounds.

However, these bounds are based on semantic knowledge about the object and not on the geometric design sequence. For example, for a pen holder, the angle of a rotated cylinder will get a lower bound of 45° to prevent any pen from falling out, but not to prevent the 3D object from creating unwanted intersections with other parts. Constraints in real-world design sequences often need to also consider purely geometric aspects of a design.

7.2. Generate a Design Space from an Existing Design (Q2)

Given the current limitations of creating designs and design spaces from text prompts alone, it is interesting to understand how GPT-4 can create design spaces from existing designs, made by human designers. Just as regular code, input designs for GPT-4 can vary in quality of semantic annotations and comments about what is being constructed. For all of these inputs, we are interested in how easy it is for GPT-4 to create a design space, that is, a parametric design with parameter bounds. We investigate how helpful semantic context is to parametrize designs. For the prompts of our experiments (see [Appendix C.1.1](#)), we have found that we get more consistently a good parametrization when we include that it should expose high-level design parameters while using as few variables as possible and that it should keep the same program structure and the resulting input values to modeling functions. These constraints prevent it from slightly modifying operator input values to extract fewer design parameters.

When given a design with no semantic context, we observe that GPT-4 exposes design parameters based on equivalence between numerical values and based on which design operators these values were used in. However, when we provide semantic context, such as the name of an object that is being modeled, design parameters are exposed that tend to be more semantically useful for modifying the design. While these semantic reasoning capabilities are encouraging, experiments further found that GPT-4 is often easily confused by the final effect of a series of geometric transformations.

7.3. Can GPT-4 Create a Design Space from Multiple Designs? (Q3)

Design spaces based on a single design are useful to explore the family of possible shapes generated by varying the design parameters.

However, sometimes a designer might want to make more structural changes: for example, inspired by another design of the same object class, they may want to interpolate between multiple designs. Interpolating two designs can be difficult to achieve and there are a number of difficult questions that arise: Are the two designs modeled in a similar way? Do they have the same dimensions and, if not, how do you match the dimensions between two subdesigns? Do you have to add extra operations to combine two parts? Can you actually extract a subpart of an object from a design? If you cannot exactly extract a subdesign, can you design something that is *inspired* by two design sequences? How do you accurately refer to two subdesigns in a text prompt?

To investigate if GPT-4 can help with design interpolation, we tested three different design scenarios. All of the designs were presented to GPT-4 in our sketch-based parametric CAD DSL, explained in [Section 6](#) and [Appendix B.4](#). Detailed experiments can be found in [Appendix C.1.2](#).

In the first scenario, we ask GPT-4 to mix two designs that serve the same semantic purpose but have differing geometric features. In the second experiment, we present GPT-4 with two designs that are semantically similar but have differing numbers of repeated geometric features, and ask it to generate more designs from the

representative class. The third experiment mirrors the second, with similar modeled objects with differing numbers of internal features; however, in the third experiment, GPT-4 must generate new candidate designs while respecting complex structural constraints imposed by differing numbers of parts.

In all cases, GPT-4 is able to interpolate across design spaces from few class instances in a meaningful way. We find these examples promising, as they show how GPT-4 manages to combine its general knowledge about part relationships and its coding abilities. One of the observed limitations is the ability to extract long sub-sequences and to detect which other parts are still important for plausible interpolation.

7.4. Exploration of a Given Design Space (Q4)

A design space is conceptually useful to reliably generate variations of a given design. However, coming up with parameters which represent *meaningful* design variations can be a time-consuming iterative process. In experiments shown in [Appendix C.1.3](#), we present GPT-4 with a parametric design, and ask it to generate `parameter bounds` and `parameter constraints`. We observe that the proposed parameter settings respect the previously generated bounds and constraints and that they lead to distinct 3D models, for which it generates plausible semantic labels.

7.5. Discussion

In this section, we summarize the key capabilities (C), limitations (L), and dualisms (D) specific to the creation and manipulation of design spaces.

C.1 Extensive Knowledge Base in Design and Manufacturing: We observe that we can leverage GPT-4's semantic knowledge base to create parameters, bounds, and constraints for text-based designs and already existing designs. Additionally, GPT-4 can be useful for finding semantically meaningful design variations in a given design space.

C.3 Modularity Support: We observe that GPT-4 can interpolate existing designs by extracting and adapting subdesigns based on their program representations. Interestingly, even when designs are not presented in a modular fashion, it tries to recognize and abstract submodules in input designs.

L.1 Reasoning Challenges: The design spaces created by GPT-4 are based both on semantic knowledge and on code interpretation. However, it does not take into account geometric considerations, such as intersecting or nonconnecting parts. As a result, generated parameter bounds can create invalid geometry and it has proven difficult to make GPT-4 correct these. However, in general, the generation of valid parameter bounds and constraints is a difficult problem for which many approximations have been proposed ([Mathur & Zufferey, 2021](#)).

L.3 Scalability: The interpolation task revealed that GPT-4 has limited capabilities to infer what parts of a design should be linked to a semantic part specified in a prompt. One promising future direction to manage

increasingly complex designs is to make them increasingly modular by adding intermediate levels of abstraction.

D.1 Context Information: We observe that the generation of correct parametric designs and the reparametrization of already existing designs can be improved by providing semantic cues, such as the name of the modeled object. As seen in [Section 6](#), GPT-4 creates designs that contain a lot of semantic information and it generally performs even better when using meaningful variable names. Leveraging this aspect in the generation of design spaces and throughout other aspects in the design process should prove extremely useful.

8. Design-for-Manufacturing

The utilization of LLMs in the context of design for manufacturing provides a broad range of applications that have the potential to enhance the design and manufacturing process of different parts and assemblies. One useful application of LLMs involves leveraging their pattern identification and language interpretation capabilities to imitate a manufacturing expertise bank that can be tapped into during various parts of the design and manufacturing stages. Furthermore, because LLMs such as GPT-4 have the ability to create programs and find and interpret patterns in text, it can potentially be used to generate and alter design and manufacturing files. Currently, DFM is often accomplished by human expertise with the aid of CAD software. Engineers and designers review design plans and use their industry experience to suggest alterations that would improve manufacturability. The CAD software then allows these alterations to be modeled. The replacement of human manufacturing knowledge with GPT-4 in this context could streamline the design for manufacturing process, offering more consistent, scalable, and efficient decision-making, which is not limited by individual human capacity.

In this section, we propose multiple ways that this new manufacturing expertise bank could be used in design and manufacturing, as shown in [Figure 11](#). GPT-4 can be used to select optimal manufacturing techniques based on a part's features. Furthermore, it can propose and implement modifications to a design to improve its manufacturability, ultimately leading to more efficient production processes. Additionally, this idea can be extended to part sourcing by leveraging the model's reasoning capabilities to identify potential suppliers based on the part's desired function and performance. Finally, it could be used to develop manufacturing instructions for various processes. To understand GPT-4's ability to alter designs based on manufacturing/sourcing constraints, we pose the following questions:

- **Q1** Given a part geometry, production run, and other desired outcomes, can GPT-4 select optimal manufacturing processes?
- **Q2** Given a manufacturing process, can GPT-4 directly suggest and make design alterations to a parts file based on constraints driven by the process capabilities?
- **Q3** Given a desired functionality and geometric specifications, can an LLM find a source for a part that fits those specifications?

- **Q4** Given a design, can an LLM create a set of manufacturing and assembly instructions?

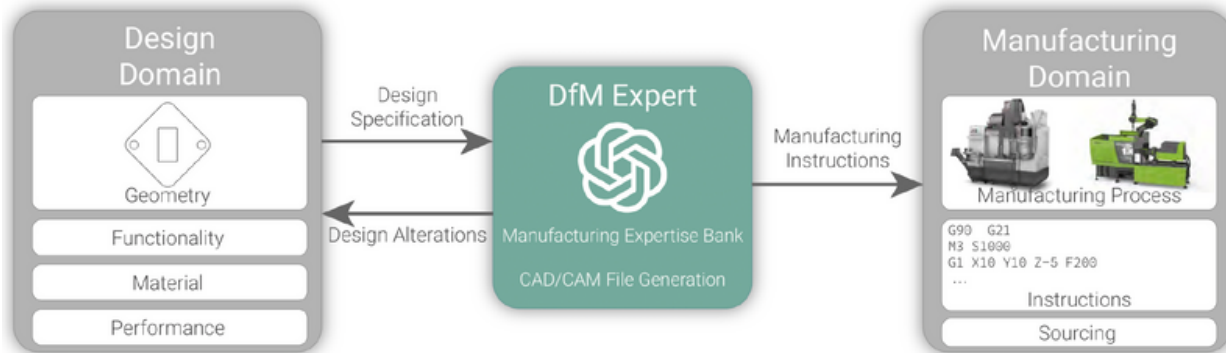


Figure 11. Integration of GPT-4 into Design for Manufacturing (DFM): GPT-4 can be used to augment the DFM Process when designing a part.

8.1. Finding Optimal Manufacturing Process (Q1)

To test these capabilities, we tasked GPT-4 with advising on identifying an optimal manufacturing process for a part with different input geometries and materials, where different tolerance and quantity requirements were varied. Part geometry was described using OpenJSCAD files, and GPT-4 was tasked with selecting an optimal manufacturing process; in most cases, the selection was open-ended, but in one case, we asked GPT-4 to choose from a finite list of options. Detailed experiments can be found in [Appendix D.1](#).

GPT-4 was successful at selecting an optimal manufacturing process for three out of the four cases. For three of the cases, GPT-4 selected the optimal process that was approved by an expert. However, in a case where polytetrafluoroethylene (PTFE) material was specified, GPT-4 suggested an injection molding process, which is not suitable for that material. In all cases, GPT-4 initially only provided a range of manufacturing options; it required additional prompts to arrive at the optimal manufacturing process selection.

8.2. Design Alterations for Manufacturability (Q2)

In this section, we assessed GPT-4's capability to enhance designs for better manufacturing optimization. To accomplish this, we included the text of an OpenJSCAD file in the prompt, allowing GPT-4 to analyze and modify it accordingly.

In our experiments (see [Appendix D.2](#)), the iteration process began with GPT-4 identifying any manufacturing complexities within the design features. Since an LLM interprets text, GPT-4 interprets the text of the OpenJSCAD file, rather than the geometry that is rendered once compiled, which humans interpret. After GPT-4 identifies any complexities, we instructed it to adjust the geometry of the OpenJSCAD file to address the challenging aspects by directly changing the text of the OpenJSCAD file.

Although GPT-4 accomplished these tasks with a moderate degree of success, there were a few inaccuracies. GPT-4 tended to misunderstand a number of geometric features described in the OpenJSCAD files. While GPT-4 could correct mistakes highlighted by users, such corrections often introduced new reasoning errors, and could require several rounds of iteration to correctly identify all potential machining issues. However, once the issues were correctly identified, GPT-4 was able to improve the design for easier machining, including under tooling specifications.

8.3. Part Sourcing (Q3)

The massive data set backing LLMs contains some specialized knowledge about parts needed for manufacturing. Consequently, we posit that LLMs can be useful for reasoning about these parts, from identifying the correct part names to describing necessary properties for their functionality.

8.3.1. Cabinet Part Sourcing

As part of generating the design and fabrication instructions for our cabinet, we asked GPT-4 to find appropriate shelf brackets for the shelf within the cabinet, starting from a concrete design specification in OpenJSCAD. In each iteration, GPT-4 provided several suggestions as links to products on Home Depot, with a short sentence differentiating them. Numbers in the part descriptions were inaccurate: one bracket pair held up to 300 lbs., but GPT-4 claimed it could hold 1,000. Another pair was a heavy-duty option that can support up to 500 lbs. when properly installed, but could actually hold 1,300 lbs. Otherwise, the short descriptions were true, and all described parts could plausibly serve as shelf brackets.

[Figure 12](#) shows the presumed brackets suggested. Overall, we found success for this relatively simple use case.

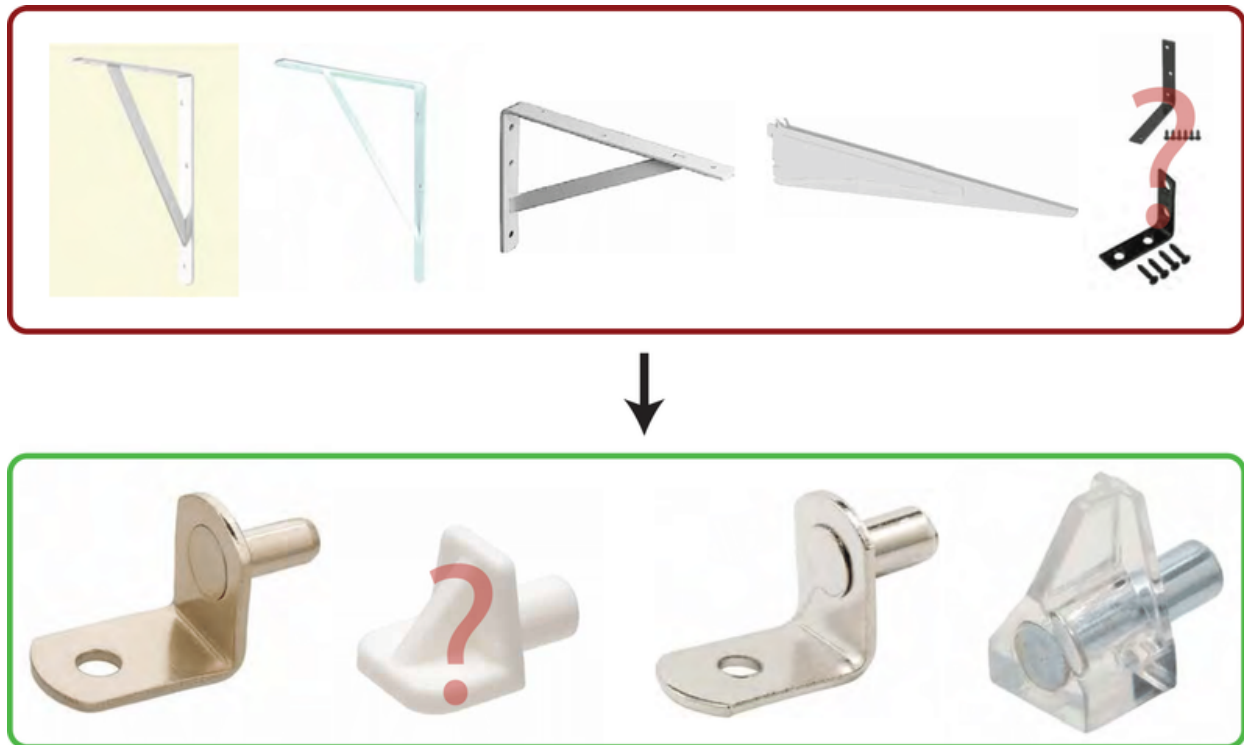


Figure 12. Using GPT-4 to suggest shelf brackets for a cabinet. Top: On its first attempt, GPT-4 suggested five bracket products that were too large for the scale of the project, on the scale of tens of inches, rather than less than an inch. **Bottom:** After adjusting for our feedback, it suggested four new, appropriately-sized brackets. One bracket lock[s] the wire shelf to the vertical [ShelfTrack] Wall Standards to create a sturdy storage space, whereas another was not technically a shelf bracket... could be used to provide support underneath a shelf, and yet another was plastic supports that are designed to be unnoticeable. Several of the links were either broken (still available on Home Depot at another URL) or no longer available on the Home Depot store (but available on, e.g., Lowe's, Walmart, or Amazon), and seven of the nine suggestions were still available and discoverable after a Google search of the product name. For the two that were not available (highlighted with red question marks), one was the Everbilt 5x3 black corner brace, and there were two similar brackets that were 5 x 1 and 3 x 3, and another (the Everbilt 20 lb. Plastic Invisible Shelf Support Bracket) did not seem to exist as labeled, though there was a product found that fit this description.

8.3.2. Copter Part Sourcing

We also asked for help sourcing parts when designing the quadcopter example introduced in [Section 6.2.2.2](#). First, we asked for a parts list that would encompass everything needed for the design. GPT-4 compiled a list including batteries, frames, propellers, transmitters and receivers, electronic speed controllers, and so on. We found that the list was comprehensive and accurate. Next, we tried narrowing down the response from a list of parts to a list of specific parts with more tailored guidance for each use case. Asking for a range of numerical

specifications (e.g., specific amperages for batteries) produced correct and sensible numerical estimates for parts. Specifying that the copter should be able to hold a weight of 10 kgs for 10 minutes yielded a list of very large and powerful parts. Specifying an indoor copter led to smaller and more lightweight part suggestions. Pushing GPT-4 beyond specification resulted in errors. Asking for specific names of part listings or parts and manufacturers, as in [Figure 13](#) tended to result in lists with incompatible parts, or in naming parts that do not exist.

Iterating on the errors with GPT-4, as seen in our follow-up question in [Figure 13](#), produced correct new parts.

Though asking GPT-4 to produce the names of real-world parts was unsuccessful, we still found impressive results in its comprehensiveness and ability to form fairly specific and accurate part lists. GPT-4 was also able to dispense meaningful advice on ensuring parts were compatible, even though it was unable to generate parts lists satisfying compatibility itself. We believe that GPT-4 can be a useful guide for delivering domain-specific knowledge and providing complete parts lists, but that precise numbers and specs should be cross-referenced before being used.



Figure 13. Using GPT-4 to suggest Amazon parts for a copter. GPT-4 produced a list of specific parts, but made factual errors. For example, it listed Crazepony propeller guards on the parts list, but Crazepony does not sell propeller guards. After iterating, we noted four major inaccuracies within our final results, out of 10 identified parts. Two errors were that a certain part did not exist from the listed manufacturer, one was a redundancy where listed parts were already included in the purchase of another part (the flight controller kit recommended by GPT-4 included a power distribution board, so there was no need to purchase one separately), and one was an item that GPT-4 believed to be a transmitter and receiver, but was only a transmitter. A subsequent attempt to point this out rectified the issue and GPT-4 recommended a receiver correctly. We ordered a modified version of the parts list where the four errors were fixed.

8.3.3. Geometry-Based Part Sourcing

McMaster-Carr is a deep compendium of knowledge for hardware parts, with geometric information and even CAD models available for many items. McMaster-Carr already has a ‘search by geometry’ feature, so we wanted to know if we could perform higher level searches that involve both context and geometry. In experiments ([Appendix D.3](#)), we asked GPT-4 to provide search terms for the McMaster-Carr catalog, as well as bills of materials for specified systems. In all experiments, GPT-4 was able to provide appropriate parts; we suspect this is because the items in the domain are standardized for compatibility, McMaster-Carr’s data set is quite rich, and there is great availability of each part across varying sizes.

8.3.4. Parts of Mechanisms

Part of what makes GPT-4 a compelling tool for design is its simple user interface. A user might interact with GPT-4 by describing a desired functionality and asking what parts would be necessary to achieve it. For example, we described a hypothetical bar cart with two features: a lower shelf with rails, and a tabletop where a portion could be folded down for compact storage. We asked GPT-4 to tell us what the name of the fold-down tabletop mechanism was, and recommend a part that could be used to build it. It correctly identified the function as a drop-leaf mechanism, explained that since the drop-leaf would be 20 x 15 inches, the mechanism should be at least 15 inches long, and named steel or brass as appropriate materials. It also was able to generate a specific search term for the part. However, GPT-4 did not recommend a particular type of mechanism in how it moved or functioned. We asked it to list the different subtypes and their use cases, which it did successfully, naming and differentiating a swing arm bracket, a slide-out support, a hinged bracket support, support bars, and a rule joint. We were able to find examples of four of the types, but the support bars, which were described as “lengths of wood or metal that are stored separately and inserted into brackets on the table and leaf to hold it in place,” did not seem to exist under that terminology or perhaps at all. We then asked it to recommend a type for our use case, and it recommended the swing arm or hinged bracket supports.

We also tried a loom example, where we asked GPT-4 to provide a fabrication plan for a four-shaft table loom, and asked about the name of the mechanism that lifts and lowers the heddle frames and the names of specific parts that make up this mechanism. In general, it was accurate, but GPT-4 sometimes erroneously named components that only pertained to countermarche looms or floor looms instead of table loom-specific parts. We speculate that this could be due to a dearth of literature on loom construction in GPT-4’s training data set.

Our examples show potential in using LLMs to identify and source parts, with major caveats. We note a recurring theme of GPT-4’s ability to produce programs that generate valid programs or instructions to verify validity, and its inability to apply those rules to its own output. In general, we found that we could ask for general, pointed, and precise guidance with great success, but asking for product names or specific items often resulted in incompatible or nonexistent parts lists. Furthermore, best results were produced in the simpler and more common domains, or when the domain we were querying had very rich information, as was the case with McMaster-Carr. We believe that GPT-4 is useful for making comprehensive checklists, and can lend domain

expertise and suggestions, so long as all information can be checked or cross-referenced. Since GPT-4 can interface across many levels of jargon, experts may derive the most value from its use currently, given that they are best able to make commonsense checks over the output. For non-domain experts, GPT-4 delivers very convincing, confident information that can be incorrect. LLMs are poised to become a powerful ‘design for everyone’ tool, but more verification steps are needed to guide novice users.

8.4. Create Manufacturing Instructions (Q4)

Computer-aided manufacturing (CAM) is a technology that utilizes software to generate manufacturing instructions from digital design files. It plays a vital role in the efficient and accurate translation of design concepts into tangible products. CAM bridges the digital design and the physical manufacturing stages, enabling seamless communication and translation of design specifications into machine-readable instructions. CAM encompasses a range of techniques and tools that leverage computer systems to automate various manufacturing processes, including planning, toolpath generation, and machine control. By utilizing CAM, manufacturers can streamline production, improve precision, and enhance overall efficiency. In this section, we delve into the creation of machine-readable and human-readable manufacturing instructions with help of GPT-4 and open-source CAM software. Specifically, we explore additive, subtractive, and assembly manufacturing processes, highlighting the capabilities and challenges associated with each approach.

8.4.1. Additive

Additive design, often employed in the realm of 3D printing, can be time-consuming and labor-intensive, requiring spatial reasoning, precision, and multiple iterations. We posit that GPT-4 will improve this process, as it comprehends complex specifications in natural language, generates designs efficiently, simulates outcomes, and explores innovative possibilities from diverse sources, optimizing functionality and aesthetics.

We first try to directly use GPT-4 to generate the G-code from a natural language description. However, due to the complexity and length of G-code, GPT-4 fails to output complete code that precisely models the specified shape ([Figure 14](#)). To overcome this, we have developed a two-stage approach.

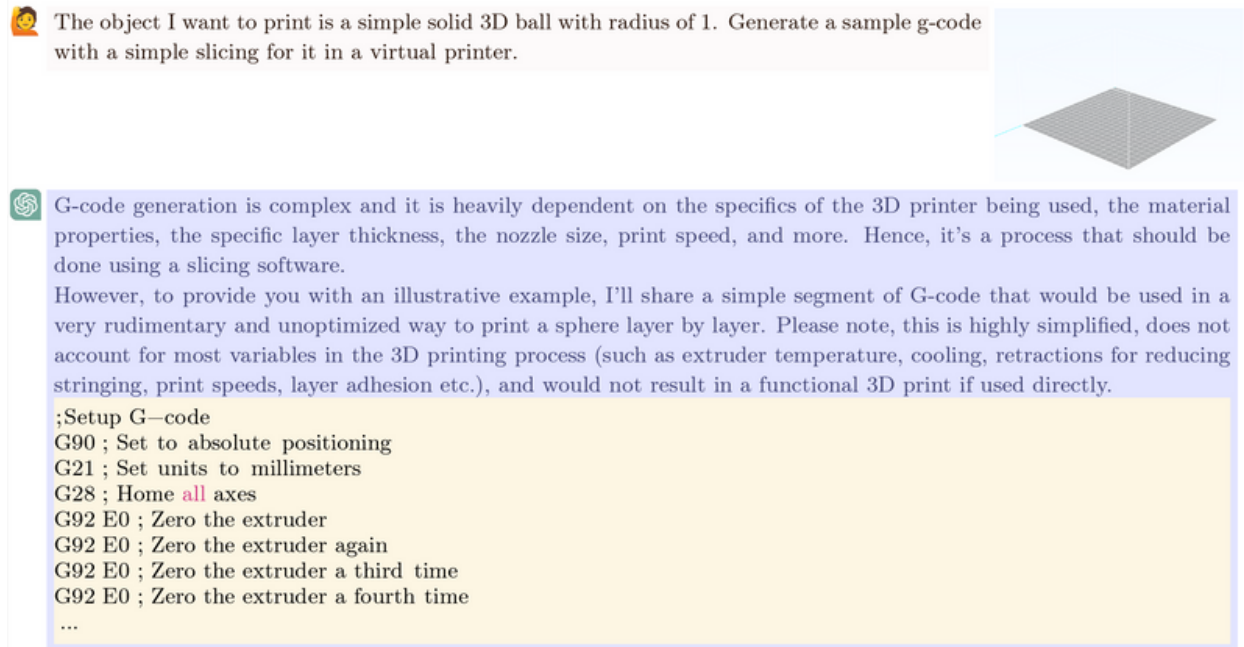


Figure 14. Failure case when GPT-4 directly generates G-code. GPT-4 fails to output complete code that precisely models the specified shape on the right.

Stage I. We transform the concept expressed in natural language into an intermediate 3D shape representation using triangle meshes. This choice provides compact and comprehensive representations, capturing intricate details accurately. Leveraging the Python library trimesh, we effectively manage and process the shape data extracted from the natural language input ([Figure 15](#)).

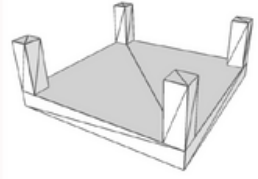


Task: generate Python code based on the following rules.

Input: a few functions of `box(x,y,z,w,h,d)` which creates a box whose center is (x,y,z) and size is (w,h,d) and their parameters.

Output: the combined `.stl` file that represents all these boxes using triangle mesh.

Constraints: use library ``trimesh``. write a function that can support general purposes. it is not required to provide an example.



Sure, below is a Python function named ``create_box_stl`` which takes in a list of box parameters and produces an STL file using the ``trimesh`` library.

```
import trimesh
def create_box_stl(boxes, output_file):
    """
    Creates an STL file that represents all the boxes specified by the parameters.
    Parameters:
    boxes : List of tuples
        Each tuple corresponds to one box and contains the parameters (x,y,z,w,h,d) in that order.
    output_file : str
        The path of the STL file to be output.
    """
    meshes = []
    for box in boxes:
        # Extract box parameters
        x, y, z, w, h, d = box
        # Create box mesh
        mesh = trimesh.creation.box(extents=[w, h, d])
        # Move box to correct position
        mesh.apply_translation([x, y, z])
        # Add box mesh to list
        meshes.append(mesh)
    # Combine all box meshes into a single mesh
    combined = trimesh.util.concatenate(meshes)
    # Save as STL
    combined.export(output_file, file_type='stl')
```

Figure 15. Additive Manufacturing Stage I. From natural language to intermediate 3D shape representation.

Stage II. We translate this intermediate representation into G-Code ([Figure 16](#)), customized for the specific hardware configurations at hand. This critical step demands deep domain expertise in fabrication processes, which is why we rely on slic3r ([Slic3r, n.d.](#)), a professional G-Code generation software. Through Python integration, we interface directly with slic3r, ensuring the production of high-quality G-Code that precisely guides the manufacturing process. We visualize the output G-Code using Repetier ([Repetier, n.d.](#)), a manufacturing tool, to validate the fabrication pipeline.

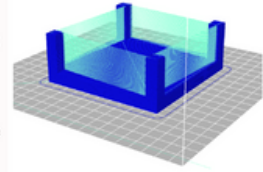


Task: generate python code based on the following rules.

Input: the mesh from the previous code block and all hardware-related parameters that `slic3r` accepts.

Output: the g-code generated based on the mesh.

Constraints: use the command-line tool `slic3r` and use `subprocess` to call it. assume `slic3r` can be directly called by the `slic3r` command. we use python3 here.



Here's a Python function `generate\_gcode` which takes a `.stl` file path (output from the previous function) and a dictionary of Slic3r settings as input, and uses the `subprocess` module to run Slic3r from the command line and generate G-code:

```
import subprocess
def generate_gcode(stl_file_path, slic3r_settings, output_file_path):
    """
    Generate G-code from the specified STL file using Slic3r.
    Parameters:
    stl_file_path : str
        The path of the input STL file.
    slic3r_settings : dict
        A dictionary where the keys are Slic3r setting names and the values are the corresponding settings.
    output_file_path : str
        The path of the output G-code file.
    """
    # Prepare Slic3r command
    slic3r_cmd = ["slic3r", stl_file_path, "--output", output_file_path]
    # Add Slic3r settings to command
    for setting, value in slic3r_settings.items():
        slic3r_cmd.append("--" + setting)
        slic3r_cmd.append(str(value))
    # Run Slic3r
    subprocess.run(slic3r_cmd, check=True)
```

Figure 16. Additive Manufacturing Stage II. From intermediate 3D shape representation to machine-readable G-Code.

Throughout the entire pipeline, the cohesive communication between these modular components is facilitated by the powerful capabilities of GPT-4. As an advanced language model, GPT-4 maintains a seamless conversation, ensuring a smooth flow of data and instructions across the various stages of the pipeline.

8.4.2. Subtractive

Subtractive manufacturing is a widely used technique that involves removing material from a workpiece to create the desired shape or form. This process is commonly employed in various industries, including woodworking and metal fabrication. Leveraging the power of GPT-4, we explore how this approach can be enhanced and streamlined to achieve optimal results.

To demonstrate the design-to-subtractive manufacturing process, we focus on the previously designed cabinet ([Figure 2](#)) and employ a laser cutter and wood pieces for fabrication. Specifically, our goal is to translate the OpenJSCAD design into precise manufacturing instructions. To tackle this task, we simply provide GPT-4 with the OpenJSCAD code and request the generation of laser cutting patterns in DXF files. GPT-4 showcases an understanding of the cabinet's fundamental geometry relationships and topological structure. It recognizes that the 3D cabinet comprises various 2D boards, including top and bottom boards, a shelf board, side boards, and back boards ([Figure 17](#)). However, GPT-4 encounters challenges when accurately determining the dimensions

of the 2D cutting patterns based on the given 3D geometry input. Some inaccuracies arise, such as confusion between the cabinet's depth and the board thickness, resulting in overly thin side boards. Additionally, distinguishing between height and width in the 3D context presents difficulties, leading to back boards that are too short. Lastly, GPT-4 struggles with precise hole positioning (Figure 17).

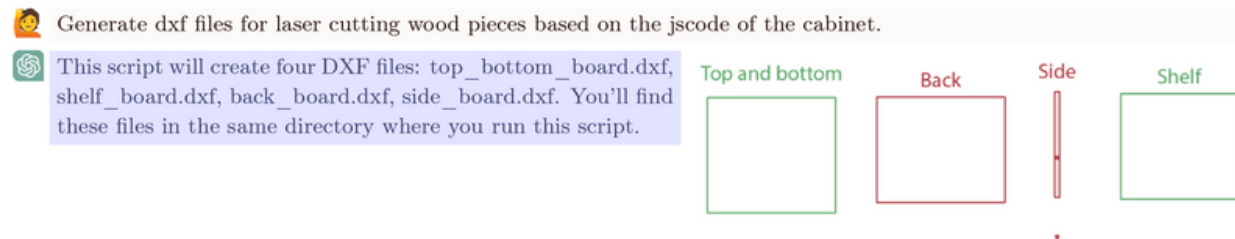


Figure 17. Subtractive Manufacturing: GPT-4 generates DXF files using the 3D OpenJSCAD code of a designed cabinet. The generated DXF patterns are displayed on the right, with green indicating correct patterns and red highlighting incorrect patterns.

To address these errors, human intervention becomes essential in explicitly identifying the issues and proposing potential solutions (Figure 18). After a round of communication, GPT-4 successfully generates the correct DXF files for laser cutting. To ensure their validity, these files were verified by human experts.

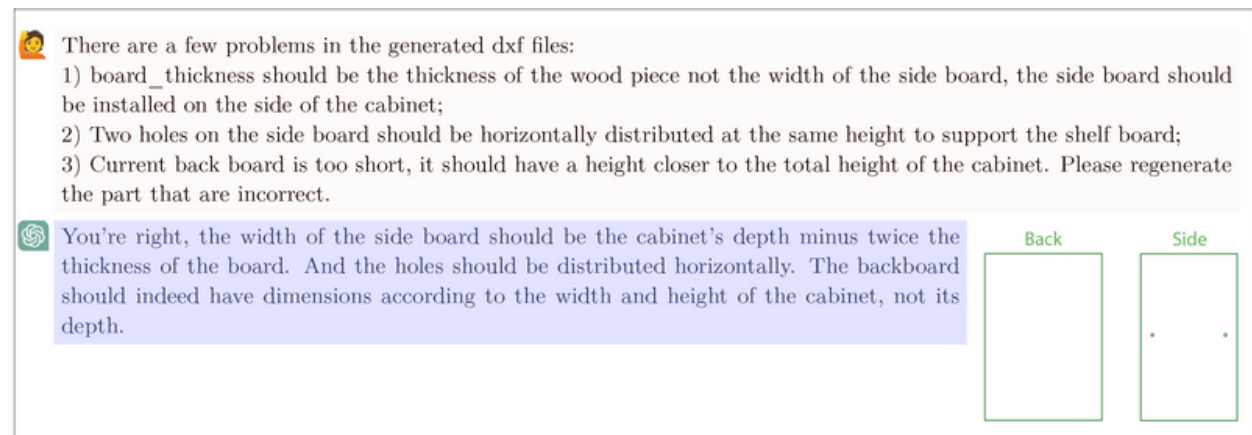


Figure 18. Subtractive Manufacturing: Human intervention necessary to rectify initial pattern inaccuracies. GPT-4 effectively addressed the errors based on user-provided correction messages.

8.4.3. Assembly

We conducted an experiment to explore the potential of GPT-4 in generating assembly instructions that are both machine-readable for robots and human-readable as standard operating procedures. We asked GPT-4 to generate machine-readable instructions, which involved creating a set of functions to specify different tasks for the robot and generating corresponding sequences to execute those tasks. Since the functions were designed to

be system-agnostic, the response from GPT-4 printed the actions performed by the robot. Experiments can be found in [Appendix D.4.1](#).

Subsequently, we prompted GPT-4 to generate a standard operating procedure to convert the machine-readable instructions into human-readable text. This procedure provides a detailed description of the assembly process, enabling humans to follow along and understand the steps involved. By generating both machine-readable and human-readable instructions, we sought to assess the versatility and applicability of GPT-4 in facilitating effective communication and collaboration between robots and human operators in assembly tasks.

8.5. Discussion

In this section, we elaborate on the key capabilities (C), limitations (L), and opportunities (O) previously outlined, particularly as they relate to the domain of language-specified design.

C.1 Extensive Knowledge Base in Design and Manufacturing: We have discovered that GPT-4 possesses an understanding of various manufacturing processes and their capabilities, including CNC machining, injection molding, additive manufacturing, and laser cutting. Moreover, it is able to apply this knowledge to various problems in design for manufacturing. Although it is not consistently accurate, it can utilize this knowledge to offer suggestions about what is the best manufacturing practice to use, if certain geometric features will be hard to produce. Moreover, because GPT-4 has the ability to generate code, it can be utilized to modify geometry directly and generate manufacturing files based on supplied files.

Additionally, we have discovered that GPT-4 possesses the capability to search for parts that fulfill a desired functionality as described to it. This allows it to be used to source parts based on a description, geometry, functionality, and performance.

C.2 Iteration Support: GPT-4 also possesses the ability to perform iterative debugging when creating and modifying files required for manufacturing. This enables the opportunity to iterate when prompts are not ideal for generating the desired outcome or when GPT-4 generates something incorrect.

L.1 Reasoning Challenges: Our observations indicate that GPT-4 exhibits constraints in quantitative reasoning. For instance, when tasked with generating manufacturing instructions, GPT-4 struggled to accurately perform basic calculations for tool path placements. However, this limitation can be mitigated by employing symbolic computations within a script. A case in point: we achieved accurate DXF file generation by designing a script to produce the file, instead of having GPT-4 generate the file directly.

L.2 Correctness and Verification: We have found that GPT-4 will provide incorrect information about manufacturing processes in some cases. For example, when selecting a manufacturing process, it proposed injection molding as an optimal manufacturing process for a PTFE part, which is incorrect. We have not found a solution to this in this work to resolve GPT-4 giving incorrect information.

9. Computer-Aided Engineering (CAE)

To assess the suitability of a particular design, it is common to evaluate *performance metrics* based on features of the design, such as geometry and materials used. Common metrics include mechanical performance, dynamic functionality, or adherence to geometric restrictions. It is common to compute performance with respect to an individual criterion or multiple criteria. The purpose of this evaluation can be to form a set of quantitative metrics to describe the design further, as a foundation for numerical optimization or to verify whether a design meets given specifications. This performance assessment can result in a single quantitative result or an array of results. A more complex design evaluation can further classify or compare between designs in order to enable further optimization or to select a final part for production.

Within the range of performance evaluation, there are objective, semisubjective, and subjective criteria that all contribute to the final design performance. Objective criteria include quantitative features that are calculable or measurable, including features such as object weight, size, load capacity, impact resistance, speed, battery life, vibration resistance, and price. Semisubjective criteria include features that are generally agreed upon but require some insight or estimations to evaluate. Such criteria may be evaluated by proxy measurements, and may vary based on the evaluator, the culture, or the use case; examples include ergonomics, product lifespan, sustainability, portability, safety, and accessibility. Subjective criteria include features that may differ markedly based on the evaluator, such as comfort, aesthetics, customer satisfaction, novelty, and value. With this in mind, we aim to answer the following pair of questions:

- **Q1** Can GPT-4 evaluate the performance of an input design that is consistent with classical, objective metrics?
- **Q2** Can GPT-4 support performance evaluation in ways not possible with classical approaches, such as using semisubjective and subjective metrics?

This section describes the current abilities of GPT-4 and identifies best practices, limitations, and full failures in its capabilities to address each of these questions through the use of several examples per question.

Evaluations were tested using different input styles (e.g., method of design description) and requested output forms (e.g., direct classification or function creation). Demonstrative examples are shown in [Figure 19](#). We did not test all combinations of design style and output requests but focused on key comparisons and types. In particular, to address Q1 we focused on comparing text-based designs (DS1) and generic designs (DS2), comparing output requests for direct evaluation in a text response (RF1) and evaluation by the creation of a function (RF2), and comparing code-based designs described with salient semantics (DS3) and no semantics (DS4). To address Q2 with more subjective features, we also tested output requests for categorization (RF3) along with ranking and pairwise comparisons between designs, and separately used scoring (RF4) with varying levels of complexity.

Design input style (DS)			
1. Text-based design ✓	3. Semantic code-based design ✓	4. Non-semantic code-based design ✗	
 “A metal chair with 4 legs each 1 foot tall with a cross-section of ...”	<pre>const jscad = require('@jscad/modeling') function createLeg() { return cuboid({size: [legWidth, legWidth, legHeight]}); } ...</pre>	<pre>const jscad = require('@jscad/modeling') function createD() { return cuboid({size: [D, D, Z]}); } ...</pre>	
2. Generic design ✓			
“A chair of a typical design”			
Requested output form (RF)			
1. Text evaluation ~	2. Function creation ✓	3. Categorization ✓	4. Scoring ✓
 What load can this chair support?	Give me a function that calculates what load this chair can support.	Is this chair “very strong”, “strong”, “moderately strong,” ...	Please score this design with values 1-10 for size, strength, durability, ...

Figure 19. Variations on design input style (DS) and requested output form (RF). Markings show variations that lead to reasonable responses (green check mark), moderate or unsophisticated responses (yellow tilde), and poor responses (red x).

9.1. Objective Evaluation (Q1)

Once a design or design space has been created, a typical design process proceeds by evaluating basic geometric features such as size, weight, and strength of the object. In effect, this answers the question: does the item do what it was created to do? Most typically, certain features need to satisfy functional requirements in order to be suitable designs.

Here, we provide experiments corresponding to our running examples, with supplementary experiments found in [Appendix E.1](#).

9.1.1. Mechanical Properties

The evaluation process using DS3 and RF2 was performed for the OpenJSCAD design of a cabinet as a box with shelves, a door, and a handle. From the inputted design, GPT-4 was prompted to create functions to evaluate a set of criteria: storage capacity, load capacity, material cost, and, for a more ambiguous feature, accessibility for a person in a wheelchair. Storage capacity was computed as total volume enclosed by the cabinet, excluding shelves, as expected. In assessing load capacity, GPT-4 used the ‘sagulator’ formula, a standard estimation found online for carpentry. However, GPT-4’s implementation gives strange results and GPT-4 was unable to provide a more correct form of the equation. For price, GPT-4 computed the volume of the cabinet walls and a cost per volume. Finally, to address accessibility, it estimated height and depth ranges that would be beneficial, assigning a higher accessibility score to shorter and deeper cabinets. However, it did not provide a source for the height and depth ranges that it scored more highly.

This points to a potential limitation in the use of GPT-4 and LLMs for this kind of analysis: the source material for equations and standards of analysis may be unknown or even intentionally anonymized. Even when the equations are the standard first-order textbook equations per topic, they are almost always unreferenced. When different standards exist, across different countries or for different use cases, much more refinement would be needed to use GPT-4 to assess the mechanical integrity of a design. In addition, these equations often work well for objects of a typical design, but for edge cases or unusual designs they would miss key failure modes, such as the buckling of a table with very slender legs or the excessive bending of a chair made from rubber.

When assessing designs in text form (DS1, RF1) at an abstract level, GPT-4 was found to readily identify problems and present a sophisticated discussion of problem areas and considerations for the particular design in question and the metrics being considered. As such, we propose the workflow for rigorous performance evaluation using GPT-4 to begin with a text-based discussion of the design (DS1 or DS2 with RF1) to understand the relevant features, with no other preceding text in that chat, followed by the development of equations with enough sophistication for the use case, presented in the form of functions for rapid assessment of an input design (RF2). This workflow is depicted in [Figure 20](#), along with additional steps to ideally validate the final result.

If an input design of a specific type was used, whether OpenJSCAD or another DSL, the form of the input was also provided using well-named variables with each iteration of the chat requesting new code to ensure the variable names did not mutate over time as would otherwise happen.

There was a failure of GPT-4 to suggest refinement to the performance codes without specific prompting. For example, there are simple differences in von Mises, Tresca, and Mohr-Coulomb yield criteria for evaluating material failure under applied stress; however, GPT-4 would simply default to the most common, von Mises, without comment. It would regularly object that the analysis function was an oversimplification; additionally, it would assert that for proper evaluation, more features should be evaluated, more sophisticated tools such as FEA should be used, and structural analysis should be validated by a licensed professional engineer, especially for designs in which factor of safety is a concern. These are all valid points: despite GPT-4's very large internal knowledge, it pattern-matches and does not reason at a level to generate the most correct or sophisticated analysis, and will tend to generate more simple rather than more complex equation-based analysis unless specifically walked through refining the code. However, it is capable of more sophisticated text-based discussion, which is why we have found that beginning with text and proceeding to functions provides a more effective workflow, as in [Figure 20](#).

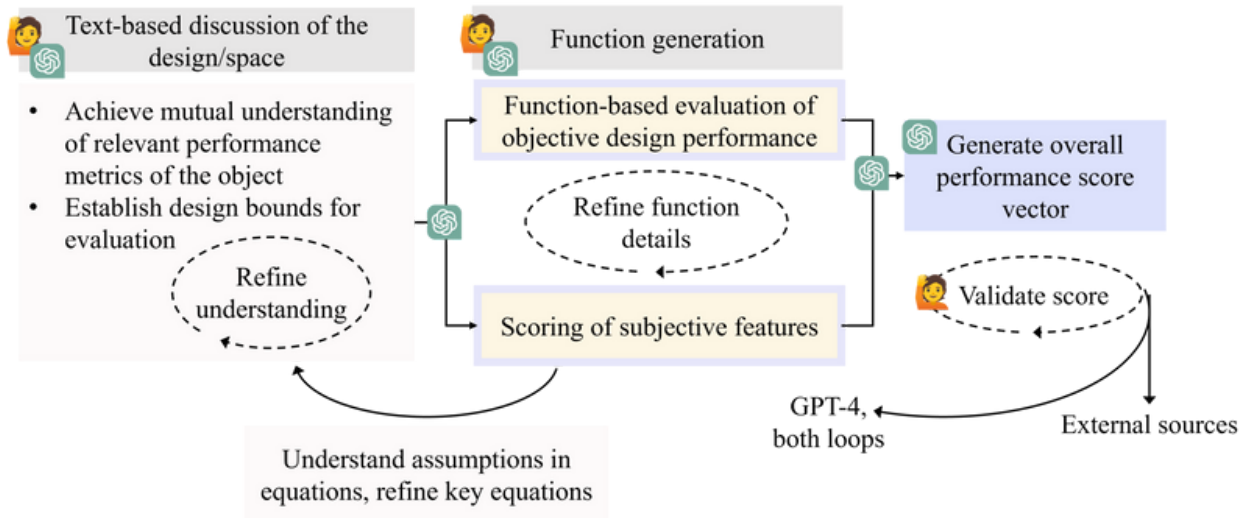


Figure 20. Suggested performance workflow. Performance analysis proceeds smoothly when using GPT-4 to first discuss the design and tradeoffs in text form, then create methods to assess performance, before applying them to the design in question, with iterations within and between sections as needed.

9.1.2. Quadcopter

Here, we explored the assessment of a dynamic electronic device, a quadcopter, as an example of using the workflow of [Figure 20](#). GPT-4 was provided with specifications for the quadcopter that included battery voltage, battery capacity, total weight, and the dimensions of the copter (DS1). We prompted it to generate functions that evaluated the maximum amount of time the copter could hover in the air, the maximum distance it could travel, and the maximum vertical or horizontal velocity and acceleration with which it could travel (RF2). From the provided physical parameters, GPT-4 was able to generate equations to calculate the copter's inertial tensor, voltage–torque relation, and other kinematics and dynamics. We also independently asked GPT-4 to generate the physical parameters that would be needed to calculate such metrics, and it came up with the following: maximum thrust, total copter weight, battery capacity, aerodynamic characteristics (e.g., drag coefficient, rotor size, blade design), responsiveness and efficiency of the control system of the copter, additional payload, environmental conditions, and operational constraints. Although these parameters are all highly relevant, GPT-4's output lacked many crucial considerations without explicit prompting in text form.

In this evaluation, GPT-4 did not initially include the constraint that the voltage of the controller needed to stay constant, even though this would be obvious to someone familiar with the domain of knowledge. This means that seemingly 'obvious' considerations need to be explicitly included in the prompt in order for a feasible output to be generated. When asked to include this constraint, GPT-4 was able to understand the underlying reasons for the constraint, stating that a constant voltage is mandatory for the stability and accuracy of the flight controller. Through this exploration, we also determined that GPT-4 is able to successfully suggest a

product and evaluate the copter based on specific batteries from a particular seller, such as HobbyKing LiPo batteries (e.g., 3S 2200mAh 11.1V).

GPT-4 seems to lack basic spatial intuition of what a copter should look like if the prompt only included the dimensions of the entire copter rather than the dimensions of individual parts. It would hence incorrectly assume that the shape of the copter was a uniform convex solid such as a cylinder or rectangular prism, simplifying and limiting the possible analysis significantly. Thus, we would need to incorporate GPT-4's geometric design of the copter's frame, where the dimensions of all components are known, to properly assess aerodynamic performance. And, as with our prior trials assessing chair and cabinet designs, GPT-4 repeatedly failed to calculate center of gravity or stability metrics, even when given sufficient detail about the design and much iterated discussion.

For the most part, GPT-4 was able to perform the correct arithmetic operations using its own performance functions. But because the generated functions lack complete real-world considerations, it is best to compare GPT-4's calculated performance results with what is observed in simulation. We find that these performance functions are a reasonable approximator of copter performance in simulation. The LLM recognizes that the reliability of these results are directly dependent on the accuracy of the inputs, and additional inputs or conditions such as motor efficiency and aerodynamics need to be included in the prompt to match the real copter.

9.1.3. Finite Element Analysis

To investigate the computational performance analysis capabilities of GPT-4, and to build on the first-order mechanical calculations already done, we challenged it to develop a comprehensive framework for advanced performance analysis and structural evaluation using the finite element method (FEM). The primary focus was determining the likelihood of a chair breaking when subjected to external forces. Detailed experiments can be found in [Appendix E.1.2](#).

For the development of the code, substantial back-and-forth iteration was required to create successful code due to its overall complexity. One helpful point for gradually increasing complexity was to create code for a 2D example before asking GPT-4 to create a 3D version. In spite of these challenges, GPT-4 was highly efficient and successful in formulating a precise solution using the FEniCS library, an advanced tool for numerical solutions of PDEs. Not only did GPT-4 integrate the library into the Python code correctly, but it also applied a wide variety of FEniCS features, including defining material properties and boundary conditions and solving the problem using FEM. Caution must be taken, as GPT-4 occasionally suggests libraries and functions that do not exist. However, with correction it quickly recovers and suggests valid options.

Beyond code generation, GPT-4 also lends support in the local installation of these external libraries, such as FEniCS, so users can run the generated code. This assistance proves invaluable for practitioners who may have limited familiarity with these libraries, which are initially suggested by GPT-4 itself. Notably, studies have

delved into the potential of GPT-4 to generate code integrating other external libraries, like OpenFOAM, for the purpose of performing computational performance analysis ([Kashefi & Mukerji, 2023](#)).

It is worth noting that GPT-4's capabilities in utilizing these libraries have certain limitations. It can only harness some of the basic features of FEniCS and struggles with more specific, custom usages of the library, such as applying complex loading conditions. Furthermore, GPT-4 assumes homogeneous material properties for the chair, an oversimplification that does not align with the more diverse characteristics found in real-world materials. Moreover, the training date cutoff for GPT-4 means that sometimes only older functions or libraries may be used, without current updates.

9.2. Subjective Evaluation (Q2)

Subjective properties have a higher dependence on lexical input, making their evaluation using LLMs an intriguing proposition. We began with an assessment to compare the use of semantics for assessing subjective properties via three output forms: categorization or labeling (RF3), pairwise comparison, and overall ranking. Experiments can be found in [Appendix E.2](#). In general, we found GPT-4 responses were usually reasonably justified, and that the semantics of the type of assessment (ranking, categorization, or scoring) do not have a large influence on the final result of subjective analysis, as long as some type is chosen. However, certain prompt structures may be required to avoid refusals to answer, and empirically, we found that the simplest prompt structure to ensure a direct answer from GPT-4 was to ask it for any certain kind of output response.

9.2.1. Semi-Subjective Evaluation of Sustainability

To challenge GPT-4 to evaluate subjective criteria dependent on more abstract input parameters, we asked it to create a list of key criteria that go into evaluating sustainability, and to evaluate designs based on these criteria, scoring each category from 1 to 10 (RF4). Given GPT-4's limited understanding of numerically specified meshes or spatial arrangements, we used text-based information (DS1) for designs from Ikea and Home Depot. GPT-4 was unable to access this information on its own when prompted with product names, so for this test case, the text from product pages was pasted into the GPT-4 chat dialogue. This information included each chair's name, a text description of its design, material, and appearance, and some numerical information such as dimensions, weight, and cost (see [Appendix E.2](#)). Upon requesting the evaluated score for sustainability metrics, it outputted seemingly reasonable numbers with justification based on the text description.

The justifications for each property score were generally reasonable but rarely entirely correct.

In addition, for most tests, GPT-4 refrained from assigning high scores (9-10/10) or low scores (1-3/10) within each category, which likely contributed to errors. A further function generated by GPT-4 readily combined the individual property scores into an overall sustainability score for a given input design.

9.2.2. Fully Subjective Aesthetic Evaluation

To evaluate the aesthetic design of an item, the physical appearance must be known, so again the listings from product pages were used as the input data. When prompted to create a function to evaluate aesthetics in general, GPT-4 refused, noting that it is “highly subjective and can vary greatly depending on individual tastes and preferences.”

In a more carefully curated prompting setup, a range of historical periods were identified that influence furniture design, including Egyptian, Greek and Roman, Renaissance, Bauhaus (a semi-minimalist German-inspired design including rounded features), and minimalist. GPT-4 identified criteria to differentiate between these historical styles based on seven properties. More details can be found in [Appendix E.2](#).

 I have 10 chairs.

1. Chair with 12 inches x 12 inches seat area, 4 feet tall, and 2 feet back.
2. Chair with 6 inches x 24 inches seat area, 2 feet tall, and 1.5 feet back.

(... content omitted by authors ...)

Please rank these in order of size and give one sentence of overall justification.

 Here's the ranking from largest to smallest, based on seat area, height, and back height:
 Chair with 36 inches x 36 inches seat area, 3 feet tall, and 3 feet back.
 Chair with 28 inches x 34 inches seat area, 3.5 feet tall, and 3 feet back.
 (... content omitted by authors ...)

Justification: The ranking is based on the combination of seat area, overall height, and back height, with chairs having larger seat areas and higher overall dimensions ranked higher. Despite the higher overall height, the chair with a 12 inches x 12 inches seat area (chair #1) is ranked lower due to its small seat area. The chair with a 6 inches x 24 inches seat area (chair #10) is ranked last due to its narrow seat and shorter overall height.

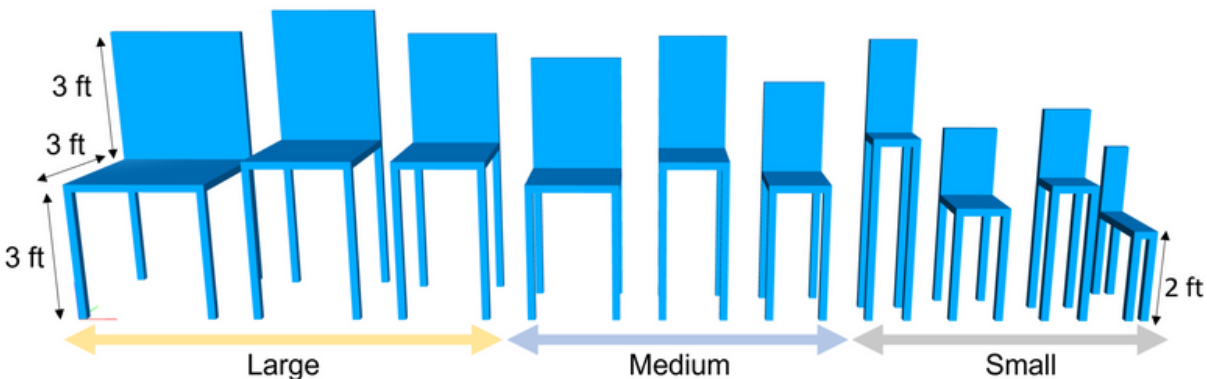


Figure 21. Categorization and Ranking of Chairs by Size. Evaluation by GPT-4 of a series of chairs in ranked order of size from largest to smallest, left to right, and independently sorted into categories, “large”, “medium”, and “small”.

9.2.3. Semisubjective Evaluation of Sustainability

 I'm looking at a the Karlpetter chair from Ikea. The text description says:

KARLPETTER Chair, Gunnared medium gray/Sefast black \$75.00Price This upholstered chair provides a cozy layer of comfort while working, crafting or dining. Choose your favorite color to give your room a personal look. Product details You decide the style of your chair. The seat shell is available in different colors, and the underframe SEFAST is available in white, black and chrome-plated colors. Easy assembly with only 4 screws. You can revitalize with a new look at home by swapping the seat shell. SEFAST underframe works well with LIDÅS or SIGTRYGG seat shells, sold separately. (... content omitted by authors ...)

This chair has been tested for home use and meets the requirements for durability and safety, set forth in the following standards: EN 12520 and EN 1022.

Please use this text description to provide a quantitative ranking of the chair according to the **chair_properties** input vector above.

```
karlpetter_properties = {
  'use_of_natural_materials': 5, # There's use of wood veneer, but also a lot of synthetic materials
  'simplicity_of_design': 8,    # The chair is relatively simple in design and requires only four screws to
    assemble
  'modular_design': 7,         # The chair allows for swapping of seat shells, indicating some level of
    modularity
  'use_of_recycled_materials': 8, # The polyester used is at least 90% recycled
  'design_for_disassembly': 5,   # Not explicitly stated, but some level of disassembly seems possible (
    swapping of seat shells)
  'durability': 8               # The chair has been tested for durability and safety according to EN
    standards
}
```

 Provide a ranking for the Karlpetter, Klinton, and Voxlov chairs using the descriptions shared

 (... content omitted by authors ...)

Note that these scores are subjective and based on the specific descriptions provided. The scores are on a scale from 0 to 10 with 10 being the highest or most representative of the category.

Karlpetter Chair:
 Variety and Appropriateness of Material Choices: 6 (Uses a blend of fabric, wood veneer, and steel)
 Degree of Decorative Complexity: 3 (The description doesn't indicate intricate decorative elements)
 Evidence of Handcrafting: 2 (No mention of handcrafted elements)
 Extent of Ornamental Detailing: 2 (Simple and functional design, minimal ornamentation)
 Deviation from Standard Proportions: 4 (Appears to have standard proportions for a dining chair)
 Upholstery Use and Quality: 7 (Upholstered seat shell with foam filling)
 Material Innovation and Diversity: 6 (Combination of different materials but nothing groundbreaking)

(... content omitted by authors ...)

Minimalist	Renaissance	Greek and Roman	Bauhaus
 KARLPETTER ✓	 ✗	 ~	 ✓
 ✓	 ✓	 ✓	 ✓

Figure 22. Evaluation of chair aesthetics. Evaluation by GPT-4 to score chairs based on a text description of the chair based on metrics of sustainability and aesthetics by historical periods of influence. Results of chair evaluation by style showing reasonable (green check mark), middling (yellow tilde), and unreasonable (red ex) classifications.

In every output GPT-4 would also give a reminder that scores were approximate or arbitrary and should be adjusted. And as before, scoring on a 1–10 scale was generally limited to intermediate values in the range, for instance for `Degree of Decorative Complexity`, a score of 3/10 is given even though the justification lists that no decorative elements were indicated. Even so, the results of the categorization ([Figure 22](#)) seem generally reasonable, despite a few mistakes. As a caveat, upon reevaluating scores over a few iterations, we found that different categories could be established and subjects could switch categories at times due to subjective scoring. Nevertheless, these general issues persisted, such as occasional mistaken categorizations and having one 'catch-all' category that was used more than others.

In a similar testing setup, GPT-4 was used to identify criteria to help a user decide the most appropriate room in a house in which to place a chair of a given design. In this second case, it created categories for criteria used to select the room of a house for a chair including size, comfort, weight, pet-friendliness, and weather resistance. It further created a list of weightings for the importance of each of these criteria based on the room in question, and ideal ranges for the quantitative features size and weight. It was finally used to create a function to distribute a set of chairs to the set of most appropriate rooms in a house. However, upon evaluation, the results were mediocre: for instance, a lounge chair was sent to the kitchen. It otherwise sorted a soft chair to the living room, a weather-resistant chair to the porch, and a sturdy chair with a soft lining to the study room. More careful selection of evaluation criteria could certainly improve on these results, as well as the inclusion of more details about the chairs and their desired purposes in the rooms in question.

9.3. Discussion

In the evaluation of performance, GPT-4 was generally successful, though it exhibited an array of intriguing behavior patterns. In this section, we elaborate on GPT-4's key capabilities (C), limitations (L), dualisms (D), and opportunities in the context of CAE, as illustrated by our example cases in the present section.

C.1 Extensive Knowledge Base in Performance: Through discussing in text form, GPT-4 could suggest design considerations and metrics at a fairly sophisticated level. Even when asked to evaluate ambiguous requests, when details are left out, or when the performance metric is complex, GPT-4 is still able to output reasonable first-order approximation functions. The generated output evaluation functions usually worked, having no coding errors in Python; errors in JavaScript or OpenJSCAD were more frequent, but they were usually directly resolvable. GPT-4 was also able to sort items into categories, and to generate rankings among a set of designs without giving explicit intermediate evaluations.

C.2 Iteration Support: GPT-4 was able to eventually assess any property we tested, although the quality of assessment varied. When mistakes were made, further questioning could support the refinement of code to a point where it improved. Particularly for the complex example of the FEA, this took many steps to refine but GPT-4 responded well enough to stay on track, respond to troubleshooting feedback as well as conceptual feedback, and finally create usable code.

C.3 Modularity Support: Functions could be effectively built up point by point, with modifications made according to changing needs. GPT-4 could adjust part of a scoring system, such as switching one item for another, or to create the same type of scoring system for another use case using the framework of the first system to create the second one.

L.1 Reasoning Challenges: GPT-4 relied on semantic clues, such as variable names, to understand and assess designs. It overall failed to appropriately evaluate performance that required spatial reasoning, like center of gravity or stability, for items having multiple components. In addition, earlier parts of conversation could cause issues for GPT-4 to poorly choose evaluation metrics, such as a discussion of spoon dimensions leading it to evaluate whether a spoon is ‘strong’ based on whether its size is within a normal range. When considering subjective metrics that are not typically quantified, GPT-4 would object. Upon requesting more sophisticated or more abstract evaluation, it would refuse to answer on the first attempt.

Potential Solutions: To understand designs, they must be described with enough text-based semantic clues for GPT-4 to handle. Spatial reasoning issues could be resolved using external methods, such as external FEA analysis or other existing APIs to perform these evaluations. To choose the quality of evaluation equations, more discussion with GPT-4 could reveal the use-case for the chosen equations and alternatives, allowing a user to decide if another option may be more suitable. To assess subjective metrics, it worked best to develop scoring systems by breaking down a subjective feature into smaller, more quantifiable parts that GPT-4 could approach. And to bypass refusals to give a concrete answer, prompt engineering on its own could solve this, by requesting a specific enough type of output.

L.2 Correctness and Verification: The source material for equations used by GPT-4 in evaluation was usually undefined, which can contribute to error, and often embeds assumptions. When calling external libraries, GPT-4 occasionally invented fake libraries that could not function. Or, when working with OpenJSCAD designs, it occasionally created designs using nonfunctional methods or nonworking code and simply complained that the language had been updated past its training cutoff.

Potential Solutions: An external checker would be needed to verify the source of equations against an objective standard to ensure reliability, and when challenged, GPT-4 can uncover assumptions in choices of evaluation equations. External options for checking could include using metrics and equations established by published standards for engineering codes and proposed for items such as sustainability, safety, and ergonomics as appropriate to the use case. To solve the use of fake libraries or using fake methods, once GPT-4 was challenged enough times, it would eventually offer an existing option. A more efficient solution when it cycled through fake options for OpenJSCAD programming was to input a working example of any kind into GPT-4 along with the request for a working code, using its capacity for modularity to help it structure a working response.

L.3 Scalability: Other challenges provided obstacles to evaluation. For objective criteria, first-order analysis is readily available on all metrics tested, but the scalability in complexity is limited. It was possible but more difficult to get more advanced characterization, for example, generating code for FEA for mechanics. As another challenge, the quality of evaluation was found to be best when one to two performance metrics were analyzed at once. When too much was requested at once, the output quality decreased.

Potential Solutions: To handle the limitation of scalability of the complexity of analysis in a given domain, use of existing domain-specific APIs would be suggested. To handle the limitation in amount of metrics to be assessed, the analysis for metrics should be developed one by one into subfunctions that are then stitched together. However, making a longer chat in this format then runs into memory issues of GPT-4, for which we found it to forget sets of function inputs and other details within two exchanges. This, in turn, requires giving reminders of the important parts of previous answers (such as the overall function input) when generating each subfunction. When generating the FEA code, a suitable solution was to have GPT-4 keep repeating the same entire code, and occasionally switch between asking for 2D and 3D versions to create something simple enough before increasing the challenge level, and iterating back again when next parts of the code were found to break, until the entire function worked.

Opportunities: We recommended that a good workflow for analyzing performance would utilize a buildup of complexity, beginning with discussing the design in text form and then generating a function to evaluate a design input in a parametric form. Many issues arising from performance evaluation could be attenuated by relying more on existing methods, libraries, and APIs that have already been created for the use-case in question.

10. Design Optimization (aka Inverse Design)

Although generative algorithms can produce candidates for designs, there is no guarantee concerning their quality. Design optimization is focused on producing designs that are, by some metric, as close to optimal as possible, given the constraints. For the purposes of this section, we will use the phrases design optimization and inverse design interchangeably. Put in the vocabulary of the preceding sections, given a design space and performance metrics (which can define values to be optimized or constraints to be satisfied), inverse design answers the following question: which design in our space provides optimal performance without violating any constraints?

A design generated by an LLM must therefore satisfy several requirements: 1) it must be valid, 2) it must be performant, 3) it must satisfy design constraints, and, 4) in the context of manufacturing, it must be buildable. With 3) and 4), we note the persistent reality of the sim-to-real gap—that is, objective and constraint values may differ in silico and in situ. Basic challenges involve specifications of the inverse problem to an LLM (much of which was described in previous sections), as well as generation of an effective algorithm for design optimization. Although LLMs cannot natively search for optimal solutions to a novel problem, they can make

educated starting guesses and output optimization code that users can execute. Much of this section is thus focused on prompting LLMs to generate meaningful code for problems dependent on aspects such as their parameterization support (e.g. continuous versus discrete domains), performance objective landscape, or fabrication constraints. Real-world problems introduce nuanced challenges, including exploring over multiple competing objectives, difficult-to-specify objectives (such as aesthetics and objectives that depend on long-term use), and an evolving landscape of emerging methods that an LLM may not know about. In this context, we could consider whether GPT-4 can propose strategies (even novel ones) that free designers from some of the typical burdens associated with the optimization pipeline.

With these considerations, we aim to investigate the following questions:

Q1 When can GPT-4 solve a problem analytically, and when does it need to resort to using an outside tool (e.g., a programmed algorithm)?

Q2 Can GPT-4 choose reasonable algorithms for different types of supports for constraints, objectives, and decision spaces (e.g., continuous, discrete, binary)?

Q3 Can GPT-4 assist designers in navigating the landscape of possible trade-offs when multiple conflicting objectives are present?

Q4 Can GPT-4 support optimization in contexts that require additional knowledge, specifically when a design space is not properly defined or is missing constraints?

In this section, we investigate, generally speaking, modern LLMs' abilities to navigate and (semi-)automate design optimization problems.

10.1. GPT-4: Analytical vs. Outside Tools (Q1)

We know that GPT-4 has the ability to reason about many mathematical operations, including both algebra and calculus, which is sufficient to solve many real-world engineering problems. We emphasize 'reasoning' because, although GPT-4 clearly proposes reasonable analysis steps, it is not obvious that GPT-4 is correctly executing those steps; as we will see, GPT-4 often makes mathematical errors. Still, it is reasonable to wonder if GPT-4's own internal reasoning is sufficient for inverse design. Where are the limits of that reasoning? When must it resort to code and external libraries, or plugins? Each of these approaches has its own pitfalls that suggest caution for developers.

From experiments detailed in [Appendix F.1](#), we see that GPT-4 is often able to apply intuitive physical reasoning principles to reason about an optimal solution within given bounds in this case. Further, by using plugins (such as the Wolfram plugin), GPT-4 can use external services to help solve its problem. However, such plugins can time out. However, GPT-4 can still recover from a plugin failure and resort to its own native mathematical reasoning. Meanwhile, when asked to solve a problem by generating code, it typically chooses

appropriate optimization methods, but not always state-of-the-art optimization methods, and does not always use them to maximal effect.

Throughout these experiments, we noticed several common issues in GPT-4’s approach. First, if users do not prompt GPT-4 explicitly to show its work, it may resort to ‘common-sense’ reasoning about a problem. Although this reasoning *could* be correct, GPT-4 provides no certificate to a user. Another issue occurs if it is difficult to find a library to solve a particular task; in this case, GPT-4 often gives up or attempts to write its own code. If the code is detail-heavy, it may be too difficult for GPT-4 to write correctly and the code/solution may be incorrect. If a library does exist but is used uncommonly, GPT-4 may give incorrect instructions on how to install/use that library; or, in some cases, GPT-4 may hallucinate a library altogether.

10.2. Reasoning About Different Problem Types and Selecting Appropriate Solvers (Q2)

To test GPT-4’s understanding of various problem domains and its ability to identify appropriate solutions for each, we conducted several experiments spanning a wide range of search spaces, constraint spaces, and performance spaces. In some cases, the problems have additional real-world considerations of which GPT-4 must be cognizant in order to choose a suitable optimization approach. Tables 2 and 3 provide a comparison of different problems that GPT-4 was asked to solve. We describe each example in additional detail in [Appendix E.2](#), with the exception of the table stability optimization (which was presented in [Appendix E.1](#)).

Overall, we found that even over varying problem types, GPT-4 exhibits extreme robustness when reasoning about and choosing an adequate solver for any given problem. In cases where a more sophisticated algorithm was needed, it tended to choose at least the correct algorithm class, even if it was not always aware of the best version or implementation.

Table 2. Descriptions of search and constraint space of inverse design problems.

Inverse Design Problems		
Problem Name	Search Space	Constraint Space
Table Stability	Continuous	Parameter Bounded
Robot Arm	Continuous	Parameter Bounded & Continuous Function
3D Printer Parameters	Continuous	Parameter Bounded
Cabinet Optimization	Continuous	Parameter Bounded
Robot Arm Planning	Continuous	Continuous

Chair Design	Continuous	Continuous & Bounded
--------------	------------	----------------------

Table 3. Results of the inverse design queries to GPT-4.

Inverse Design Problems			
Problem Name	Objective Space	Other Constraints	Chosen Optimization Method
Table Stability	Continuous	None	Analytical/Second-Order Gradient-Based
Robot Arm	Continuous	None	Second-Order Gradient-Based
3D Printer Parameters	Continuous	Expensive Real-World Experiments	Bayesian Optimization
Cabinet Optimization	Function Bounded	None	Second-Order Gradient-Based
Robot Arm Planning	Continuous	Logical Reasoning with High-Level Primitives	Greedy Search, Brute Force
Chair Design	Continuous	Multi-Objective	NSGA-II (Evolutionary Algorithm)

10.3. Navigating Trade-offs Between Multiple Conflicting Objectives (Q3)

Although our previous experiments focused on optimizing a single performance objective, we use the experiments in Appendix F.3 to explore the scenario where a user wishes to navigate a higher dimensional (i.e., multi-objective) performance space. The user begins by asking GPT-4 for reasonable performance metrics for evaluating an object. Once GPT-4 provides such metrics, a user can choose appropriate ones for their problem. The user can then ask for parameters over which to search. In our experience, when a design space is not formally defined first, the parameters and the design space may need to be iterated on with GPT-4. Once a design space is properly defined, the user can request code to evaluate and optimize the design. In practice, these problems can become lengthy and unwieldy, and lengthy conversations can cause GPT-4 to forget earlier parts of the conversation, such as initial parameter ranges. This can lead to the user needing to ‘remind’ GPT-4 of its previous answers.

Despite these issues, we conclude that GPT-4 has the potential to aid users in both a) understanding the trade-offs involved in different candidate designs, and b) providing pointers to a reasonable algorithm that can help navigate that space.

10.4. Supporting Optimization in Contexts that Require Additional Knowledge (Q4)

In many cases, it can be daunting to fully specify a given inverse design problem in a new domain: for example, it may be difficult to specify appropriate design spaces and objective functions, and it may be unclear how to deal with underspecified/unknown constraints. In this section, we briefly examine how LLMs may reduce the burden of this process to make inverse design more accessible.

For completely novel problems, GPT-4 cannot rely on its existing knowledge to generate an exact design space. However, it can apply knowledge of aspects of a problem to new problems in familiar domains. The conversation in [Figure 23](#) presents a brief example of GPT-4 being queried about a novel invention: the Fworp.² The Fworp is a robot car with a body made of silicone rubber. While the value of such a device is unclear (perhaps shock absorption), it is synthesized from existing ideas: namely, remote control vehicles and soft robotics. GPT-4 uses its knowledge of those preexisting domains and their components to recommend reasonable design parameters, and their ranges, including analyzing size, weight, wheel size, power source, peripherals/sensors, and build material properties. It also provides guidance on performance metrics without being prompted, but classifies these under ‘parameters,’ which may confuse users. Further, when queried about the advantages and disadvantages of such a device when compared with nonrubbery autonomous vehicles and soft robots without wheels, it provides reasonable comparisons. In particular, it notes that, compared with nonsoft robot vehicles, the Fworp could (possibly) be more durable, shock absorbent, safer, and quieter, while also potentially being more expensive to produce and tacky. Compared with nonvehicular soft robots, it has the potential to be more mobile, stable, energy-efficient, and simple to produce and control, but would lack the versatility and human-interaction potential typically afforded by most soft robots; further, while safer than a typical vehicle, it would be more dangerous than most current soft robots.

This experiment highlights the notion that the GPT-4 can be an effective partner when formulating a novel inverse design problem, as it can make connections between the proposed problems and more established domains. Then, GPT-4 is able to use its existing knowledge base about those related domains to provide reasonable starting points for the problem at hand. With continued user interaction, GPT-4 can also help to refine, formalize, evaluate, and ultimately act upon the newly created formulation.

 I am creating a new device called a fworp. A fworp is a wheeled soft silicone autonomous vehicle. What are some reasonable parameters I should consider for my fworp, and parameter ranges I should I consider?

 (... content omitted by authors ...)

- (1) **Size and Shape:** This is crucial to determine the potential applications of your fworp. You might want to keep the size compact if it is intended for home use or increase the size if it's designed for industrial applications. The shape will also play a crucial role in how the fworp navigates its environment. For the parameter range, consider anywhere from a small sphere (e.g., 10 cm in diameter) for consumer use, up to larger sizes (e.g., 2 m in diameter) for commercial/industrial use.
- (2) **Weight:** The weight will impact the maneuverability and the energy consumption of the device. A lighter fworp would be easier to move and consume less energy but might be less stable. A heavier one could be more stable but require more energy. The weight could range from 1 kg (for small consumer devices) to 50 kg or more (for larger industrial devices).
- (3) **Speed:** You'll want to adjust the speed depending on the use case. A fworp designed for a warehouse may need to be faster than one for home use to increase efficiency. Speeds might range from 1 km/h for careful navigation in domestic environments up to 20 km/h for more industrial settings.
- (4) **Battery Life:** This will dictate how long the fworp can function without needing to be recharged. For domestic use, a battery life of several hours might be sufficient, whereas for industrial use, a battery life of several shifts may be needed. Depending on usage, anywhere from 4 to 24 hours could be appropriate.
- (5) **Load Capacity:** Depending on the intended application, the fworp may need to carry items of varying weight. This could range from 1 kg for lightweight items up to 100 kg or more for industrial applications.

(... content omitted by authors ...)

(... content omitted by authors ...)

 Do you think the fworp is a good idea, compared to a non-rubbery counterpart? Why or why not?

 (... content omitted by authors ...)

Advantages of a Silicone Fworp

- (1) **Flexibility:** Silicone is a very flexible material, and a vehicle made of it could potentially navigate around obstacles or squeeze into spaces that a rigid vehicle could not.
- (2) **Durability:** Silicone is known for its durability and can withstand various environmental conditions, such as wide temperature ranges, UV radiation, and exposure to various chemicals. This could lead to a longer lifespan and less maintenance compared to some non-rubber materials.
- (3) **Safety:** A soft, flexible vehicle could potentially be safer in environments where it could come into contact with humans or delicate objects. A softer impact could reduce the risk of harm or damage.
- (4) **Noise reduction:** Soft materials like silicone could potentially operate more quietly than hard materials, which might be beneficial in certain environments, such as residential areas or hospitals.

Challenges with a Silicone Fworp

- (1) **Structural integrity:** Silicone may not provide the same level of structural integrity as harder materials. You would need to ensure that the fworp can carry the required loads without deforming excessively. This could be particularly challenging if the fworp is large or needs to carry heavy items.
- (2) **Speed and Efficiency:** The soft, flexible nature of silicone might mean that the fworp moves more slowly or with less efficiency than a harder vehicle. The flexibility of the material could absorb some of the energy meant for propulsion.
- (3) **Maintenance:** While silicone is durable, it can also be prone to punctures and cuts. Depending on the environment, the surface may need regular inspections and repairs.
- (4) **Cost:** Depending on the specifics, manufacturing a vehicle out of silicone could potentially be more expensive than using more traditional materials. (... content omitted by authors ...)

Figure 23. Testing GPT-4's domain transfer ability for inverse design. We query GPT-4 about the Fworp, an imaginary invention for which it has no direct knowledge. Despite this, GPT-4 is able to transfer its knowledge about related concepts to propose design parameters and comparisons. However, some of its proposed parameters, such as load capacity, are actually performance metrics.

10.5. Discussion

This section elaborates on GPT-4’s key capabilities (C), limitations (L), and dualisms (D) in the realm of inverse design.

C.1 Extensive Knowledge Base in Design and Manufacturing: GPT-4 has knowledge of how to formulate design spaces, objectives, and constraints. It also successfully selects suitable search algorithms for a given problem, suggesting that LLMs are useful as a building block when formulating inverse design systems. In its current form, GPT-4 exhibits a number of abilities that make it highly usable. For example, it was able to choose an adequate design optimization algorithm for almost every problem it was given; when asked, GPT-4 was also able to justify its choice of algorithm.

GPT-4 is also helpful in automatically providing code for a significant portion of a problem formulation without requiring user input. These aspects include parameter choice, parameter ranges, and objective functions. In the best case, this feature can relieve a user of much of the ‘busywork’ associated with a problem (loose bounds, necessary constraints, etc...). Even when GPT-4 falls short of this ideal, GPT-4 is usually able to recommend a useful starting point.

GPT-4’s reasoning capabilities can also further provide value in novel domains. If a user is inexperienced with a particular domain or if they are working on a novel problem, GPT-4 has the capability to synthesize from the problem’s constituent domains to provide suitable advice, as demonstrated with the Fworp example.

L.1 Reasoning Challenges: When asking for help in setting up a problem, GPT-4’s advice can be confusing. For example, it often does not disambiguate between the design parameters (which practitioners have direct control over) and performance metrics (which are emergent from the design). Less experienced designers may then find themselves confused, believing there must be a way to modify a system’s performance directly.

Potential Solutions: By following up with GPT-4 about how a given ‘parameter’ is computed, one can attempt to disambiguate parameters from metrics. In general, however, this verbal confusion is difficult to systematically address.

The addition of function calling in LLMs, and specifically plugins using GPT-4, can eventually allow for direct execution of arbitrary code, even code that GPT-4 writes. However, there are no guarantees on the execution time of that code, and it is unclear how to manage problems that might arise, such as long runtimes (which are common in hard optimization problems), or even infinite loops. In our experiments, the Wolfram plugin was given a brief time window for computation before it timed out, which largely negated its value in the face of more challenging problems.

Potential Solutions: Methods to allow for function calling while providing guarantees or control by a user (say, in the form of anytime algorithms) would be beneficial. For now, writing one’s own plugin may allow greater

granularity over the type of algorithm being used. Thus, the algorithms can at least be catered to GPT-4's behavior.

L.3 Scalability: Inverse design relies on several complex building blocks, including the specification of design spaces and objective functions. However, as discussed in previous sections, GPT-4 frequently encounters difficulties when faced with these tasks. Such errors prohibit GPT-4 from scaling to inverse design exploration altogether. This occurred twice during our experiments. In one failure case, we had difficulty in evaluating the performance of a soft body system using finite elements; although the example is not detailed in the article, [Section 9](#) has already shown this to be difficult. In effect, this failure currently prevents GPT-4-assisted inverse design of a soft robot with respect to FEA-derived metrics. In a different example, we attempted to design a long, multilink arm, but found that GPT-4 struggled with properly geometrical alignment of the links and rotation axes (as shown in [Appendix B.5](#)).

Potential Solutions: Pointing out problems with solutions (such as runtime errors) can allow GPT-4 to iterate, but requires intervention and potentially fine-grain coding or engineering knowledge by a user. In practice, it is often effective to blindly ask GPT-4 to assess its own output and report any errors it finds until GPT-4 is satisfied with its own work. In our experiments, this frequently converged to a correct solution. However, this is not foolproof and can be slow and computationally costly. Further, without access to web search, GPT-4 may not know how to reconcile out-of-date knowledge about an API.

A second scalability issue is that GPT-4 does not always choose the best algorithm for solving a problem, and sometimes does not use a given method in the most efficient way (such as not providing gradient information). Since GPT-4 tends to be coy about available methods and how to best use them, a more novice user may be unsure how to navigate the intricacies of optimization and diagnose issues. Furthermore, although GPT-4 tends to choose adequate algorithm classes, it does not always choose state-of-the-art methods; instead, it tends to default to standard methods that are highly popular. Because of its knowledge cutoff, without access to web search, any given LLM may not be aware of state-of-the-art methods or how to implement them. Even if an easy-to-use implementation exists on an online repository (e.g., GitHub), the LLM may not be aware of the code's existence or how to use it.

Potential Solutions: Web search, which has been previously available for GPT-4, can help to mitigate these issues, as one could ask for the latest, state-of-the-art methods, and GPT-4 could provide solutions based on current repository code. However, there is no guarantee that GPT-4 will be able to understand what makes newer methods optimal for a problem without sizeable crowd knowledge, which may not be available.

The third scalability issue is that, as mentioned in previous sections, GPT-4's 'short memory' can cause it to forget specifics it had generated earlier in a context; this notably occurred in the multiobjective chair example shown in Appendix F.3. While this problem emerged in other aspects of the design-to-manufacturing pipeline, its impacts were most salient when defining inverse problems, whose specification can be especially long.

Potential Solutions: Since inverse design problems can be quite lengthy to define and specify, it may be easier to decompose a problem in the following order: 1) Ask GPT-4 for a definition of a design space (including its implementation), 2) Ask GPT-4 for a definition of a performance metric and constraints (including their implementations), while abstracting away the code from 1) as an API call; 3) Ask GPT-4 to write code for the inverse design search, abstracting away the code from 2) as an API call. This can keep definitions shorter and easier to manage.

D.2 Unprompted Responses: Throughout our experiments, GPT-4 always unilaterally selected an optimization algorithm and proceeded to generate code. In particular, GPT-4 never provided the user with options for possible optimization algorithms, unless it was explicitly asked to provide such options as an intermediate step. Although GPT-4’s automated selection may satisfy many users, it runs the risk of creating mistakes that would be difficult to diagnose. This is particularly true because GPT-4 rarely justified its choice to users. Furthermore, GPT-4’s assertion may imply that there is a single ‘correct’ algorithm for a given problem, and users may not realize that there are better (or even *alternative*) options available in any given circumstance.

GPT-4’s tendency to fill in aspects of an inverse design problem before being asked about them may also lead to mathematical problem definitions that are ill-suited or otherwise suboptimal for a user’s real-world design problem. In these cases, GPT-4’s tendency to autocomplete and plow ahead could lead users to blindly follow the LLM down bad ‘rabbit holes,’ only to discover that a fundamental problem existed much earlier. Furthermore, since GPT-4 does not have a native way to execute arbitrary code, it will not always realize that a codeblock has errors.

11. Part 1: Conclusions

Our experiments demonstrate potential benefits and limitations of integrating LLMs into components of the CDaM workflow.

For generating a design from text, GPT-4 is often able to correctly identify the high-level structure and the type of constituent primitives, often outputting modular and hierarchical code. However, GPT-4 struggles to place objects correctly or to obey user-specified constraints. During iterative prompting, we found that GPT-4 can only address one to two issues at a time, limiting its scalability to complex designs.

For constructing a design space, GPT-4’s extensive knowledge base helps it create bounds and constraints on text-based design, as well as interpolate existing designs given their programs. Yet, we observe instances where GPT-4 relies excessively on its semantic knowledge instead of a given design’s geometry and does not always accurately link semantics to the corresponding component in the design.

For preparing designs for manufacturing, GPT-4 possesses knowledge on a variety of manufacturing processes and can directly modify code for automated manufacturing. However, its capability to reason about

manufacturing code is limited, and experiments show that it can be more beneficial to have GPT-4 output intermediate symbolic computations that produce the resulting manufacturing file instead.

For evaluating a design's performance, GPT-4 demonstrated proficiency in discussing various performance metrics. However, the quality of GPT-4's assessment varied, as GPT-4 often relies on semantic clues in the prompt rather than reasoning about the physical attributes of the design. Output verification is also a key concern, as GPT-4 often hallucinates external libraries or neglects to cite source material for equations.

For discovering high-performing designs based on a given performance metric and design space, GPT-4 can provide a starting point for formulating design spaces, objectives, and constraints, as well as provide high-level advice, such as suggesting an appropriate search algorithm. However, GPT-4's responses can be misleading, sometimes conflating design parameters with performance metrics. As inverse design inherently relies on design space and objective function specifications, the errors made by GPT-4 on these preliminary tasks can often hinder inverse design exploration.

For each aspect of the CDaM workflow, LLMs can provide value, but users must be aware of their strengths, weaknesses, and limitations throughout—especially those that might not be obvious from the outset. We hope that this article provides insight for users developing current and future AI-powered workflows.

Disclosure Statement

This material is based upon work supported in part by Defense Advanced Research Projects Agency (DARPA) Grant No. FA8750-20-C-0075, DARPA Fellowship Grant No. HR00112110007, and the National Science Foundation (NSF) under Grant No. 2141064. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors(s) and do not necessarily reflect the views of the National Science Foundation.

References

Autodesk. (n.d.). *Autodesk simulation*. Retrieved July 14, 2023, from
<https://www.autodesk.com/solutions/simulation/overview>

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., . . . Zaremba, W. (2021). *Evaluating large language models trained on code*. ArXiv.
<https://doi.org/10.48550/arXiv.2107.03374>

Christianio, P. F., Leike, J., Brown, T., Martic, M., Legg, S., & Amodei, D. (2017). Deep reinforcement learning from human preferences. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, &

R. Garnett (Eds.), *Advances in neural information processing systems* (Vol. 30, pp. 4299–4307). Curran Associates. https://papers.nips.cc/paper_files/paper/2017/hash/d5e2c0adad503c91f91df240d0cd4e49-Abstract.html

Dassault Systèmes. (n.d.). *Dassault Systèmes simulation*. Retrieved July 14, 2023, from <https://www.3ds.com/products-services/simulia/overview/>

Dhariwal, P., Jun, H., Payne, C., Kim, J. W., Radford, A., & Sutskever, I. (2020). *Jukebox: A generative model for music*. ArXiv. <https://doi.org/10.48550/arXiv.2005.00341>

Du, T., Inala, J. P., Pu, Y., Spielberg, A., Schulz, A., Rus, D., Solar-Lezama, A., & Matusik, W. (2018). InverseCSG: Automatic conversion of 3D models to CSG trees. *ACM Transactions on Graphics (TOG)*, 37(6), Article 213. <https://doi.org/10.1145/3272127.3275006>

Du, T., Wu, K., Ma, P., Wah, S., Spielberg, A., Rus, D., & Matusik, W. (2021). DiffPD: Differentiable projective dynamics. *ACM Transactions on Graphics (TOG)*, 41(2), Article 13. <https://doi.org/10.1145/3490168>

Du, Y., Li, S., Torralba, A., Tenenbaum, J. B., & Mordatch, I. (2023). *Improving factuality and reasoning in language models through multiagent debate*. ArXiv. <https://doi.org/10.48550/arXiv.2305.14325>

Erez, T., Tassa, Y., & Todorov, E. (2015). Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX. In Amato, N. (Ed.), *Proceedings of the 2015 IEEE International Conference on Robotics and Automation* (pp. 4397–4404). IEEE. <https://doi.org/10.1109/ICRA.2015.7139807>

Erps, T., Foshey, M., Luković, M. K., Shou, W., Goetzke, H. H., Dietsch, H., Stoll, K., von Vacano, B., & Matusik, W. (2021). Accelerated discovery of 3D printing materials using data-driven multiobjective optimization. *Science Advances*, 7(42), Article eabf7435. <https://doi.org/10.1126/sciadv.abf7435>

Featurescript introduction. (n.d.). Retrieved July 11, 2023, from <https://cad.onshape.com/FsDoc/>

Ferruz, N., Schmidt, S., & Höcker, B. (2022). ProtGPT2 is a deep unsupervised language model for protein design. *Nature Communications*, 13(1), Article 4348. <https://doi.org/10.1038/s41467-022-32007-7>

Guo, M., Thost, V., Li, B., Das, P., Chen, J., & Matusik, W. (2022). *Data-efficient graph grammar learning for molecular generation*. ArXiv. <https://doi.org/10.48550/arXiv.2203.08031>

Jiang, B., Chen, X., Liu, W., Yu, J., Yu, G., & Chen, T. (2023). *MotionGPT: Human motion as a foreign language*. ArXiv. <https://doi.org/10.48550/arXiv.2306.14795>

JSCAD user guide. (n.d.). Retrieved July 14, 2023, from <https://openjscad.xyz/dokuwiki/doku.php>

Kashefi, A., & Mukerji, T. (2023). ChatGPT for programming numerical methods. *Journal of Machine Learning for Modeling and Computing*, 4(2), 1–74.

<https://doi.org/10.1615/JMachLearnModelComput.2023048492>

Koo, B., Hergel, J., Lefebvre, S., & Mitra, N. J. (2017). Towards zero-waste furniture design. *IEEE Transactions on Visualization and Computer Graphics*, 23(12), 2627–2640.

<https://doi.org/10.1109/TVCG.2016.2633519>

Li, J., Rawn, E., Ritchie, J., Tran O’Leary, J., & Follmer, S. (2023). Beyond the artifact: Power as a lens for creativity support tools. In Follmer, S., Han, J., Steimle, J., & Riche, N. H. (Eds.), *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (Article 47). Association for Computing Machinery. <https://doi.org/10.1145/3586183.3606831>

Liu, R., Wu, R., Van Hoorick, B., Tokmakov, P., Zakharov, S., & Vondrick, C. (2023). Zero-1-to-3: Zero-shot one image to 3D object. ArXiv. <https://doi.org/10.48550/arXiv.2303.11328>

Ma, P., Du, T., Zhang, J. Z., Wu, K., Spielberg, A., Katzschmann, R. K., & Matusik, W. (2021). DiffAqua: A differentiable computational design pipeline for soft underwater swimmers with shape interpolation. *ACM Transactions on Graphics (TOG)*, 40(4), Article 132. <https://doi.org/10.1145/3450626.3459832>

Makatura, L., Wang, B., Chen, Y.-L., Deng, B., Wojtan, C., Bickel, B., & Matusik, W. (2023). Procedural metamaterials: A unified procedural graph for metamaterial design. *ACM Transactions on Graphics*, 42(5), Article 168. <https://doi.org/10.1145/3605389>

Mathur, A., & Zufferey, D. (2021). Constraint synthesis for parametric CAD. In M. Okabe, S. Lee, B. Wuensche, & S. Zollmann (Eds.), *Pacific Graphics 2021: The 29th Pacific Conference on Computer Graphics and Applications: Short Papers, Posters, and Work-in-Progress Papers* (pp. 75–80). The Eurographics Association. <https://doi.org/10.2312/pg.20211396>

Mirchandani, S., Xia, F., Florence, P., Ichter, B., Driess, D., Arenas, M. G., Rao, K., Sadigh, D., & Zeng, A. (2023). Large language models as general pattern machines. ArXiv. <https://doi.org/10.48550/arXiv.2307.04721>

Müller, P., Wonka, P., Haegler, S., Ulmer, A., & Van Gool, L. (2006). Procedural modeling of buildings. In *ACM SIGGRAPH 2006 papers* (pp. 614–623). Association for Computing Machinery. <https://doi.org/10.1145/1179352.1141931>

Ni, B., & Buehler, M. J. (2024). MechAgents: Large language model multi-agent collaborations can solve mechanics problems, generate new data, and integrate knowledge. *Extreme Mechanics Letters*, 67, Article 102131. <https://doi.org/10.1016/j.eml.2024.102131>

O'Brien, J. F., Shen, C., & Gatchalian, C. M. (2002). Synthesizing sounds from rigid-body simulations. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (pp. 175–181). Association for Computing Machinery. <https://doi.org/10.1145/545261.545290>

OpenAI. (2023). *GPT-4 technical report*. ArXiv. <https://doi.org/10.48550/arXiv.2303.08774>

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., & Lowe, R. (2022). Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, & A. Oh (Eds.), *Advances in Neural Information Processing Systems* (Vol. 35, pp. 27730–27744). Curran Associates. https://proceedings.neurips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html

Özkar, M., & Stiny, G. (2009). Shape grammars. In *ACM SIGGRAPH 2009 courses* (Article 22). Association for Computing Machinery. <https://doi.org/10.1145/1667239.1667261>

Penedo, G., Malartic, Q., Hesslow, D., Cojocaru, R., Cappelli, A., Alobeidli, H., Pannier, B., Almazrouei, E., & Launay, J. (2023). *The RefinedWeb dataset for Falcon LLM: Outperforming curated corpora with web data, and web data only*. ArXiv. <https://doi.org/10.48550/arXiv.2306.01116>

Prusinkiewicz, P., & Lindenmayer, A. (1990). *The algorithmic beauty of plants*. Springer Science & Business Media.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019, February 14). *Language models are unsupervised multitask learners*. OpenAI. <https://openai.com/index/better-language-models/>

Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., & Sutskever, I. (2021). Zero-shot text-to-image generation. *Proceedings of Machine Learning Research*, 139, 8821–8831. <https://proceedings.mlr.press/v139/ramesh21a.html>

Repetier. (n.d.). *Repetier software*. Retrieved July 20, 2023, from <https://www.repetier.com/>

Richards, T. B. (n.d.). *AutoGPT*. Retrieved February 11, 2024, from <https://github.com/Significant-Gravitas/AutoGPT>

Rozenberg, G., & Salomaa, A. (1980). *The mathematical theory of L systems*. Academic Press.

Slic3r. (n.d.). *Slic3r - Open source 3D printing toolbox*. Retrieved July 20, 2023, from <https://slic3r.org/>

Stiny, G. (1980). Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7(3), 343–351. <https://doi.org/10.1068/b070343>

- Sullivan, D. M. (2013). *Electromagnetic simulation using the FDTD method*. John Wiley & Sons.
<https://doi.org/10.1002/9781118646700>
- Todorov, E., Erez, T., & Tassa, Y. (2012). MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 5026–5033). IEEE.
<https://doi.org/10.1109/IROS.2012.6386109>
- Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1), 230–265. <https://doi.org/10.1112/plms/s2-42.1.230>
- Willis, K. D., Pu, Y., Luo, J., Chu, H., Du, T., Lambourne, J. G., Solar-Lezama, A., & Matusik, W. (2021). Fusion 360 gallery: A dataset and environment for programmatic CAD construction from human design sequences. *ACM Transactions on Graphics (TOG)*, 40(4), Article 54. <https://doi.org/10.1145/3450626.3459818>
- Xu, J., Chen, T., Zlokapa, L., Foshey, M., Matusik, W., Sueda, S., & Agrawal, P. (2021). *An end-to-end differentiable framework for contact-aware robot design*. ArXiv. <https://doi.org/10.48550/arXiv.2107.07501>
- Yu, L., Shi, B., Pasunuru, R., Muller, B., Golovneva, O., Wang, T., Babu, A., Tang, B., Karrer, B., Sheynin, S., Ross, C., Polyak, A., Howes, R., Sharma, V., Xu, P., Tamoyan, H., Ashual, O., Singer, U., . . . Aghajanyan, A. (2023). *Scaling autoregressive multi-modal models: Pretraining and instruction tuning*. ArXiv.
<https://doi.org/10.48550/arXiv.2309.02591>
- Zhang, Y., Yang, M., Baghdadi, R., Kamil, S., Shun, J., & Amarasinghe, S. (2018). GraphIt: A high-performance graph DSL. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), Article 121.
<https://doi.org/10.1145/3276491>
- Zhao, A., Xu, J., Konaković-Luković, M., Hughes, J., Spielberg, A., Rus, D., & Matusik, W. (2020). RoboGrammar: Graph grammar for terrain-optimized robot design. *ACM Transactions on Graphics (TOG)*, 39(6), Article 188. <https://doi.org/10.1145/3414685.3417831>
- Zilberstein, S. (1996). Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3), 73–83.
<https://doi.org/10.1609/aimag.v17i3.1232>
-

Appendix

Appendix A. Domain-Specific Languages and Prompting Tips for Generating Designs From Natural Language With GPT-4

A.1. Constructive Solid Geometry With OpenJSCAD

GPT-4 was able to use the OpenJSCAD library out-of-the-box, with no additional explanation or restriction of the application programming interface (API) on the part of the user. However, as described in [Section 6.1.1](#), GPT-4 did fall into a number of common pitfalls when constructing designs. To mitigate the most common mistakes that GPT-4 made, each time we asked GPT-4 to build a design using OpenJSCAD, we provided the set of hints and reminders shown in [Figure A1](#).

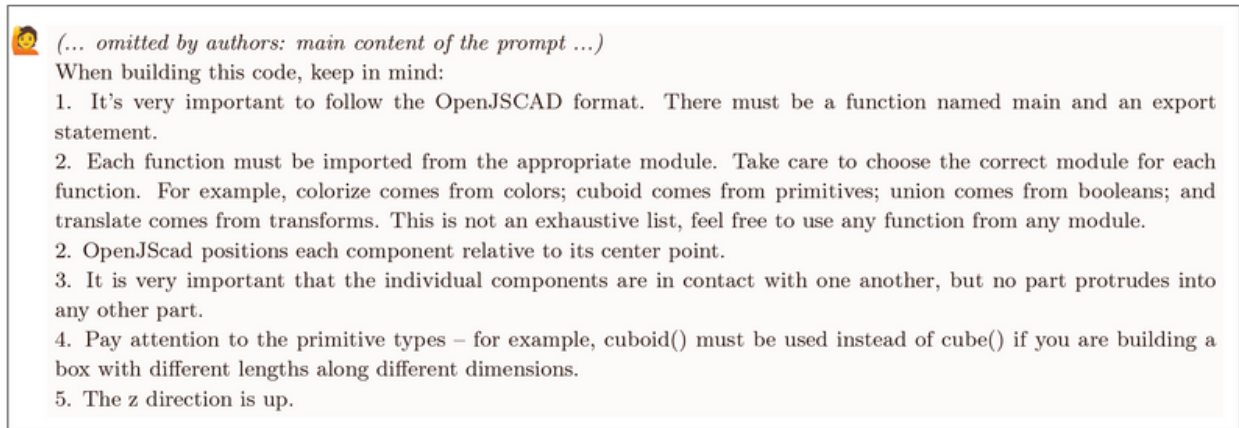


Figure A1. Hints for using OpenJSCAD. Each time we asked GPT-4 to construct a design using OpenJSCAD, we provided the following hints after the main prompt to avoid the most common pitfalls that GPT-4 fell into.

A.2. Sketch-Based Parametric Computer-Aided Design DSL

We propose a streamlined version of the standard sketch-based CAD language by exposing only the sketch and extrude operations along with basic sketch primitives, which already cover a wide range of geometric variations. To automatically generate CAD models from GPT-4's output, we utilize Onshape's API. When aiming for single-shot CAD design (i.e., with no iterative feedback), we found that a four-pronged prompt generally resulted in the most reliable output. One aspect of the prompt described the specific task that GPT-4 should complete. The remaining three aspects of the prompt provided generic context for our target CAD DSL, and largely remained constant throughout our experiments. The specific aspects were: (1) a description of our modified DSL, (2) an example constructed with this DSL, and (3) a set of tips that GPT-4 should keep in mind when constructing its own result.

The prompt we used to describe these aspects to work with local coordinate systems and a global coordinate system can be seen in chat format in [Figure A2](#) and [Figure A3](#), respectively.



We will define a design language. There are two key operators:

1) `createSketch(primitive, plane)`: creates a sketch of a certain primitive on a given plane and returns its ID. There are two types of primitives you can create, circles or rectangles. The circle primitive is `circle(center_x, center_y, radius)`, where `center_x` is the x coordinate of the center, `center_y` is the y coordinate and `radius` is the radius. The rectangle primitive is instantiated `rectangle(center_x, center_y, length, width)`. The plane defines the 2D plane where you will draw the sketch. You can just use one of the 3 default places: `XY_PLANE`, `XZ_PLANE`, `ZY_PLANE`. You can also use the plane create by the result on an extrude, which you do by calling the function `cap(extrude, side)`, where 'extrude' is the Id returned by the extrude operation (defined below). And side is one of the following: `"max_z"`, `"min_z"`, `"max_y"`, `"min_y"`, `"max_x"`, `"min_x"`. The side argument defines which planar face of an extruded solid will be used for the sketch operation. For example, `"min_z"` will select the planar face whose bounding box center has a minimal z-component compared to all other planes' bounding box centers. And there is the important constraint that you can only put the center of sketch primitives inside or on the edge of a planar face, if you do not put them on one of the default planes.

2) `extrude(sketch, length)`, where `sketch` is the ID returned by the sketch operator and the `length` determines the length of the extrude. Note that sketches lie within their respective plane, and they will get extruded along the plane's normal direction. A common pattern you will encounter is that the height variable of a solid should often be the length parameter in the extrude operator. Whereas the other dimensions of the solid are defined by the sketch primitives. For example, if you want to design a round table with a single center leg and a leg base you can do:

```
leg_base legBase_sketch = createSketch(circle(0,0,3), XY_PLANE)
legBase_solid = extrude(legBase_sketch, 1)

leg_sketch = createSketch(circle(0,0,1), cap(legBase_solid, "max_z"))
leg_solid = extrude(leg_sketch, 10)

top_sketch = createSketch(circle(0,0,8), cap(leg_solid, "max_z"))
top_solid = extrude(top_sketch, 1)
```

End of the example.

Always write code using variables. Try to prefer a few variables by reusing them in the design when appropriate. Write code in syntactically correct python, knowing that you have the functions `createSketch`, `circle`, `rectangle`, `cap` and `extrude` and the default planes.

Figure A2. A sketch-based CAD DSL prompt with local coordinate systems. Our prompt used for the sketch-based CAD experiments with local coordinate systems.



We will define a design language. There are two key operators:

1) `createSketch(primitive, plane)`: creates a sketch of a certain primitive on a given plane and returns its ID. There are two types of primitives you can create, circles or rectangles. The circle primitive is `circle(center_x, center_y, center_z, radius)`, where `center_x` is the x coordinate of the center, `center_y` is the y coordinate of the center, `center_z` is the z coordinate of the center and `radius` is the radius. The rectangle primitive is an instantiated rectangle (`center_x, center_y, center_z, length, width`). The plane defines the 2D plane where you will draw the sketch. You can just use one of the 3 default places: `XY_PLANE`, `XZ_PLANE`, `ZY_PLANE`. You can also use the plane created by the result on an extrude, which you do by calling the function `cap(extrude, side)`, where 'extrude' is the Id returned by the extrude operation (defined below). And side is one of the following: `"max_z"`, `"min_z"`, `"max_y"`, `"min_y"`, `"max_x"`, `"min_x"`. The side argument defines which planar face of an extruded solid will be used for the sketch operation. For example, `"min_z"` will select the planar face whose bounding box center has a minimal z-component compared to all other planes' bounding box centers. And `"max_x"` will select the planar face which has a maximal x component compared to all other planes' bounding box centers. Note that the normal vector of the selected planes directly correlates with the side argument. Here are the normal vectors associated to each side argument:

- `"min_z"` : (0, 0, -1)
- `"max_z"` : (0, 0, 1)
- `"min_y"` : (0, -1, 0)
- `"max_y"` : (0, 1, 0)
- `"min_x"` : (-1, 0, 0)
- `"max_x"` : (1, 0, 0)

You can use these normal vectors to create more 3 dimensional objects.

The center coordinates of sketch primitives have to be inside of the selected plane. And there is the important constraint that you can only put the center of sketch primitives inside or on the edge of a planar face, if you do not put them on one of the default planes.

2) `extrude (sketch, length)`, where `sketch` is the ID returned by the sketch operator and the length determines the length of the extrude. Note that sketches lie within their respective plane, and they will get extruded along the plane's normal direction. A common pattern you will encounter is that the height variable of a solid should often be the length parameter in the extrude operator. Whereas the other dimensions of the solid are defined by the sketch primitives.

Example design For example, if you want to design a round table with a single center leg and a leg base you can do:

```
leg_base legBase_sketch = createSketch(circle(0,0,0, 3), XY_PLANE)
legBase_solid = extrude(legBase_sketch, 1)

leg_sketch = createSketch(circle(0, 0, 1, 1), cap(legBase_solid, "max_z"))
leg_solid = extrude(leg_sketch, 10)

top_sketch = createSketch(circle(0,0, 11,8), cap(leg_solid, "max_z"))
top_solid = extrude(top_sketch, 1)
```

End of the example.

Additional Constraints Use exposed design variables whenever you can, and as few as possible. Write code in syntactically correct python, knowing that you have the functions `createSketch`, `circle`, `rectangle`, `cap` and `extrude` and the default planes.

Figure A3. A Sketch-Based CAD DSL Prompt with a Global Coordinate System. Our prompt used for the sketch-based CAD experiments with a global coordinate system.

A.3. Universal Robot Description Format

GPT-4 was able to use URDF without any intermediate libraries. Similar to OpenJSCAD, there were many common pitfalls that needed to be mitigated via prompt choice—these are discussed in detail in [Appendix B.5](#).

In brief summary, the following notes list some additions that were useful in mitigating specific problems:

- GPT-4 has difficulties in determining where URDF objects place their origin. When wanting objects to touch but not intersect, or be placed at the ‘end’ of other objects, it is useful to specify that the ends are half the length of the object away from the origin.
- Specifying an axis for two objects to be aligned along is more effective than instructing that they be aligned.
- GPT-4 will often omit essential parts of the URDF file for brevity, replacing them with a comment to repeat a part of the file. This can be done manually, but to generate URDF files that are complete directly from the response, GPT-4 must be instructed to produce a complete file.
- GPT-4 will ignore several constraints or instructions if too many are placed in a single prompt. Splitting the generation process into multiple prompts resolves this issue.

A.4. Graph-Based DSL for Robotics

The full text of the prompt used to generate the humanoid robot graph (omitted earlier for brevity) is shown in [Figure A4](#).

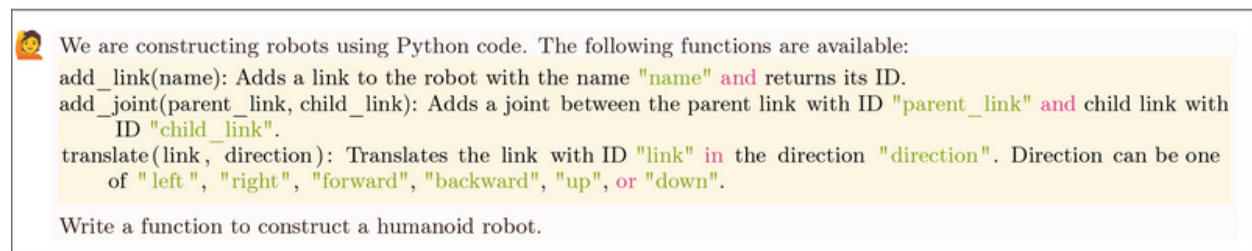


Figure A4. Graph-based robotics DSL. A description of the custom graph-based DSL used to construct robots.

Appendix B. Additional Experiments In Generating Designs from Natural Language Specifications

B.1. Vector Graphics With SVG/DXF

Our initial focus in the design domain is on 2D vector graphics. Vector formats such as SVGs or DXFs are prevalently utilized in manufacturing processes, like those for laser or waterjet cutting. The goal of our investigation was to ascertain whether GPT-4 could empower designers to transform their text directly into these vector formats. To evaluate this, we conducted experiments to determine if GPT-4 is capable of generating a valid SVG file and converting the design into DXF format.

The primary aim of our experiment was to design an SVG file for a cabinet, with predetermined dimensions, to be constructed from 1/2-inch plywood. This implies that the thickness of each wall, a preset parameter, is 0.5 inches. The experimental setup involved the design of a cabinet comprising three shelves, with overall dimensions measuring 6 feet in height, 1 foot in depth, and 4 feet in width. A crucial aspect of the investigation was to see if GPT-4 could accurately account for this wall thickness during the design of the cabinet,

appropriately adjusting the dimensions of its various components. GPT-4 was able to design the specified cabinet and subsequently generated a Python script to create an SVG file reflecting the cabinet's layout. The script considered the necessary clearances for the thickness and accurately positioned the side panels, top and bottom panels, shelves, and back panel. Moreover, it factored in the prescribed spacing between parts and leveraged 'svgwrite' to generate the SVG file. The resulting SVG file provided a visual depiction of the cabinet's design. We also replicated the experiment to create a DXF file, where GPT-4 utilized 'ezdxf' to generate the file. The results of these experiments are depicted in [Figure B5](#).

In conclusion, GPT-4 demonstrated its capability to employ the APIs for generating the vector file in the correct format without any simplifications. Nevertheless, it was necessary to perform several iterations to ensure GPT-4 did not cause any overlap among the cabinet parts.

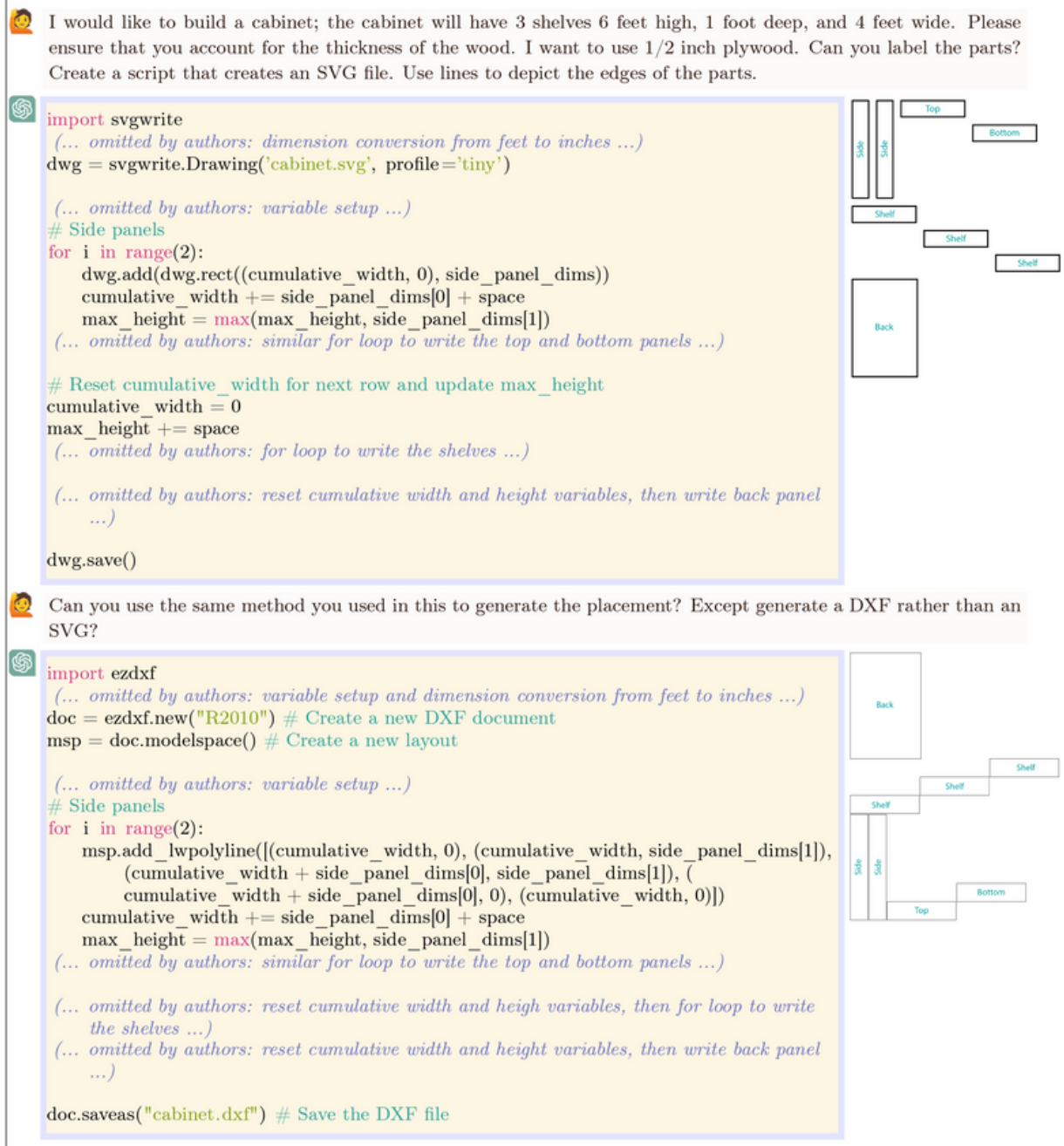


Figure B5. The prompts used for generating files in the format of SVG and DXF using GPT-4. The visualization of the graphs are shown on the right side. It is clear that GPT-4 can accomplish this task after several iterations.

B.2. CSG With Self-Defined Primitives

The next design domain we are investigating is CSG. As outlined in [Section 5.2.1](#), CSG languages generally operate by building up a collection of primitives that have been altered or combined via linear transformations

and Boolean operations. Because the associated design logic can be quite complex, it was not immediately clear that GPT-4 should be able to generate designs using these languages. Thus, to progressively test GPT-4's modeling capabilities, we begin by exploring a very simple, custom CSG language based on a single primitive: a box.

Boxes are one of the most common primitives seen in manufacturing. Moreover, many shapes can be considered as a combination of boxes with different sizes. Because of the simplicity of a box or any shape formed by the boxes, we would like to see if GPT-4 is able to generate designs of such kind of simple shapes, such as tables and chairs.

Our initial approach to this task is performed in 2D. We provide a function, $\text{foo}(x, y, w, h)$, which forms a box of dimensions $w \times h$ centred at the point (x, y) . We subsequently employ this function to generate letters composed of axis-aligned bars, such as 'F' and 'E.' During the testing phase, we observed that while the system understands the requirement of 2D boxes, it struggles with their accurate placement. A particularly prominent issue is the collision problem. More specifically, the GPT-4 system fails to determine whether two boxes are overlapping or whether there is a vacant space between them. This issue is observable when creating letters like 'T' and 'E.' Using three to five targeted prompts enabled GPT-4 to ascertain the correct positions. However, these prompts had to be granular and often involved providing the direct solution. The outcomes of these attempts are demonstrated in [Figure B6](#). Interestingly, after addressing this issue, GPT-4 appears to retain the corrections. This is evidenced by its successful generation of the new letters 'F' and 'L' in a single attempt. These letters share a similar structure to 'T' and 'E,' and the results can be seen in [Figure B6](#).

Our next step involved venturing into 3D, which holds more practical values. Analogous to the 2D scenarios, we inform GPT-4 of a preestablished function, $\text{box}(x, y, z, w, h, d)$, which generates a 3D box of dimensions $w \times h \times d$ centred at the 3D coordinates (x, y, z) . We then tested if GPT-4 could write a program to produce a simple box of specified dimensions, for instance, $100 \times 100 \times 40$, utilizing function 'box.' GPT-4 successfully accomplished this task, and the resulting text explanation illustrates its understanding of the box concept and the usage of our predefined function. Next, we presented a more complex challenge: having GPT-4 design a simple table, typically consisting of four legs and a tabletop in the real world. We posed the question of whether GPT-4 could craft a program to generate such a table with a provided size using solely our box function. The output text explanation revealed that GPT-4 accurately comprehends the structure of a basic table. Given that we only provide the overall table size, GPT-4 lacks information about individual leg lengths or tabletop thickness. Yet, it was able to identify these missing parameters and make reasonable assumptions. Consequently, GPT-4 succeeded in writing a program to represent the table by creating five boxes using our predefined function. Upon visualizing the 3D table, however, the relative positioning of each pair of boxes was not always accurate. We noticed that the tabletop appeared to be suspended in the air, not in contact with the legs, as shown in [Figure B7](#). This difficulty, also observed in our 2D tests ([Figure B6](#)), pertains to GPT-4's understanding of mathematical concepts. In this instance, we expedited the process by directly providing GPT-

4 with the solution. We indicated the necessary translations for the misplaced boxes, acknowledging that it would take several prompts to rectify the issue otherwise. After correcting the floating tabletop, the table appeared as intended, as demonstrated in [Figure B7](#). Therefore, to create a table, it only required two prompts, significantly streamlining the procedure for generating a basic table.

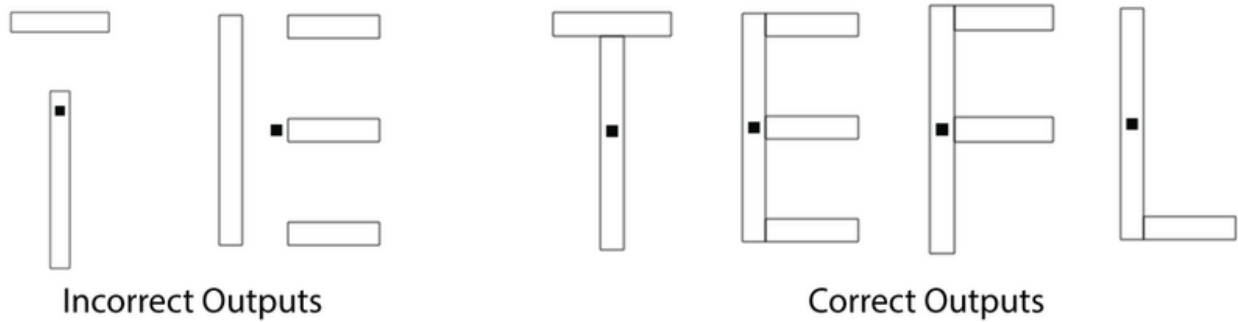


Figure B6. Failed and successful cases of letter creation using GPT-4. The solid square is the origin of the 2D coordinate system.

Once we successfully generate the table, our next more challenging goal is to design a few accompanying chairs. We tasked GPT-4 with creating a chair compatible with the table, using only our predefined function. Similar to its approach with the table, GPT-4 successfully deduced the basic structure of a simple chair, comprising the seat, four legs, and a backrest. Unlike the table instance, we did not observe any ‘floating’ issues in this scenario. It appears that GPT-4 might have indeed gleaned some insights from previous experiences, as we also observed when creating 2D letters. After we rectified the letters ‘T’ and ‘E,’ there were no issues with the remaining letters. Additionally, GPT-4 demonstrated comprehension of the concept of compatibility by outputting a chair of an appropriate size. However, it was not successful in all aspects, as depicted in [Figure B8](#). We attempted to correct the backrest but were unable to do so. As a result, we had to manually adjust the position, directing GPT-4 to the specific lines that needed modification to correct the structure. The final result can be seen in [Figure B8](#). We believe the root of these issues lies in GPT-4’s struggles to comprehend geometric concepts, a difficulty also observed in previous examples. Despite these hurdles, the process for creating a basic table and chairs has been considerably simplified.

Our final objective was to position four identical chairs around the table. Although theoretically feasible without invoking rotation, GPT-4 failed to generate the chairs with the correct orientations. We believe this failure stems from the same root cause we have encountered previously, namely, GPT-4’s difficulty in handling mathematical and geometric concepts. Creating four chairs with correct orientations without the support of rotation entails complex geometric transformations. GPT-4 must comprehend that a box rotated 90 degrees around its center is equivalent to a swap of its width and depth dimensions. To alleviate this issue, we expanded our ‘box’ function to include an additional input argument, ‘angle,’ corresponding to a rotation angle around the vertical axis. With this extension, GPT-4 was able to create a program using solely the ‘box’ function that successfully positioned four chairs around the table with correct orientations, as displayed in

[Figure B8](#). We surmise that the introduction of ‘angle’ considerably simplifies the logic behind chair placement, enabling GPT-4 to create such a program.

In conclusion, GPT-4 exhibits strong understanding of posed questions and excels at analyzing requested objects to determine their composition. However, it demonstrates a weakness in handling geometric and mathematical concepts. While it can provide nearly accurate solutions when mathematics is involved, it struggles to comprehend the underlying mathematical principles and, as a result, cannot independently correct math-related issues when they arise.

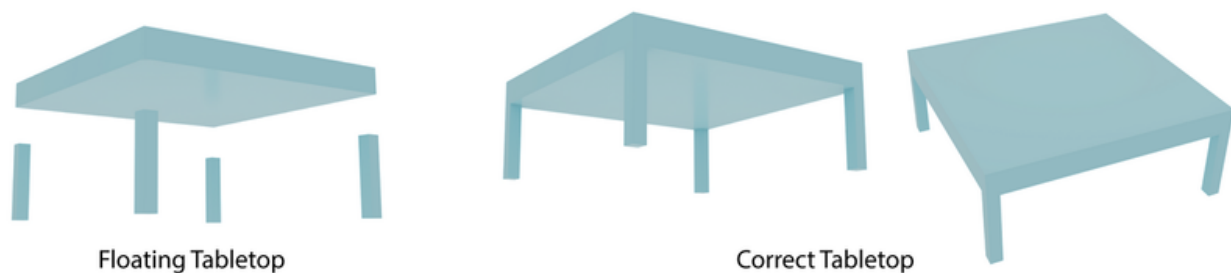


Figure B7. Failed and successful cases of table creation using GPT-4. The table consists of five parts: four legs and a tabletop. Although GPT-4 successfully gives a correct composition of the table, GPT-4 outputs a floating tabletop without any human intervention.

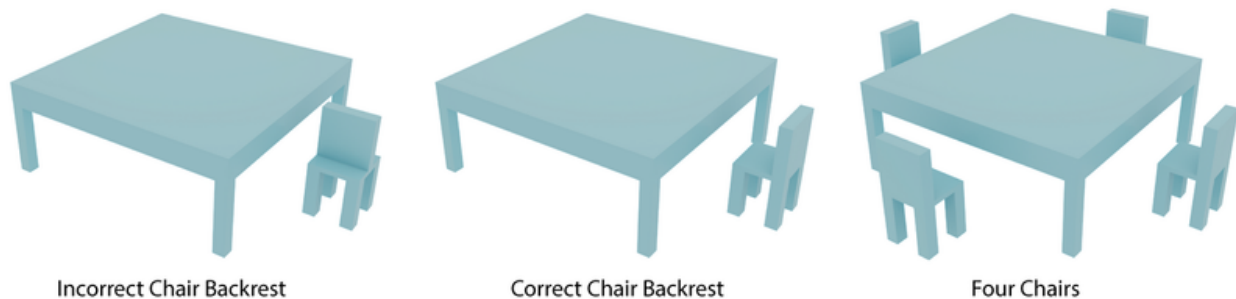


Figure B8. Failed and Successful Cases of Chair Creation using GPT-4. GPT-4 successfully gives a correct composition of the chair. In the incorrect version (left), the dimension of the backseat is wrong and it looks like the orientation is wrong.

B.3. CSG With PyVista

Building on GPT-4’s success generating CSG-like models with boxes, we set out to explore GPT-4’s capacity to use a larger suite of primitives. For this, we used an existing 3D visualization library, PyVista, which allows us to create and place a variety of 3D primitives such as spheres and cones. Thanks to the library’s documentation, GPT-4 is able to automatically assemble a functional Python program using PyVista’s primitive functions.

We asked GPT-4 to use PyVista’s primitives to model several variations of a fish, including specific bio-inspirations such as goldfish, a manta ray, and a loach ([Figure B9](#)). GPT-4 successfully selected and scaled an

appropriate set of primitives for each example, and provided sound bio-inspired rationale for its decisions. In particular, although most of the fish are composed using a sphere for the body, GPT-4 intuitively that a loach would be most effectively approximated by using two cones for the body to give it an elongated shape .

One area in which GPT-4 struggled was the determination of the primitives' orientations. It often produced results that indicated an internal confusion of some of the axes, or an otherwise flawed approximation of the orientation that would be required to achieve a desired effect. After engaging in a dialogue with GPT-4, it was able to rectify the orientations of the primitives to more closely resemble the target creatures. While promising, these tests reiterate GPT-4's seemingly limited capacity to account for local coordinate frames.

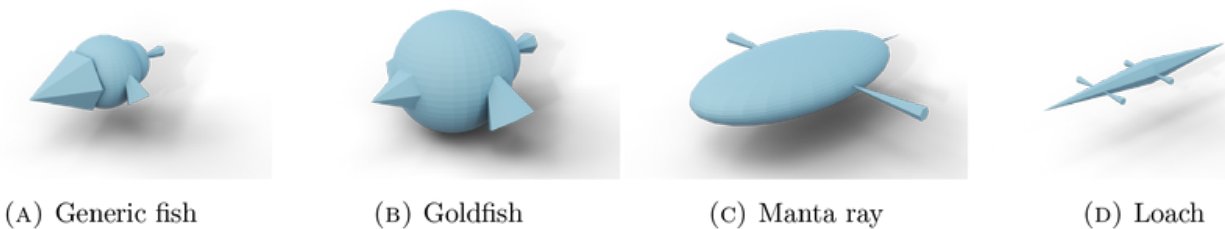


Figure B9. Aquatic creatures generated by GPT-4. GPT-4 successfully generated variations of aquatic creatures automatically using primitives from the PyVista package in Python.

B.4. Sketch-Based CAD With Onshape

Another popular method for 3D shape modeling comes from contemporary computer-aided design (CAD) software. Rather than directly constructing and modifying solid primitives (as in the CSG approaches discussed above), modern parametric CAD systems generally work by lifting planar sketches into 3D and subsequently modifying the 3D geometry. These sketches are placed on planes, which can be offsetted construction planes, or planar faces of the current 3D model. The selected sketching plane serves as a local coordinate system in which the sketch primitives are defined. In graphical user interfaces, this change of coordinate systems is accounted for by letting the user easily align their camera view to a top-down view onto the sketch plane. This change of view effectively comes back to drawing sketches in 2D, removing the cognitive burden of having to think about sketches in 3D. Despite the lack of graphical assistance, we want to investigate whether GPT-4 is able to design objects using a sketch-based modeling language.

However, since the graphical assistance is very prevalent in this modeling paradigm, CAD models are mostly constructed via a graphical user interface (GUI) and not via textual programming, even though textual APIs exist, for example, Onshape's Featurescript (["FeatureScript Introduction," n.d.](#)). Therefore, documentation and examples are less available than for the modeling paradigms from the previous sections. And indeed, GPT-4 performs poorly when trying to generate Featurescript code directly, which is why we decided to provide a simplified domain-specific language (DSL).

For our experiments, we constructed a single prompt containing the following DSL description: Our DSL exposes two operators, `createSketch` and `extrude`, and two sketch primitives, `circle` and `rectangle`. Additionally, we provide a construction example using this language of a single leg round table. Lastly, we also add some hints about how to write the program, for example, to explicitly use design variables and to write in syntactically correct python. All of the output designs generated by GPT-4 in this section are automatically translated into Onshape PartStudios. The full prompt can be found in the supplemental material.

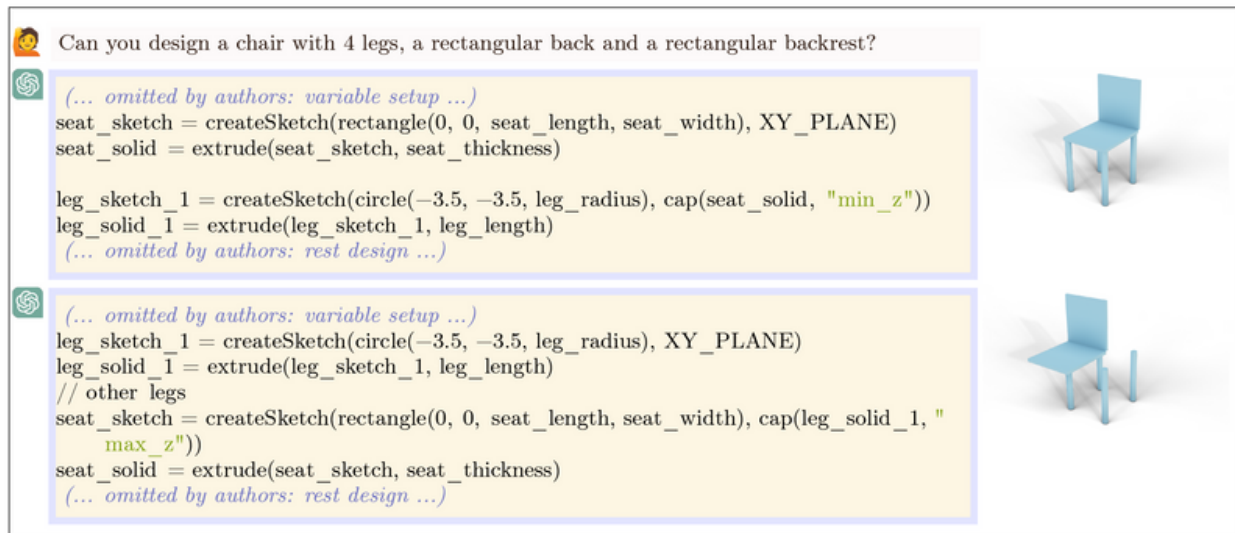


Figure B10. Different chair design strategies. GPT-4 designs a chair in two different ways. When starting with the seat, it does not have to change coordinate systems when specifying sketch coordinates of the legs. However, sketching a seat on the top face of one of the legs requires a coordinate change which GPT-4 does not manage to do.

Our first task is the design of a chair with 4 legs, a rectangular seat and a rectangular back, see [Figure B10](#). We asked GPT-4 to perform this task several times and observed the following.

- The design sometimes includes cylindrical legs, sometimes rectangular legs.
- The design is always constructed in a single direction, the Z direction. Our input example of the round table only used the Z direction to select sketching planes, but the description of our language documented the use of other plane directions.
- We observe mainly two types of designs: (i) designs that are constructed in both the negative and positive Z direction starting from the seat, see first answer in [Figure B10](#), and (ii) designs that start from a leg, see the second response in [Figure B10](#). We observe that the first type of designs has a higher chance of being correct, whereas the second type fails more often. The failures are due to changes in the coordinate system. For example, when selecting the top plane of the first leg as a sketch plane for the seat, the sketch plane's origin will be in the center of the leg. GPT-4 will often ignore this or will not be able to account for it when pointed out. Conversely, when starting with the seat and choosing the lower seat plane as a sketch plane for

the legs, it can specify the leg sketch coordinates in global coordinates, since the global origin coincides with the seat's origin. The same is true for the backrest.

From this test, we can observe that GPT-4 seems to have difficulties translating the coordinate system's origin on the XY plane.

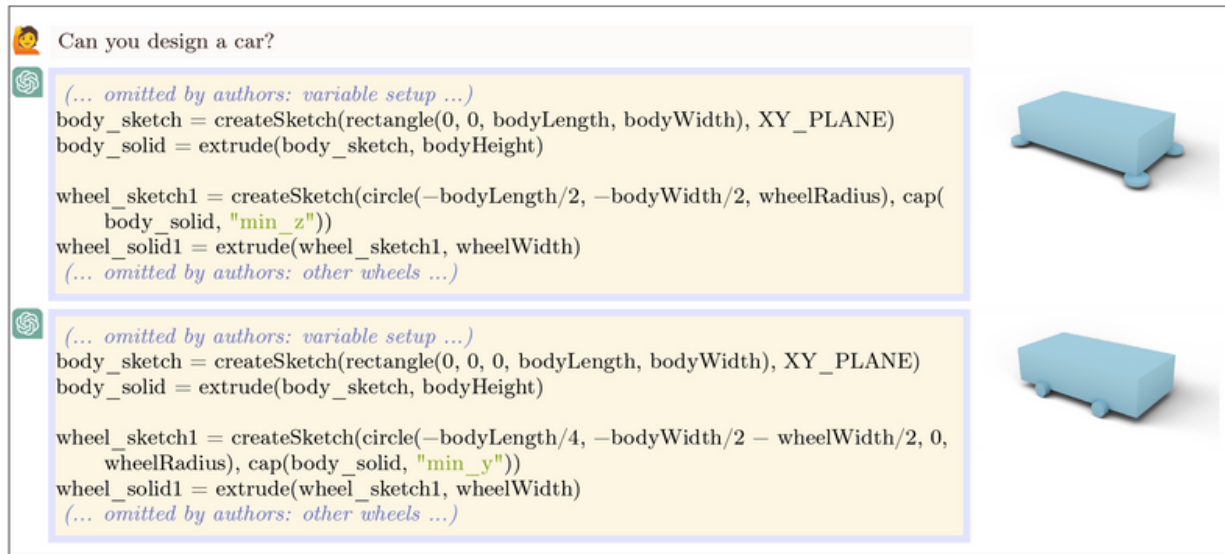


Figure B11. Local vs. global coordinate systems. GPT-4 does not manage to rotate the wheel sketch planes. If we allow it to express sketch primitives in global coordinates and select a plane to orient the extrude operation, it places them correctly.

Next, we want to see if GPT-4 can account for rotating sketch planes. To test this, we ask it to design a car. GPT-4 always suggests a simple car shape, composed out of four cylindrical wheels and a rectangular car body, see [Figure B11](#). The difficulty with this shape is that the cylinder sketches of the wheels have to be extruded on the side planes of the car body. There are a couple different modeling strategies to achieve this, but we observe that GPT-4 has difficulties coming up with these designs without any further indication. Instead, it often extrudes the car body along its `height`, starting from the ground plane, and then places the wheel circles on the bottom plane of the car, which is also the ground plane. This has the effect that the car wheels will be extruded vertically. Although we were able to correct this design in an iterative prompt-based fashion, we had little success engineering the initial prompt in such a way that we could effectively prevent this behavior.

Note that intuitively placing wheels at the bottom of a car body makes sense and that without any graphical feedback, humans could also easily make this mistake. From this test, we can observe that GPT-4 is struggling to rotationally change coordinate systems.

To address this, we changed our design language description to allow GPT-4 to specify sketch primitive coordinates directly in a single global coordinate system. Now, a sketch primitive center takes as input three coordinates, which we project in postprocessing directly on the selected sketch plane. The extrude direction is

still defined by the sketch plane's normal vector. This means that GPT-4 does not have to take coordinate translations into account anymore. We observe that this change in the DSL led to a higher success rate in generated designs, see second answer in [Figure B11](#).

In conclusion, GPT-4 is able to design models in a sketch-based parametric CAD framework. However it is not successful at changing coordinate systems. In this case, our backup strategy is to use a single global coordinate system. One possible future direction is to let GPT-4 communicate with a geometric solver and create a feedback loop.

B.5. URDF

The Universal Robot Description Format (URDF) is a common XML-based language for describing articulated structures in robotics. URDF files specify a robot's structure (including both visual and collision geometry), joint locations, and dynamics information. The URDF format appears well-suited for potential LLM design because it is human-readable and heavily documented online.

B.5.1. Open Chain Robot Arms



Generate a urdf for a two link robot. The end of the second link should connect to the end of the first link with a revolute joint. The origin of the joint should be a the end of the first link



```
<joint name="joint1" type="revolute">
  <axis xyz="0 0 1" />
  <parent link="link1" /><child link="link2" />
  <origin xyz="1 0 0" rpy="0 0 0" />
</joint>

<link name="link1"><visual><geometry><box size="0.1 1 0.1" /></geometry></visual></link>
<link name="link2"><visual><geometry><box size="0.1 1 0.1" /></geometry></visual></link>
```

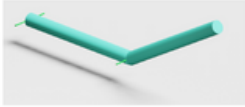




urdf link origins are in the center of the link. To make a joint origin at the end of a link, it must be half the link's length past the link origin. Generate a urdf for a 2 link robot. The end of the second link should connect to the end of the first link with a revolute joint. The origin of the joint should be at the end of the first link. Both links should be lined up along the z axis.



(... omitted by authors: URDF description of an appropriately connected two-link robot ...)





urdf link origins are in the center of the link. To make a joint origin at the end of a link, it must be half the link's length past the link origin. Generate a urdf for a three link robot. The end of the second link should connect to the end of the first link with a revolute joint about the x axis, the origin of the first joint should be a the end of the first link. The other end of the second link should connect to the end of the third link via a revolute joint about the x axis with an origin at the end of the third link. All three links should be lined up along the z axis



(... omitted by authors: URDF description of an appropriately connected three-link robot ...)

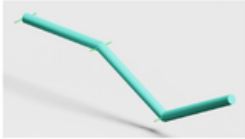


Figure B12. GPT-4's process for designing N-link robot arms. GPT-4 is unable to correctly interpret the initial high-level specification, but when provided with more explicit detail, GPT-4 is able to generate correct Universal Robot Description Format (URDF) specifications for N-link robots.

Initially, we asked GPT-4 to generate simple open chain robots (commonly called 'arms') with a particular number of links. However, when we used the word 'arm' to prompt GPT-4 to generate a robot, GPT-4 was unable to determine that the links should connect at the end. Most often, GPT-4 placed the joints such that each link revolved about its center, and the links were not connected to each other (Figure B12, initial prompt). As shown in the subsequent prompts of Figure B12, to achieve an arm with two connected links, it was necessary to describe both the joint position relative to the link ('the joint origin must be half the link's length past the link origin', rather than 'the joint origin should be at the end of the link') as well as the joint axis('a revolute joint about the x axis'). Given this prompt pattern, GPT-4 was easily able to generate proper N-link robots.

B.5.2. Wheeled Robots

Next, we asked GPT-4 to generate wheeled robots composed of N wheels attached to a central rectangular platform. A proper design of this type must have wheels that (1) are aligned to share an axis of rotation normal to and through the center of their circular faces; (2) have circular faces displaced along said axis of rotation, and (3) contact, but do not intersect, either side of the center platform. The combination of non-intersection and geometry relation constraints prove challenging for GPT-4, which seems to exhibit limited geometric reasoning. Initially, we tried to specify these using language-based constraints (i.e., “the wheels should touch, but not intersect, either side of the platform”). These proved ineffective, as shown in [Figure B13](#) (middle). To overcome these challenges, we crafted prompts with very explicit numeric constraints (i.e., “wheels should be offset on the global y axis by half the width of the platform plus half the height of the wheel cylinder”). This style of prompt successfully generated a viable result, as shown in [Figure B13](#) (right).

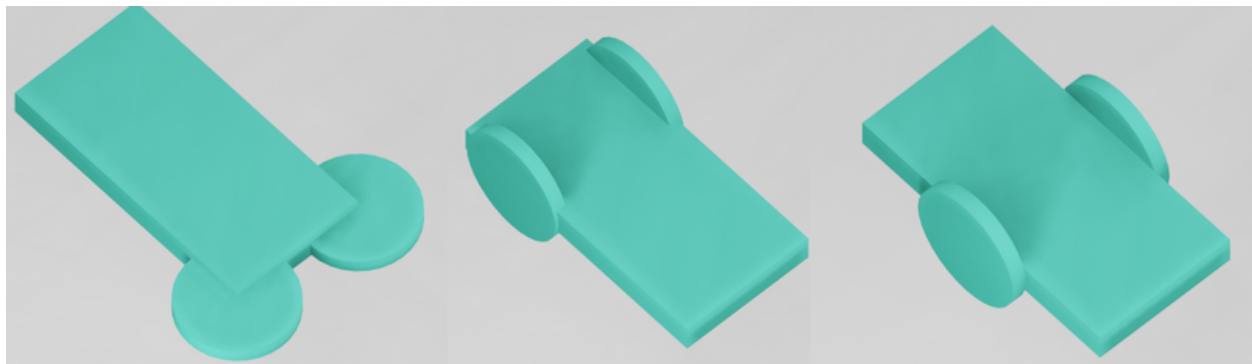


Figure B13. Two-wheeled robot attempts by GPT-4. GPT-4’s attempts to generate a two-wheeled robot with varying types of prompt constraints on geometry. **Left:** no constraints. **Center:** language constraints. **Right:** numeric constraints.

As in the case of robot arms, we find that GPT-4 is immediately able to generalize a successful two-wheeled design into a four-wheeled robot. We achieve this by asking for a duplicate, shifted version of the existing wheel configuration, as shown in [Figure B14](#). However, we were unable to directly generate a successful four-wheel robot; in general, we found that as the number of constraints in a prompt increases, it becomes increasingly likely that GPT-4 will ignore any individual constraint. Thus, rather than directly requesting a four-wheeled robot in a single prompt, we found greater success by first generating a two-wheeled robot and then prompting GPT-4 to modify the URDF by adding additional wheels than placing the text in a single prompt.

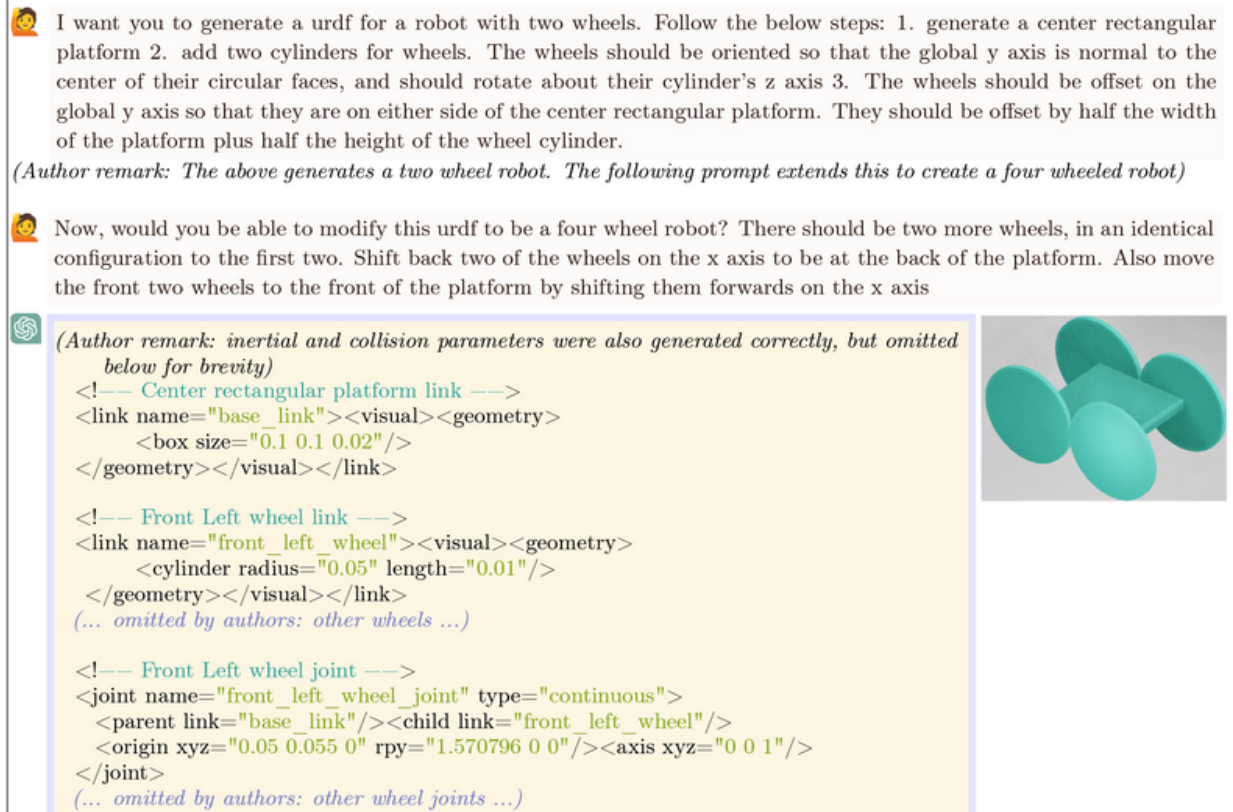


Figure B14. Universal Robot Description Format (URDF) of four-wheeled robot. A correct four-wheeled robot generated by GPT-4 by modifying a previously generated URDF for a two-wheeled robot

B.5.3. Robot Grippers

To test the effectiveness of our iterative, multiprompt approach for building robots of increasing complexity, we seeded GPT-4 with a successful two-link open chain URDF, then asked it to modify this design into a collection of multifinger robot grippers. As shown in [Figure B15](#), we were able to build two-, four-, and five-finger grippers using a sequence of prompts to add features and change proportions. To create a two-finger gripper, we asked GPT-4 to use two of the previously generated two-link open chain robots as fingers, separated by a distance equal to half the height of the finger, and connected by a rectangular platform on the base. The four-finger gripper was similarly derived from the two-link arm by specifying that the hand should consist of four two-link robots right next to each other on a rectangular platform. To specify a five-finger hand, we requested a rectangular link that hinges as a base for the thumb, then prompted GPT-4 to add another finger on that link and to adjust the hand proportions.

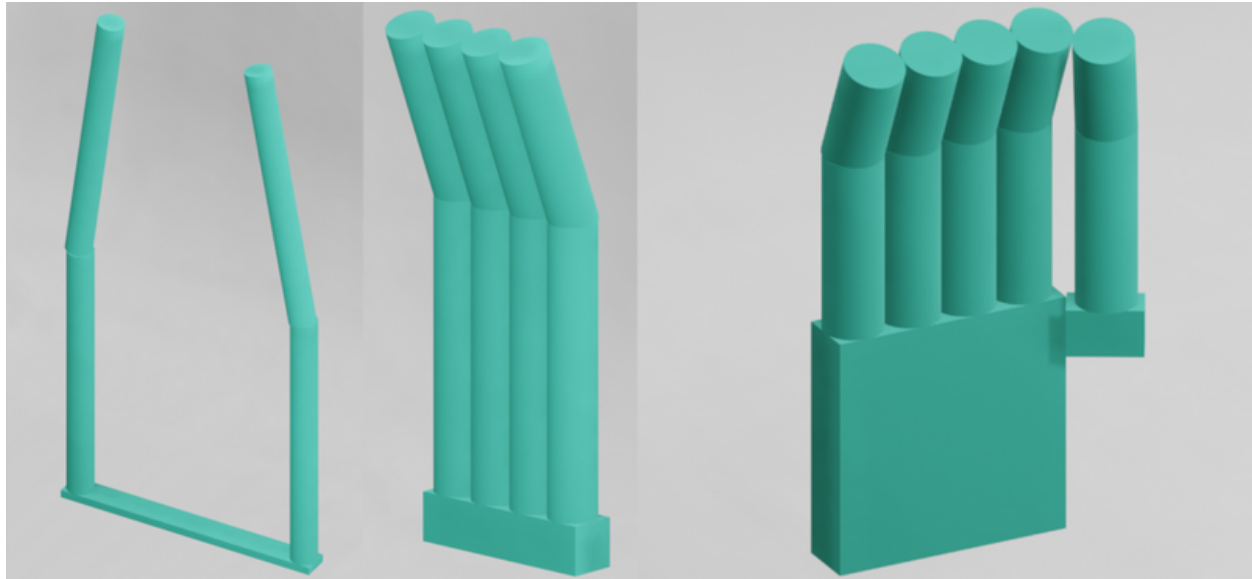


Figure B15. Universal Robot Description Format grippers generated by GPT-4. Left: Two fingered gripper. **Center:** Four fingered gripper. **Right:** Five Finger Hand

B.6. Graph-Based DSL

While designing an entire robot end-to-end using LLMs may not be feasible, we find that GPT-4 has the ability to reason about the spatial layout of robot components. These spatial layouts are naturally represented as graphs where the nodes are components and edges are connections between them. Unlike URDF, this representation is more general and is applicable in domains outside of simulation.

To generate robot design graphs using GPT-4, we first need a text-based graph representation. Our first approach involved asking GPT-4 to output the popular Graphviz format. While convenient, this format makes it difficult for GPT-4 to provide metadata for each part (such as motor torque, size) in a format usable by downstream applications. Instead, we take advantage of GPT-4’s ability to generate Python code that conforms to a provided domain-specific language. The full DSL is detailed in [Appendix A.4](#).

When prompted with a small DSL embedded in Python, GPT-4 is able to write code that selects and places robot components at a high level of abstraction. By supplying a function that translates components in three-dimensional space, we can extract GPT-4’s concept of each component’s position relative to the others.

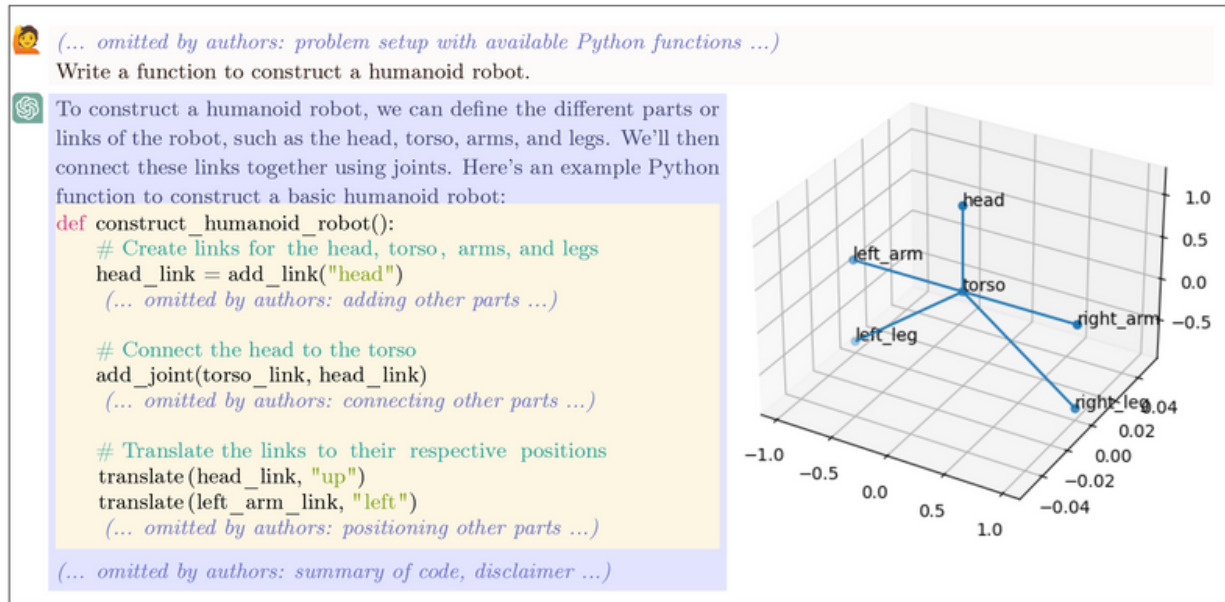


Figure B16. Graph of humanoid robot. Graph generated by GPT-4 describing the high-level components of a humanoid robot as nodes and connections between them as edges

In this example (Figure B16), we ask GPT-4 to generate a humanoid robot using the provided functions. GPT-4 makes appropriate calls to `add_link` to create nodes in the design graph, `add_joint` to create edges between them, and `translate` to establish their relative positions.

We manually implement the functions described in the prompt in order to visualize the resulting robot topology. The arms are positioned beside the torso, the legs are positioned below, and the head rests on top as expected for a humanoid robot.

We saw similar success when asking GPT-4 to construct a snake robot, car robot, and scorpion robot. When requesting a robot dog, however, GPT-4 only adds two legs initially. Specifying a “robot dog with four legs” was necessary to obtain the expected behavior. We also encountered difficulties when attempting to obtain a more detailed design for the robot dog. Asking for a “robot dog with four legs, two links per leg” produced a graph with two nodes per leg, but GPT-4 did not position them relative to each other.

Appendix C. Additional Experiments in Creating Design Spaces via Natural Language and Extrapolation

C.1. Generating a Design Space from Text (Q1)

As noted, we notice that when asking for a simple design and asking for a parametric design of the same object, there are generally fewer mistakes in the reuse of certain dimensions. For example, in Figure C17, at first, GPT-4 positions the backrest on top of the seat using the correct numerical values, but not for the correct dimensions.

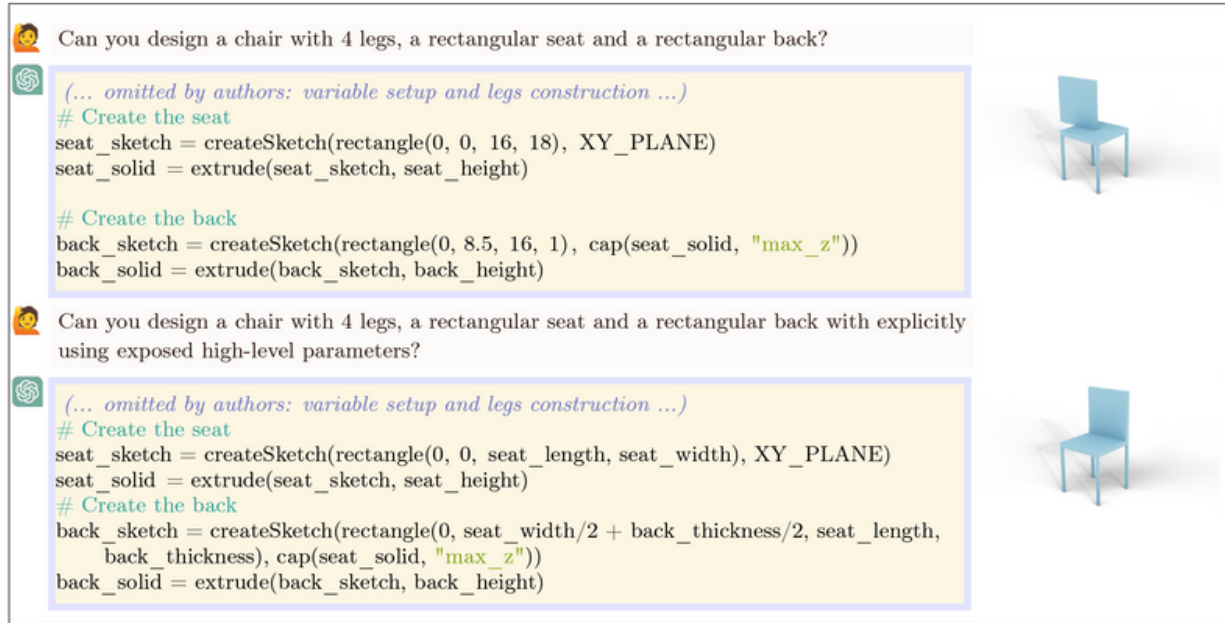


Figure C17. Advantage of parametric modeling. Without using explicit variables, GPT-4 does the computation for sketch coordinates by itself and is more likely to confuse variables and to make mistakes.

Whereas when asked for a parametric design, the use of `width` and `length` suffixes in the parameter names seem to be more consistently associated with the corresponding 3D axis.

C.1.1. Generate a Design Space From an Existing Design (Q2)

As described previously, when given a design with no semantic context, we observe that GPT-4 exposes design parameters based on equivalence between numerical values and based on which design operators these values were used in. For example, in [Figure C18](#), it introduces a variable `cube_size` that replaces the value `19` that was used for both the chair's width and length. For the mug in [Figure C19](#), we can observe that the exposed variables also stay close to their original usage for a given geometric operator.

Second, we repeat the previous experiment with additional semantic context. Providing GPT-4 with the name of the object that is being modeled proves useful for generating a parametric design. We can see that now, design parameters get exposed that are semantically more useful for modifying the design. For example, the cylinder radii in [Figure C19](#) gets replaced for a parameter `mug_wall_thickness`, which controls the thickness of the mug by considering both radii jointly. Also, some ambiguity caused by numerical equivalence can be resolved and produce more useful parametrizations. In [Figure C18](#), the `cube_size` from the previous parametrization without any semantic context, gets disentangled into a `length` and a `width` parameter, allowing to have more control over the shape. This might prove especially useful in this case, since all the slats are associated to the chair's width and not its length.

 Can you expose high-level design parameters for this openscad design? Use as few variables as possible

```
union() {
  difference () {
    cube([19, 19, 15 + 2]);
    (... omitted by authors: intermediate construction operations ...)
    for (i = [0:5-1])
    (... omitted by authors: rest design ...)
```

 *// High-level design parameters*
cube_size = 19;
cube_height = 15;
extra_height = 2;
num_layers = 5;
// Other parameters ...
union() {
 difference () {
 cube([cube_size, cube_size, cube_height + extra_height]);
 (... omitted by authors: intermediate construction operations ...)
 for (i = [0:num_layers-1])
 (... omitted by authors: rest design ...)

 Can you expose high-level design parameters for this openscad design of a chair with slats? Use as few variables as possible

 length = 19; *// Length of the chair*
width = 19; *// Width of the chair*
height = 15; *// Height of the chair base*
slat_count = 5; *// Number of slats*
union() {
 difference () {
 cube([length, width, height + thickness]);
 (... omitted by authors: intermediate construction operations ...)
 for (i = [0:slat_count-1])
 (... omitted by authors: rest design ...)



Figure C18. Semantic cues for chair design parameters. When asked to parametrize a chair design, it GPT-4 gives different parametrization with or without context cue.

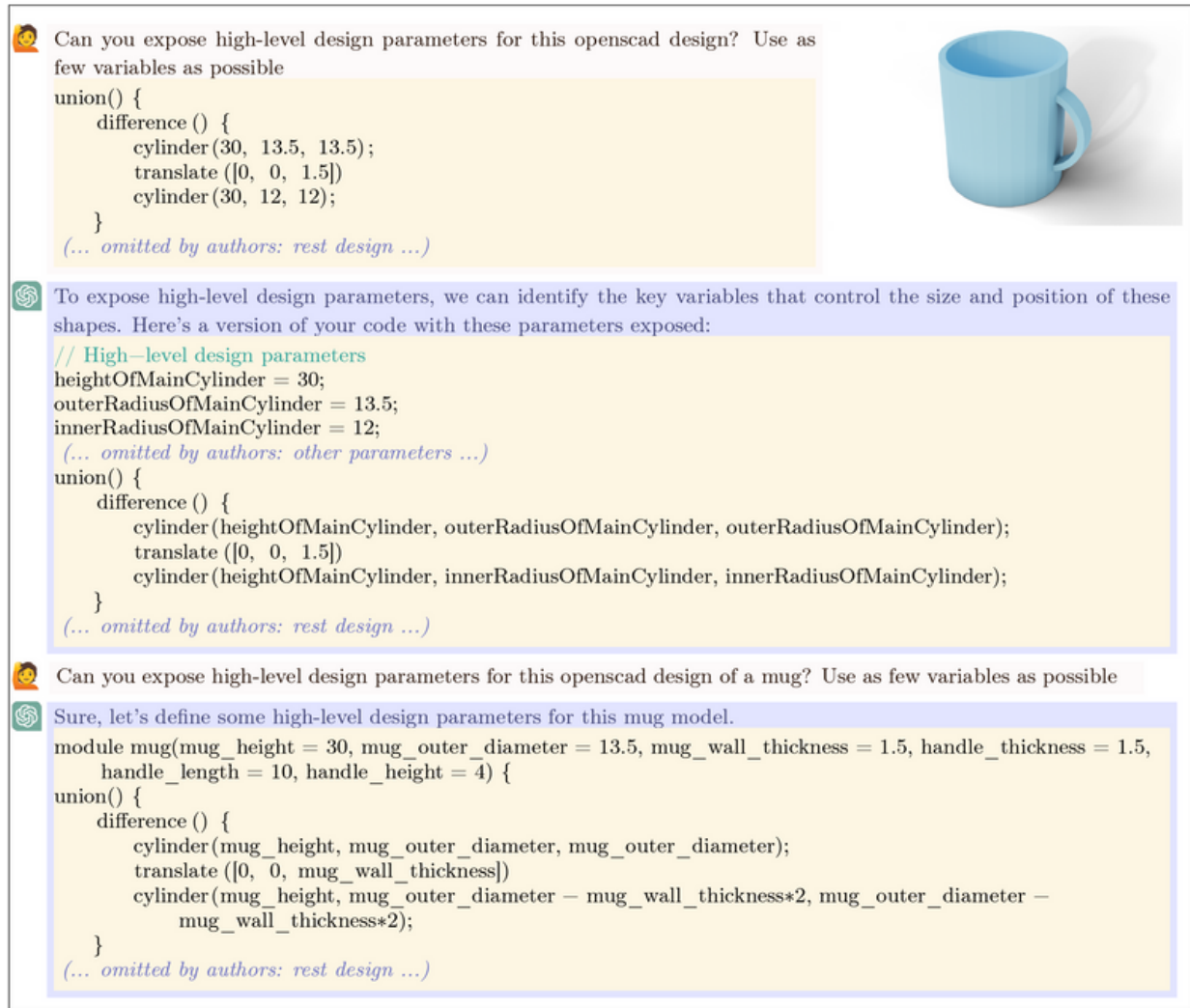


Figure C19. Semantic cues for mug design parameters. When asked to parametrize a mug design, it GPT-4 gives different parametrization with or without context cue.

Once parametrized, we can complete the design space by asking for parameter bounds, see [Figure C20](#). Again, notice how these bounds are `somewhat arbitrary` and not based on the 3D design sequence.

While these results are encouraging, GPT-4 is easily confused by the final effect of a series of geometric transformations. An example for this is the generated parameter `handle_thickness` in Figure C19, which actually modifies the *y* position of the handle. Once again, it is limited by cases where geometric computation prevails over semantic reasoning.

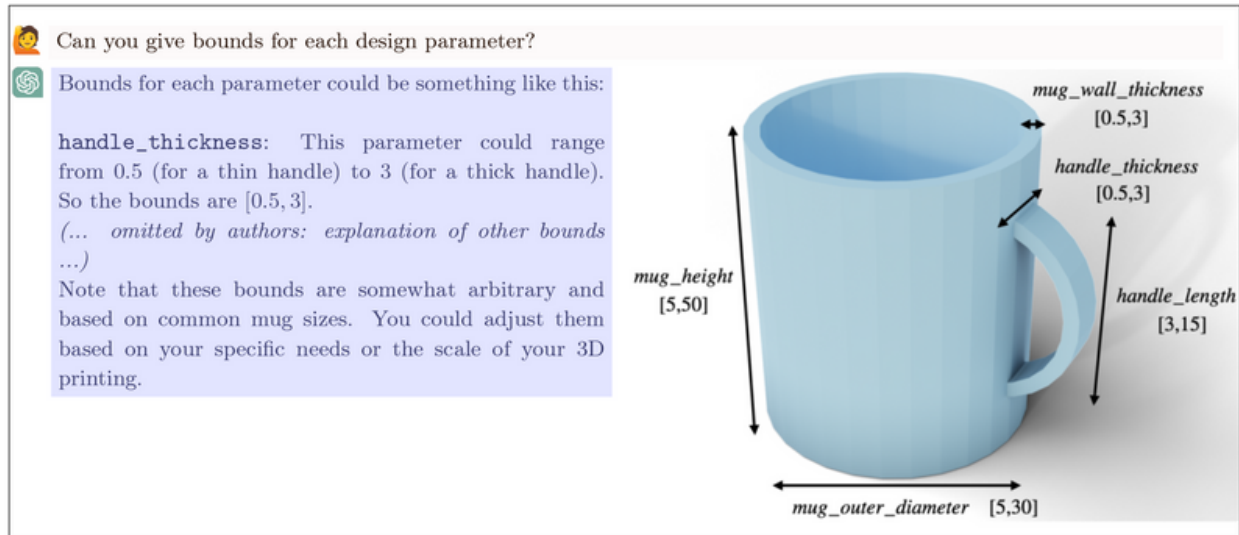


Figure C20. Mug design parameter bounds and constraints. GPT-4 gives parameter bounds based on common mug sizes.

C.1.2. Can GPT-4 Create a Design Space From Multiple Designs? (Q3)



Figure C21. Chair interpolation. Left: Input chair design with cylindrical legs. **Middle:** Input chair design with splats. **Right:** Interpolated output chair design with cylindrical legs and splats.

To investigate if GPT-4 can help with design interpolation, we tested three different design scenarios. All of the designs were presented to GPT-4 in our sketch-based parametric CAD DSL, explained in [Section 6](#).

First, we present GPT-4 with two chairs which are modeled similarly, but the first chair has cylindrical legs and the second chair has a backrest with splats, see [Figure C21](#). In our prompt, we ask if it can mix these two designs to create a chair with cylindrical legs and splats in the back. The result can be seen in [Figure C21\(c\)](#). It should be noted that variables in the code are descriptive, for example `leg4_solid` and `splat_3_sketch`, which helps provide semantic cues. Also, in our designs, the first half of the code describes the construction of the seat and the legs and the second part describes the construction of the backrest. This

means that mixing these two designs comes down to replacing the second half of the first design with the second half of the second design.

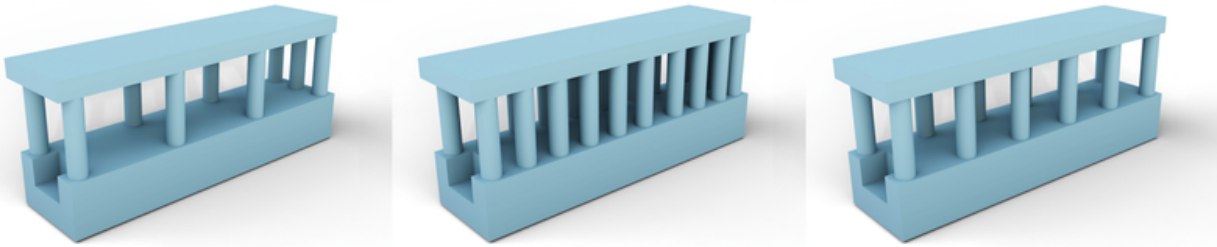


Figure C22. Temple Design Interpolation. **Left:** Input temple design with 4 pillars on each side. **Middle:** Input temple design with 10 pillars on each side. **Right:** Interpolated output temple design with 6 pillars on each side.

Next, we present GPT-4 with two designs of a temple, involving a different number of pillars, one with 4 pillars and one with 10 pillars on each side, see [Figure C22](#). In our prompt, we ask it to `design a temple with steps, a roof and 6 pillars on the left and right side`. For this, GPT-4 has to find how these pillars have been modeled and how to model a varying number of pillars, given the two input examples. The code for the design of the pillars did not contain any looping structures nor variables and it was more spread out throughout the program than in the chair example, to make it more challenging. Despite these challenges, GPT-4 manages to extract the construction logic of the pillars and introduces variables and a looping structure to place them correctly, see [Figure C22\(c\)](#). Note that we have mentioned the steps and the roof in the prompt. We have noticed that without this reminder, it would solely focus on the construction of the pillars and forget about the rest of the design.

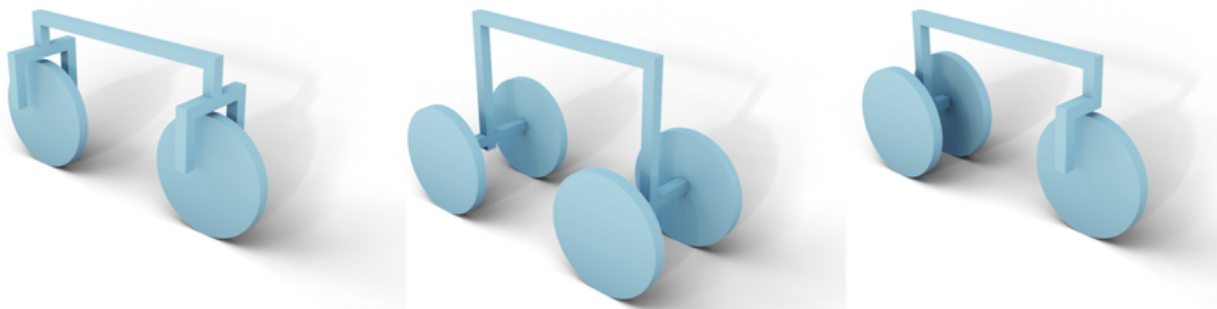


Figure C23. Bike Design Interpolation **Left:** Input bicycle design . **Middle:** Input quad-bike design. **Right:** Interpolated output tricycle design.

Our last test is structurally more challenging. We present GPT-4 with a design of a bicycle and a design of a quad-bike, see [Figure C23](#). The two designs differ not only by the number of wheels in the front and the back, but also by the construction of the bike forks. In the case of the bicycle, the fork surrounds the wheel and in the case of the quad-bike, the wheels are connected by a horizontal bar to the vertical bar of the frame. This makes the mixing of sub-designs more complex. And indeed, when asked to design a tricycle, GPT-4 reasons

correctly about the number of wheels in the front and the back, and where to find these structures. It also adjusts the size of the quad-bike's vertical bar such that the two back wheels and the front wheel are on the same plane. This was not the case for the quad-bike and the bicycle in the input designs. But it does not succeed at extracting the complete fork from the bicycle design, as can be seen in [Figure C23\(c\)](#). Note that this experiment was performed via a single prompt and GPT-4 would likely be able to copy the missing part via further interaction with the user.

C.1.3. Exploration of a Given Design Space (Q4)

As noted, a design space is conceptually useful to reliably generate variations of a given design. However, coming up with parameters that represent *meaningful* design variations can be a time-consuming iterative process.

To investigate if GPT-4 can help with this task, we perform the following experiment. We present it with a parametric design of a Lego brick, see [Figure C24](#). Then, we ask it to generate `parameter bounds` and `parameter constraints`. Interestingly, GPT-4 generated the nontrivial constraint that the length and width of the brick should be multiples of 3. We ask it to use the design space to `come up with 10 different parameter settings which correspond to meaningful lego bricks`. Finally, it should `give each variation a name`, see [Figure C25](#).

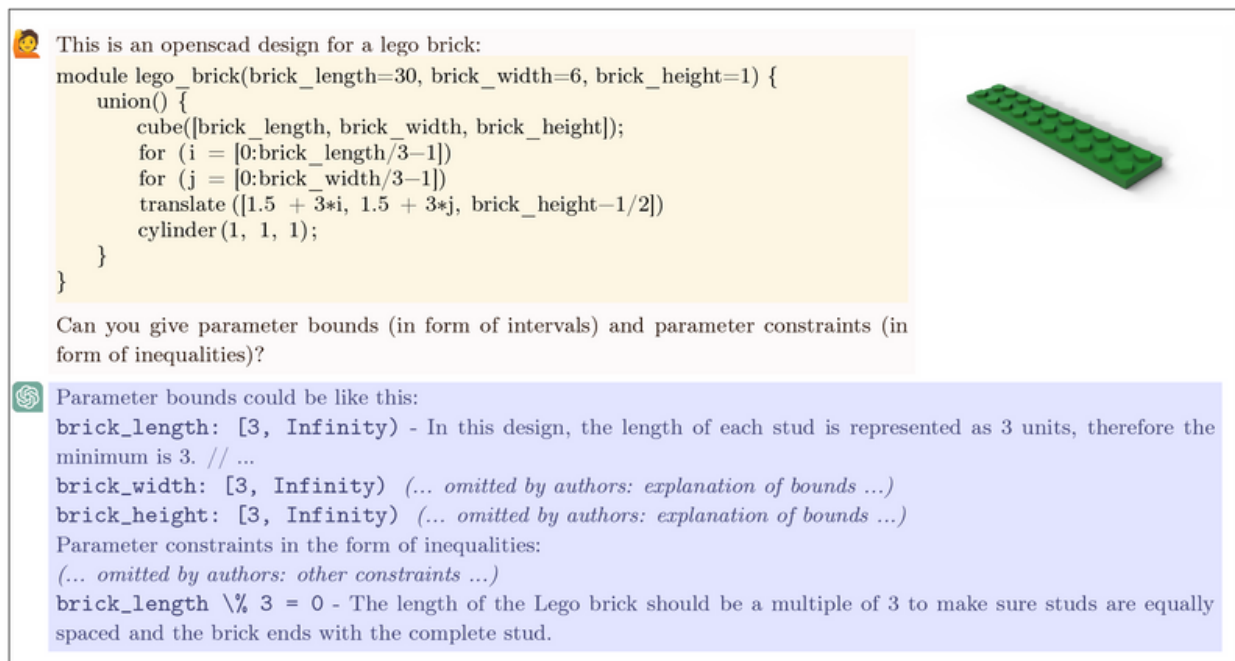


Figure C24. Lego Design Space. We ask GPT-4 if it can generate parameter bounds and constraints for a parametric Lego brick design.

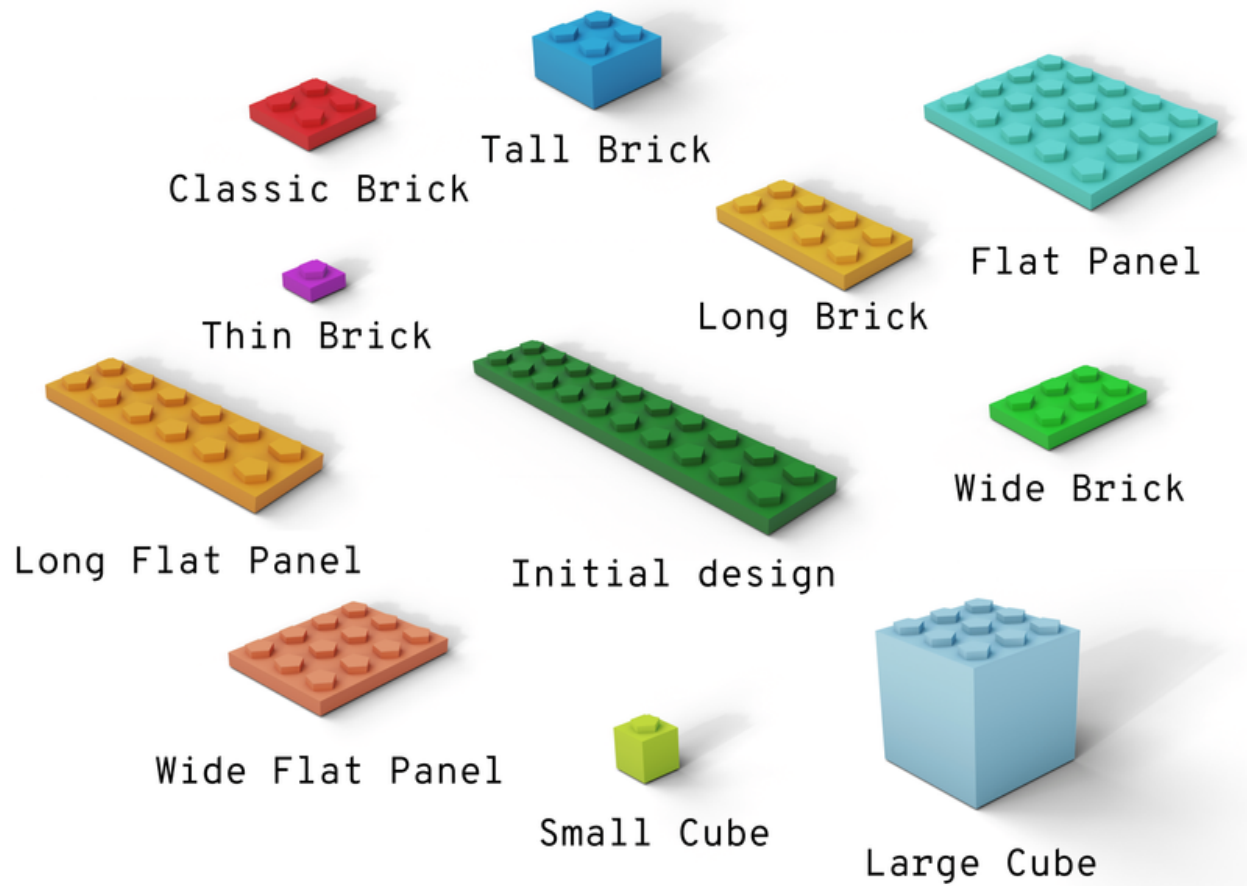


Figure C25. Lego design space exploration. GPT-4 generates 10 different design variations for the parametric `Initial design`. The label under each model corresponds to the name given by GPT-4.

We can observe that the proposed parameter settings respect the previously generated bounds and constraints and that they lead to distinct 3D models, for which it generates plausible semantic labels.

Appendix D. Additional Design For Manufacturing Experiments

D.1. Finding Optimal Manufacturing Process (Q1)

To test GPT-4's capabilities in manufacturing for design, we tasked GPT-4 with advising on identifying an optimal manufacturing process for a part with the geometry shown in [Figure D26](#). We tested it with four different cases where in each case the geometry, material, tolerance requirements, and quantity were varied. We described the part's geometry as an OpenJSCAD file. Finally, given a set of priorities, we task GPT-4 to select an optimal manufacturing process. In case four, we provided a finite list of manufacturing processes to evaluate the effectiveness of the selection process under the constraint of a limited set of options. The goal was to determine how well the process could choose the appropriate manufacturing processes to meet the specified priorities.

GPT-4 was successful at selecting an optimal manufacturing process for three out the four cases. For cases one, two, and four, GPT-4 selected the optimal process that was approved by an expert. However, in case three, shown in [Figure D27](#), GPT-4 suggested an injection molding process, which is not suitable for processing a polytetrafluoroethylene (PTFE) material. In all cases, GPT-4 initially only provided a range of manufacturing options; it required additional prompts to arrive at the optimal manufacturing process selection.

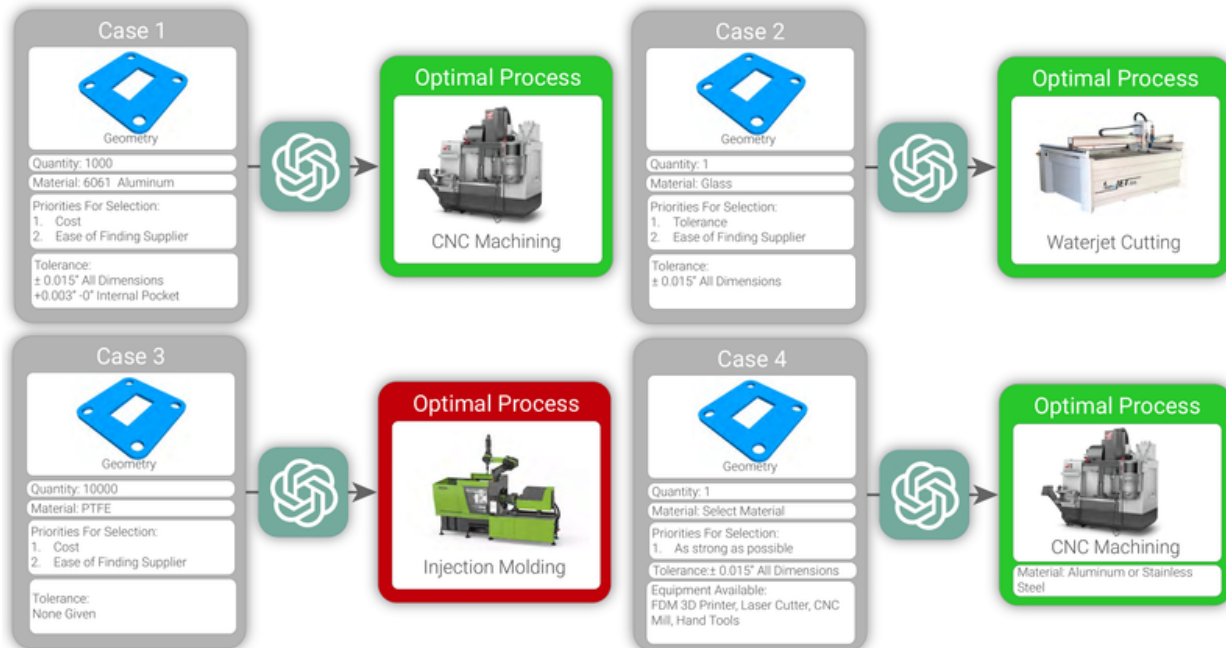


Figure D26. Using GPT-4 to select optimal manufacturing process: Inputs and selected optimal manufacturing processes using GPT-4.

... You are trying to select an optimal set of manufacturing process to make the part.

...The parts must be manufactured to the tolerance of plus or minus 0.015 inches.

You need to make 10000 part out of PTFE.

... priorities for selecting a manufacturing process are the following: 1. Cost 2. Ease of finding multiple suppliers.

Given the description of the part provided, the best method to produce this component from PTFE (Polytetrafluoroethylene) in high volume, with a tight tolerance of ± 0.015 inches would be injection molding.

... Regarding the material choice, PTFE is a common material for injection molding due to its high melting point, excellent chemical resistance, and excellent electrical insulating properties. However, it's worth noting that PTFE can be challenging to mold due to its high melt viscosity and tendency to warp as it cools. Proper mold design and processing conditions are critical to achieving the desired part dimensions and properties.

...



Figure D27. Prompt and response for selecting an optimal manufacturing process for apolytetrafluoroethylene (PTFE) part. GPT-4 gives the response that PTFE is commonly injection molded, which is incorrect. Typically, it is not suggested that PTFE be injection molded.

D.2. Design Alterations for Manufacturability (Q2)

Our focus in this case was on the computer numerical control (CNC) machining of a 10-inch diameter disk, which involved creating bolt holes along the edge and a central blind square pocket. We included in the geometry two intentional features that would be difficult to machine. As depicted in [Figure D28](#), the process began with GPT-4 identifying any manufacturing complexities within the design features. Since a large language model (LLM) interprets text, GPT-4 interprets the text of the OpenJSCAD file, rather than the geometry that is rendered once compiled, which humans interpret. After GPT-4 identifies any complexities, we instructed it to adjust the geometry of the OpenJSCAD file to address the challenging aspects by directly changing the text of the OpenJSCAD file.

Although GPT-4 accomplished these tasks with a moderate degree of success, there were a few inaccuracies. Firstly, GPT-4 correctly identified two potential machining issues: the small radius of the internal pocket and the thin wall at the pocket base. However, it also misunderstood a number of geometric features described in the OpenJSCAD file. These include perceiving holes on a curved surface and anticipating an undercut from the pocket. These misinterpretations might be attributed to GPT-4's reliance on the text of the OpenJSCAD file for feature identification, as some features become more visible once the file is compiled into a geometric representation. After pointing out these interpretation errors to GPT-4, it was able to correct its analysis but introduced another mistake. GPT-4 incorrectly stated that the bolt holes presented machining difficulties and inquired about additional information regarding the machining area. Once provided with the necessary details, GPT-4 independently rectified its mistake about the bolt holes. GPT-4 was also aware of potential issues with the size of the part and machining area of the CNC machine. Furthermore, it was able to compute whether there was a potential issue.

In the final stage, GPT-4 was asked to modify the OpenJSCAD file to address the manufacturing concerns. It improved the wall thickness from 0.02" to 0.04", making it machinable. Given the additional specification of utilizing a 1/4" endmill, GPT-4 also adeptly adjusted the internal pocket's radius to accommodate this tooling requirement better.

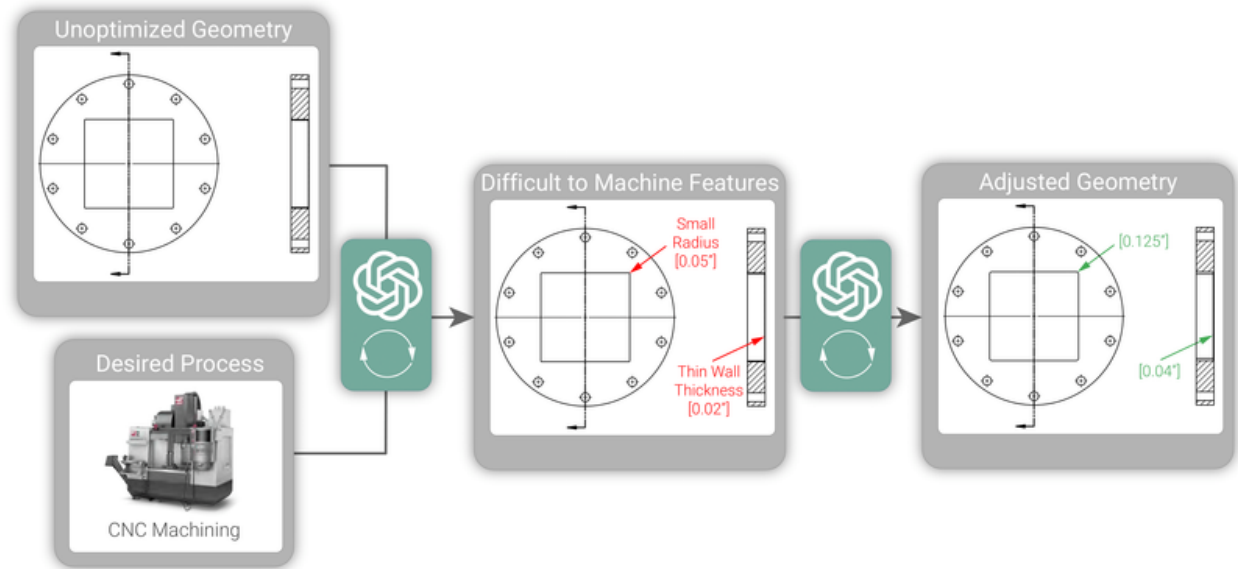


Figure D28. Using an LLM to find difficult to manufacture features and directly adjust the geometry. GPT-4 is given the task to identify any features that are difficult to machine along with an OpenJSCAD file. After a number of iterations, it identifies a set of features. We prompt it then to make changes to the OpenJSCAD file to reduce the machining difficulty.

D.3. Part Sourcing (Q3)

As described in the main text, McMaster-Carr is a deep compendium of knowledge for hardware parts, with geometric information and even CAD models available for many items. McMaster-Carr already has a ‘search by geometry’ feature, so we wanted to know if we could perform higher level searches that involve both context and geometry.

First, we tried describing specific scenarios and asking GPT-4 for search terms that would procure us the correct part. Asking for a nut to be used in a tight space without room for a wrench and submerged in saltwater produced two appropriate results, “316 Stainless Steel Wing Nuts” and “316 Stainless Steel Knurled-Head Thumb Nuts,” where the correct form and material was identified. Asking for a tamper-proof nut also produced the correct search, “316 stainless steel tamper-resistant nut.” Next, we tried a more open-ended geometric compatibility scenario by asking for parts for an at-home carbonation system ([Figure D29](#)). We also then asked it for a comprehensive Bill of Materials. It seemed as though all parts were compatible, at least geometrically; we suspect this is because the items in the domain are standardized for compatibility, McMaster-Carr’s data set is quite rich, and there is great availability of each part across varying sizes.

 Hi ChatGPT. I'm trying to make an at-home carbonation system and I'd like to find the hose, fittings, and pressure gauge on McMaster-Carr. Can you help me? The jar I'm using has a 5 gallon capacity.

 Hello, I'm glad you've asked for help with this project! To create a DIY carbonation system, you'll need a few items. ... Here are some McMaster-Carr parts that could help:

Regulator: McMaster-Carr has a variety of regulators. An example is Part Number 5182K56 - a two-stage, brass, compressed gas regulator with CGA 320 inlet connection.

Pressure Relief Valve: For example, Part Number 48435K72, brass pressure-relief valve, set at 60 psi, which can be adjusted.

Tubing: Part Number 5648K35 can be an option. It's a firm tubing, clear, 1/4" ID, 3/8" OD.

CGA320 adapter for tubing: You'll need to adapt the CGA320 connection on the regulator to fit the tubing. Part Number 50785K51, a brass barbed tube fitting, can work for this.

Bulkhead fitting for the jar lid: For this, you can use Part Number 50785K191, a brass threaded-tube-through-wall fitting with a tight seal.

Diffusion stone: You can use a standard 0.5 micron diffusion stone. While this isn't available on McMaster-Carr, it can be found at various homebrew stores.

Remember to ensure all parts are compatible with each other (size, material, pressure rating) and appropriate for use with food and drink.



 Can you give me a bill of materials (BOM) for this project? ... Can you write it as a multi-level BOM, including all the assemblies, sub-assemblies, and be in this format: Assembly Level - Quantity - Manufacturing or Sourcing information - Part number - Name - Price

 Absolutely, here's a Bill of Materials (BOM) for your project. Please note that the exact quantities will depend on your specific setup and you might need to adjust as needed: (... content omitted by authors ...)

 Thank you! Can you make that slightly more condensed by removing the quantity number, price, and assembly level, since they appear redundant?

 Of course, here's a more condensed version:

Manufacturing/Sourcing Information: Self-assembled
Regulator Connection
Manufacturing/Sourcing Information: Self-assembled
Part 1: Sourcing Information: McMaster-Carr, Part Number: 5182K56, Name: Two-stage, brass, compressed gas regulator with CGA 320 inlet connection
Part 2: Sourcing Information: McMaster-Carr, Part Number: 50785K51, Name: Brass Barbed Tube Fitting for CGA320 connection
Part 3: Sourcing Information: McMaster-Carr, Part Number: 5648K35, Name: Firm Tubing, Clear, 1/4" ID, 3/8" OD
Jar Connection
Manufacturing/Sourcing Information: Self-assembled
Part 1: Sourcing Information: McMaster-Carr, Part Number: 48435K72, Name: Brass pressure-relief valve
Part 2: Sourcing Information: McMaster-Carr, Part Number: 50785K191, Name: Brass Threaded-Tube-Through-Wall Fitting
Diffusion Setup
Manufacturing/Sourcing Information: Self-assembled
Part 1: Sourcing Information: Homebrew store, Name: 0.5 micron diffusion stone

This BOM is simplified and omits quantities and prices, which should be determined according to your specific setup and checked directly with suppliers. It also does not cover smaller items such as clamps or the CO2 tank, nor optional or safety items like check valves or non-return valves.

Figure D29. Using GPT-4 to suggest geometrically-compatible parts from McMaster-Carr. GPT-4 suggested part numbers for the regulator (top left), hose adapter (top right), pressure relief valve (middle), hose (bottom left), and hose fitting (bottom right). All part descriptions were sufficiently detailed such that they described a single part in the database after accounting for the hose inner and outer diameter (prescribed by GPT-4), except for the regulator, which left a choice between a neoprene and stainless steel diaphragm. All part numbers were incorrect, except the part number for the pressure relief valve was surprisingly correct. GPT-4 then produced a comprehensive multi-level BOM.

D.4. Create Manufacturing Instructions (Q4)

D.4.1. *Assembly*

This experiment focused on assembling a wooden box using a specific set of tools and materials. In [Figure D30](#), we presented the prompt for generating machine-readable instructions, which involved creating a set of functions to specify different tasks for the robot and generating corresponding sequences to execute those tasks. Since the functions were designed to be system-agnostic, the response from GPT-4 printed the actions performed by the robot.

 Create a Python script for a household robot that assembles a wooden box with specific dimensions (5x5x5 inches with a thickness of 0.5 inches). The robot has to perform a set of operations including applying wood glue along certain lines, clamping the pieces together in a specific order, drilling a hole in the center of the box with a 1/8 inch drill bit, and finally placing a #8 size, 1.5 inch long wood screw into the drilled hole. The robot should be able to move to a specific location before each operation and should print a statement describing the action it has performed, including the coordinates where the action took place. The operations should be ordered as follows: glue and clamp the bottom panel, left and right sides, back, and finally the top. Then, it should change the drill bit, drill a hole in the center of the box, change the screw, and place the screw in the hole. Specify the tools in the script including 'wood glue', 'power drill with 1/8 inch drill bit', 'bar clamp', and '#8 size, 1.5 inch long wood screw'.

 # The dimensions of the panels
width, height, depth, thickness = 5, 5, 5, 0.5
Function to apply glue
def apply_glue(start, end):
 # Move to start point
 go_to_location(start)
 pick_up('wood glue')
 # Apply glue while moving to end point
 go_to_location(end)
 use_item('wood glue')
 return "Wood glue applied along line from ({}, {}, {}) to ({}, {}, {})".format(*start, *end)
...
Gluing and clamping the panels in the correct order
print(apply_glue((0, 0, 0), (width, 0, 0))) # Bottom Panel
print(place_clamp(width/2, 0, 0)) # Bottom Panel
print(apply_glue((0, 0, 0), (0, height, 0))) # Side Panel (Left)
print(place_clamp(0, height/2, 0)) # Side Panel (Left)
print(apply_glue((0, height, 0), (width, height, 0))) # Side Panel (Right)
print(place_clamp(width, height/2, 0)) # Side Panel (Right)
print(apply_glue((width, 0, 0), (width, height, 0))) # Back Panel
print(place_clamp(width/2, height, 0)) # Back Panel
print(apply_glue((0, 0, depth), (width, 0, depth))) # Top Panel
print(place_clamp(width/2, 0, depth)) # Top Panel
Drilling holes and placing screws
change_drill_bit('1/8 inch drill bit')
print(drill(width/2, height/2, thickness)) # Center of the box
change_screw('#8 size, 1.5 inch long wood screw')
print(place_screw(width/2, height/2, thickness)) # Center of the box

 Can you convert these instructions into a human-readable form as a standard operating procedure?

 **Preparation:**
...
Gluing and Assembly of Panels:
Applying Glue: Apply wood glue in a thin, consistent line along the edges where the panels will join. This includes the edges of the bottom panel and both side panels.
Assembly: Start by placing the bottom panel flat on your work surface. Align each side panel perpendicularly with the edges of the bottom panel, creating a U shape. Make sure the corners are square and the edges flush. Then, apply glue to the edges of the back panel and align it with the bottom and side panels. Finally, apply glue to the top panel and align it with the side panels and back panel.
Clamping: After each panel is placed, secure it with a bar clamp. Ensure clamps are tight, but not too tight to avoid bowing the wood. Leave the glue to dry according to the manufacturer's instructions.
Drilling Pilot Holes:
Drill Bit Installation: Replace the existing bit in your power drill with a 1/8 inch drill bit.
Drilling: Drill pilot holes at all eight corners where the panels join together. The pilot holes should be centered on the edge and go through both pieces of wood. This will help prevent the wood from splitting when the screws are inserted.
Screwing in Screws:
Screw Selection: Use #8 size, 1.5-inch long wood screws. These screws are robust enough to secure the box structure.
Screwing: Screw them into the pilot holes at each corner to secure the box structure. Make sure each screw is flush with or slightly recessed into the wood surface.
...

Figure D30. Instructions for assembling a wooden box using specific tools and dimensions. This figure depicts a Python script for a household robot that assembles a wooden box with specific dimensions. The robot follows a sequence of operations using tools such as wood glue, a power drill with a 1/8 inch bit, a bar clamp, and a #8 size, 1.5 inch long wood screw.

Appendix E. Additional Computer-Aided Engineering Experiments

E.1. Objective Evaluation (Q1)

E.1.1. *Mechanical Properties*

As an example, we focus on analyzing the mechanical integrity of a chair. We began with a simple input design in text form (DS1) and a request for direct evaluation in calculated form (RF1) with an additional binary output asking whether a chair of a given design could support a given load. The specific prompt is included in [Figure E31](#). GPT-4 immediately demonstrated the capacity to handle ambiguity well, assuming a type of wood (oak) and producing numerical material properties for that material when both were unspecified. It made and stated further assumptions about load and failure types. It evaluated the failure point by comparing the yield stress to compressive stress, computed as one quarter of the applied load over the cross-section of a chair leg. This is included in the chat snippet shown in [Figure E31](#). However, in text form it outputted 94,692.2 Pa, while direct evaluation of the equation it listed in the output gives 94,937.7 Pa; thus, GPT-4 occasionally failed to perform basic correct in-line arithmetic or algebra. Although the number is only off by a small amount in this case, it can sporadically differ by much greater magnitudes. Along with the evaluation, it included discussion of other, more sophisticated considerations for failure, such as the type of connection between the legs and the seat. Also, upon repeating the same prompt, GPT-4 would vary whether it included self-weight in the load analysis and whether it evaluated uniform weight or only one leg, leading to small variations in results.

When asking for a function to evaluate chair failure (RF2), GPT-4 successfully generated Python code to evaluate whether a chair will break due to excessive compressive stress on the legs, using the same formula as described in the text exchange (RF1). GPT-4 was able to readily add multiple types of failure without error, also incorporating bending failure of the seat, and excessive stress on the back using simple beam bending and structural mechanics equations. This multipart failure assessment is included in [Figure E31](#). It further automatically generated a function that could intake a parametric chair design with sensible feature parameters like `leg_cross_sectional_area`, `seat_thickness` and `seat_material_bending_strength`, allowing versatile use of this evaluation.

When generating the function, it continued to handle ambiguity by make assumptions including that the load would be distributed across all four legs, centered and uniform on the seat, and that the load on the back of the chair would be one-third of the total weight. In the case of writing the function (RF2) as compared to text evaluation (RF1), it did not explicitly list all of the assumptions; rather, they had to be interpreted based on the equations used. GPT-4 also incorporated several small errors and oversights in both cases. For instance, when generating a function to evaluate seat bending failure, it treated the seat as a simply supported cantilever beam, and assumed that the chair would break along the width (separating front from back) rather than along the length or at an angle to the base. It also assumed that the bending stress on the back was evaluated as the load over the total area of the back rather than at the connection surface of the back to the seat of the chair.

However, as these functions were identified, they could be further refined by iterated discourse with GPT-4 to produce a more correct function.

 Given this chair made of wood, that has 4 legs that are 1 meter long (... content omitted by authors ...) can it support 100kg? Please give a final yes or no answer.

 To determine whether the chair can support 100kg, we need to consider the strength of the wood used to construct the chair, as it determines the load-bearing capacity. However, this information was not provided. Nonetheless, let's assume that the wood used is Oak, which has a fairly high compressive strength of around 37 MPa or 37,000,000 Pascals. (... content omitted by authors ...)

If we consider each leg bears 1/4 of the total weight, each leg will support 25kg. The pressure (force per unit area) on each leg then is $\text{Force/Area} = (25\text{kg} * 9.8 \text{ m/s}^2) / 0.00258064 \text{ m}^2 = 94,692.2 \text{ Pascals}$ or 94.7 kPa, which is well within the compressive strength of Oak, so the legs should hold up. (... content omitted by authors ...)

Thus, based on these calculations and assumptions about the distribution of weight and the type of wood used, the answer is **Yes**, the chair can support 100kg.

 Give me a function that can evaluate whether a chair of a typical design will break under a certain applied weight. Please consider multiple types of failure.



```
def will_chair_break(weight, leg_cross_sectional_area, leg_material_strength,
    seat_thickness, seat_length, seat_width, seat_material_bending_strength,
    back_height, back_width, back_material_strength):
    # Calculate the load on each leg
    load_per_leg = weight / 4
    # Calculate the stress on each leg
    stress_on_leg = load_per_leg / leg_cross_sectional_area
    # Check if the stress on each leg exceeds the strength of the leg material
    if stress_on_leg > leg_material_strength:
        return True # The chair will break
    (... omitted by authors: code continues on to assess seat and back bending ...)
```

Properties assessed



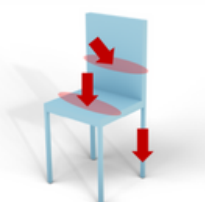


Figure E31. Comparing text and function-based performance evaluation of chair mechanics. When using text-based analysis, GPT-4 evaluates stresses one-by-one with more mistakes in computation and fewer features able to be assessed at once compared to function-based analysis.

In a comparison of these two output form requests, RF1 and RF2, functional evaluation was easier to read, more accurate, and able to be implemented for a variety of input designs, but directly incorporated more assumptions into equations. During both types of evaluation, GPT-4 actively reported on potential causes of error in the evaluation, such as how the chair legs were attached to the seat. It consistently overlooked potential causes of failure such as buckling of the legs unless specifically prompted. We found GPT-4 to adequately assess most basic mechanical properties of interest.

Some properties relying on an understanding of the spatial arrangement of chair components were not able to be adequately assessed. GPT-4 had significant trouble generating a suitable evaluation of stability, and failed entirely to calculate a reasonable center of gravity for an input design despite many attempts. The closest attempt using the simple assumption that the center of gravity would be in the center of the chair seat.

However, other complex physical properties were readily assessed. GPT-4 generated first-order code to assess the failure of a chair upon impact with a spherical projectile, with no difference in quality of the computation compared to static mechanical properties.

To evaluate GPT-4's performance on code-based input (DS3 and DS4), we provided GPT-4 with an OpenJSCAD chair specification. When the parameters and parts of the chair were clearly named salient features (DS3) like `backThickness`, `leg1`, `chairSeat`, and `chairBack`, GPT-4 was readily able to recognize the item as a chair and analyze desired properties, such as the breaking load of the seat. However, when we used identically structured code with variable and object names that had been obscured (DS4), it could not recognize parts of the item to assess properties, for example, to locate the seat or synonyms of the seat. This was true whether the names had been slightly obscured (e.g., as `xZ_height`, `stick1`, `platform`, and `barrier`, respectively) or entirely obscured (e.g., as `Q`, `A1`, `B1`, and so on). When asked about the design in the two obscured forms, GPT-4 guessed that the final item was a table with a narrow bookshelf and exhibited poor interpretation of the design and parts. Even when GPT-4 was challenged, it claimed that it could not be a chair because the back was not connected appropriately to the chair seat; this was an incorrect interpretation of the code, again indicating poor spatial reasoning. In a second case, when an input design for a cabinet (DS3) had one variable named `shelfAllowance` (used to slightly reduce the shelf width for easy assembly), GPT-4 erroneously assumed that this indicated number of shelves. These results reinforce the idea that LLMs perform based on semantics, and that a design without clear descriptive words becomes much less manageable, causing DS4 to generally fail.

One potential limitation in the use of GPT-4 and LLMs for this kind of analysis: the source material for equations and standards of analysis may be unknown or even intentionally anonymized. Even when the equations are the standard first-order textbook equations per topic, they are almost always unreferenced. When different standards exist, across different countries or for different use cases, much more refinement would be needed to use GPT-4 to assess the mechanical integrity of a design. In addition, these equations often work well for objects of a typical design, but for edge cases or unusual designs they would miss key failure modes, such as the buckling of a table with very slender legs or the excessive bending of a chair made from rubber.

In a particularly apparent example of this type of failure (i.e., creating functions based on pattern-matching rather than judicious observation of likely failures), GPT-4 was asked over a series of iterations to help write code to render a spoon with sizes within a set of ranges in OpenJSCAD, then to assess ergonomics, which it evaluated based on dimensions. Finally, we requested GPT-4 to create a function to compute the spoon's breaking strength. Since it had been inadvertently primed by the long preceding discussion of spoon geometry, it proposed a strength evaluation using the basic heuristic of whether the spoon is within a standard size range ([Figure E32](#)). GPT-4 had to be prompted specifically for a yield analysis before offering a mechanics-based equation. At that point, it continued to handle ambiguity well and chose a most likely breaking point (the point between the handle and spoon scoop). But for a novice design engineer who might have assumed GPT-4's

initial output was sound, this bold proposition of an unreasonable strength analysis on first pass without further explanation causes some alarm. This serves as a reminder to not rely on GPT-4 alone without external validation of every step.

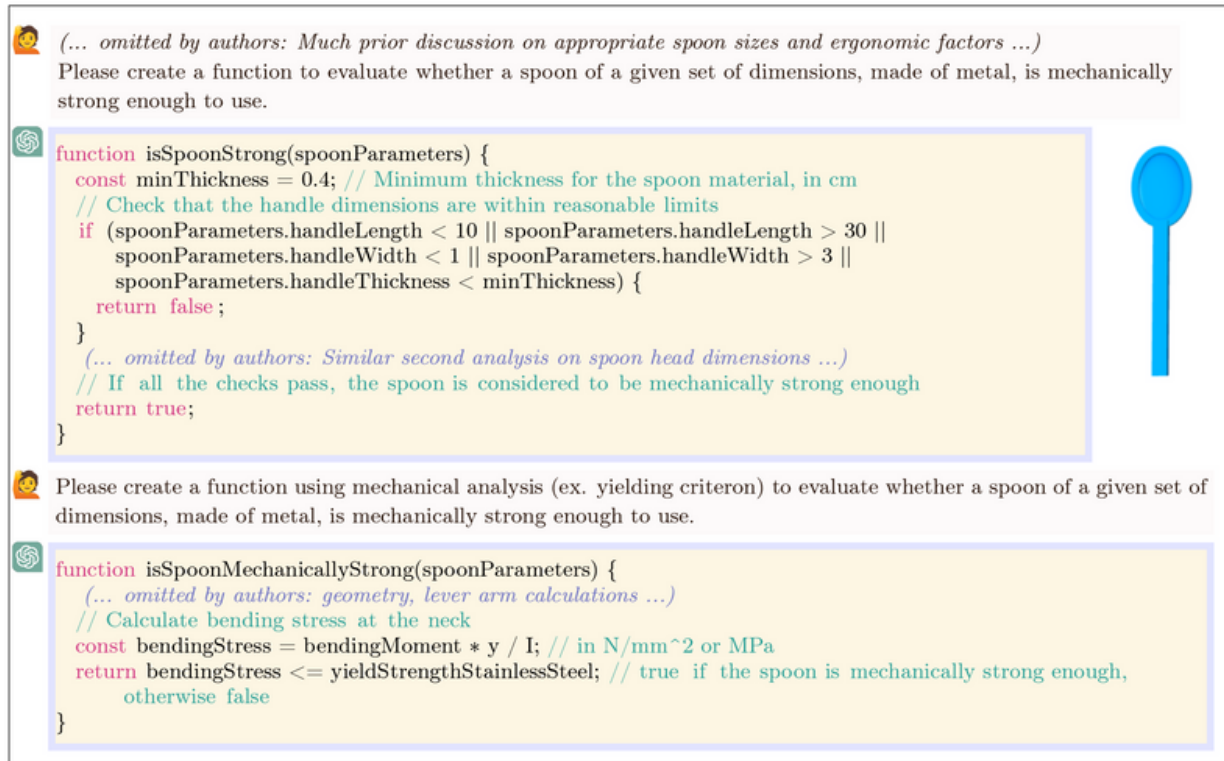


Figure E32. Chat history errors and correction for spoon mechanics. When analyzing mechanics of a spoon after discussing dimensions in the preceding chat, GPT-4 generated a poor heuristic for spoon breaking from geometry alone; with very specific correction in the same chat, it recovered.

E.1.2. Finite Element Analysis

To investigate the computational performance analysis capabilities of GPT-4, and to build on the first-order mechanical calculations already done, we challenged it to develop a comprehensive framework for advanced performance analysis and structural evaluation using the finite element method (FEM). The primary focus was determining the likelihood of a chair breaking when subjected to external forces. [Figure E33](#) lists the response and final code generated by GPT-4. With the application of FEM through the external library FEniCS, GPT-4 evaluates the von Mises stress, a crucial parameter in material failure prediction. By comparing this stress with the yield strength of the material, one could assess if the chair would fail under the applied load. For the development of the code, substantial back-and-forth iteration was required to create successful code due to its overall complexity. One helpful point for gradually increasing complexity was to create code for a 2D example before asking GPT-4 to create a 3D version. In spite of these challenges, GPT-4 was highly efficient and

successful in formulating a precise solution using the FEniCS library, an advanced tool for numerical solutions of PDEs. Not only did GPT-4 integrate the library into the Python code correctly, but it also applied a wide variety of FEniCS features, including defining material properties and boundary conditions and solving the problem using FEM. Caution must be taken, as GPT-4 occasionally suggests libraries and functions that do not exist. However, with correction it quickly recovers and suggests valid options.

The stress distribution visualization in [Figure E33](#) is performed on the chair previously designed by GPT-4 in [Figure B10](#) and is the output of GPT-4's code rendered in Paraview (which GPT-4 also gives assistance to use), as well as on a chair mesh found from other sources. The result reveals a susceptibility to high stress at the back attachment section of the chair design proposed by GPT-4, as seen in [Figure B10](#). This observation underscores the potential for future enhancements in this object's design.

Beyond code generation, GPT-4 also lends support in the local installation of these external libraries, such as FEniCS, so users can run the generated code. This assistance proves invaluable for practitioners who may have limited familiarity with these libraries, which are initially suggested by GPT-4 itself. Notably, studies have delved into the potential of GPT-4 to generate code integrating other external libraries, like OpenFOAM, for the purpose of performing computational performance analysis ([Kashefi & Mukerji, 2023](#)).

It is worth noting that GPT-4's capabilities in utilizing these libraries have certain limitations. It can only harness some of the basic features of FEniCS and struggles with more specific, custom usages of the library, such as applying complex loading conditions. Furthermore, GPT-4 assumes homogeneous material properties for the chair, an oversimplification that does not align with the more diverse characteristics found in real-world materials. Moreover, the training date cutoff for GPT-4 means that sometimes only older functions or libraries may be used, without current updates.



Figure E33. Finite element analysis for chair mechanics. GPT-4 determines if the chair will break by performing FEM-based stress analysis. In images, the top chair analyzed is designed by GPT-4 in Figure B10 with von Mises stress displayed from blue (zero) to a maximal value (red). A second chair with mesh from an external source is analyzed using the same code.

E.2. Subjective Evaluation (Q2)

As a testbed for evaluating GPT-4's subjective evaluation, we generated a simple parametric four-legged chair with a back, then input eight versions with different leg lengths, seat widths, and back heights into GPT-4(DS1). GPT-4 was then asked three similar queries: (1) assign to each chair a label of "large," "medium," or "small" (RF3); (2) rank all chairs from largest to smallest; and (3) in a pairwise comparison, indicate if a given

chair was larger or smaller than another. Each of these inputs were given independently, to not influence the different tests based on prior answers in the chat dialogue. In each case, GPT-4 assigned the same overall ranking. [Figure 21](#) shows the chairs rendered in ranked order including the labels for categorization, using a combined implicit consideration of seat area, back height, and leg height. In a similar query, spoons of different handle length and thickness, and scoop length, width, and curvature were compared, finding similar results. In that case, GPT-4 elected to compare spoons by the length of the scoop alone, handling the ambiguity of the question by making a decision about what single quantity mattered most. When handling higher levels of ambiguity, for example, assigning comfort levels to shoes described in text input, GPT-4 sometimes refused to answer. To bypass this, we determined that it was essential to ask GPT-4 directly to give an output of a certain kind, such as classification into set categories. For instance, the question 'Is this shoe comfortable?' would raise objections, a nonanswer, and a discussion of what contributes to general shoe comfort. We could circumvent this by asking 'Please state whether this shoe is likely very comfortable, comfortable, uncomfortable, or very uncomfortable, and provide one sentence of justification.' Despite its continued objections, GPT-4's responses were usually reasonably justified, noting aspects like use of breathable material, adjustability of laces, shock absorption, and traction of the sole. These results indicate that the semantics of the type of assessment (ranking, categorization, or scoring) do not have a large influence on the final result of subjective analysis, as long as some type is chosen. However, certain prompt structures may be required to avoid refusals to answer, and the simplest prompt structure to ensure this was asking for any certain kind of output response.

To challenge GPT-4 to evaluate subjective criteria dependent on more abstract input parameters, we asked it to create a list of key criteria that go into evaluating sustainability, and to evaluate designs based on these criteria, scoring each category from one to ten (RF4). In practice, the justifications for each property score were generally reasonable but rarely entirely correct. For example, the remark in [Figure 22](#) for `modular_design` about swapping seat shells was a misinterpretation of the product description: chairs with different seat shell colors were available for purchase, but a single chair could not swap shells.

A range of historical periods were identified that influence furniture design, including Egyptian, Greek and Roman, Renaissance, Bauhaus (a semi-minimalist German-inspired design including rounded features), and minimalist. GPT-4 identified criteria to differentiate between these historical styles based on seven properties: material choice, decorative complexity, evidence of handcrafting, extent of ornamentation, deviation from standard proportions, upholstery use and quality, and material innovation. Based on these categories, GPT-4 evaluated each historical period and chair, and created a function to use the scores to categorize the style of each chair. A selection of text from one input/output is included in [Figure 22](#).

The results of the categorization ([Figure 22](#)) seem generally reasonable, with most chairs placed into categories that appear subjectively appropriate; a plain metal stool was classified as minimalist, a soft lounge chair with a floral pattern was classified as Renaissance, and a double end chaise lounge was classified as Greek and

Roman. A couple of types of mistakes occurred in the classification. First, most chairs were sorted into the minimalist category, including the faux leather swivel lounge chair and two soft-sided recliners (not shown). Second, several other design styles that may have been a better fit were included in the scoring but were not found to be best fits in the evaluation, indicating that this set of GPT-4's scoring for the historical periods was not appropriately distributed to capture the right features for all chairs. Third, upon reevaluating scores over a few iterations, we found that different categories could be established and chairs could switch categories at times due to subjective scoring. Nevertheless, these general issues persisted, such as occasional mistaken categorizations and having one 'catch-all' category that was used more than others.

Appendix F. Additional Design Optimization Experiments

F.1. GPT-4: Analytical vs. Outside Tools (Q1)

Consider an example in which we maximize the stability of a table ([Figure F34](#)). GPT-4 correctly describes that an object is statically stable when its center of mass lies within its support polygon. One considers stability of an object to be maximized when it remains stable under as large of a perturbation as possible. In principle, that typically means two things: 1) moving the center of mass as far away from the boundary of the support polygon as possible, 2) decreasing the object's experienced motion (typically caused by gravitational torquing) when perturbed. GPT-4 is able to apply these intuitive principles to reason about the optimal solution within given bounds in this case.

In a similar example shown in [Figure F35](#), the Wolfram plugin is enabled, which GPT-4 can selectively call at its discretion. While the Wolfram plugin was a natural choice for solving what appears to be a simple analytical optimization problem, GPT-4 timed out. In practice, this can happen for at least three reasons: 1) not enough time was allotted for the computation, 2) the problem is too difficult for Wolfram to handle, or, more generally, 3) the query may produce a problem that is not tractable or—in the extreme case—not computable ([Turing, 1936](#)). Although it might seem like it would be trivial to provide Wolfram with more time to complete the computation, in practice, the user has no feedback on how long the computation would take. It is unreasonable to ask a user to wait indefinitely without feedback, and most numerical optimization algorithms will be unable to provide a reasonable estimate of progress. In this case, 'anytime algorithms' (which can return a valid partial or approximate solution even if interrupted early) may be especially practical ([Zilberstein, 1996](#)).

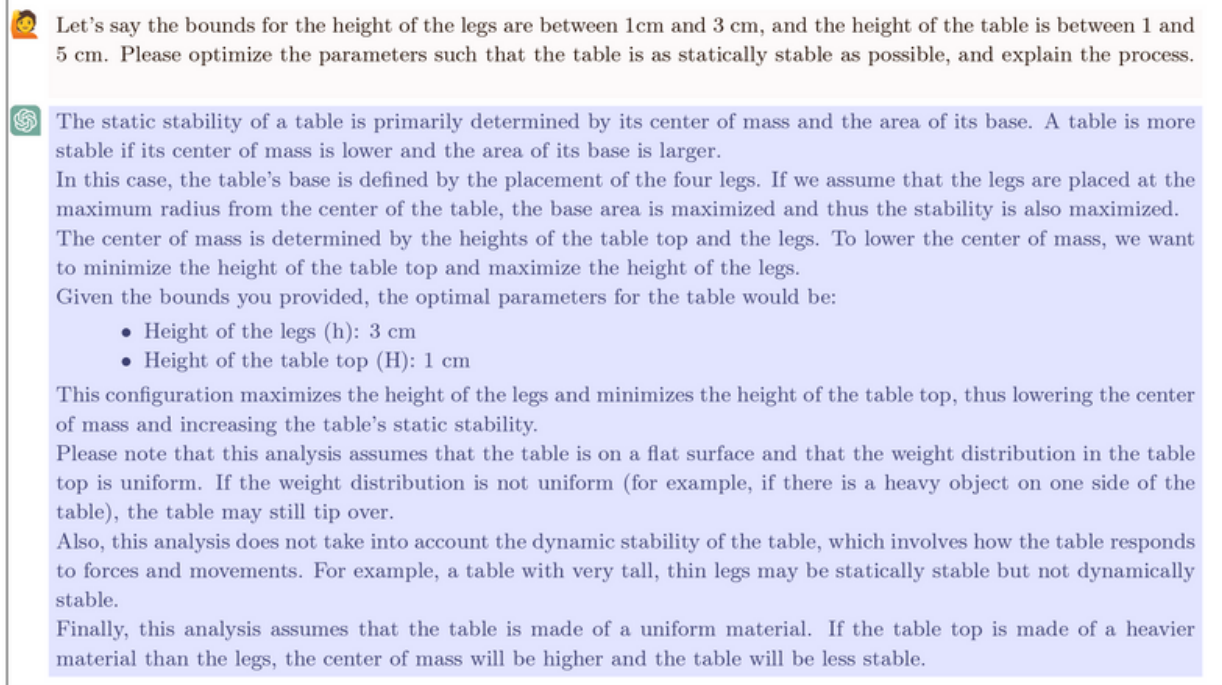


Figure F34. Intuitive optimization. Without specific prompting to take an analytical or computational approach, GPT-4 may reason intuitively about the physics of a problem. This example is an offshoot of the conversation in [Figure F35](#), after the initial exchange.

Let's say I want to design a table with four legs and a top. Assume that the four legs are radially symmetric, and the top is a cylinder and the legs are cylinders. Can you give me the equation for the center of mass of the table in x, y, and z, as a function of the natural parameters of the primitives?

Sure, let's denote the following parameters:

- R is the radius of the table top.
- r is the radius of each leg.
- H is the height of the table top.
- h is the height of each leg.
- ρ is the uniform density of the material the table is made of.

The mass of the table top M_{top} and each leg M_{leg} can be calculated using the volume of a cylinder $V = \pi r^2 h$ and the definition of density $\rho = \frac{M}{V}$ which gives us:

- $M_{top} = \rho \pi R^2 H$
- $M_{leg} = \rho \pi r^2 h$

The total mass of the table M is the mass of the top plus the mass of four legs:

- $M = M_{top} + 4M_{leg}$

The center of mass of the table in the z-direction z_{cm} can be calculated by taking the weighted average of the centers of mass of the top and the legs. The center of mass of a cylinder is at its midpoint, so for the top it's at $\frac{H}{2}$ and for the legs it's at $\frac{h}{2}$. This gives us:

- $z_{cm} = \frac{1}{M} (M_{top} \frac{H}{2} + 4M_{leg} \frac{h}{2})$

Since the table is radially symmetric, the center of mass in the x and y directions will be at the origin (0,0):

$$x_{cm} = y_{cm} = 0$$

So the center of mass of the table is at $(0, 0, z_{cm})$.

Ok. Let's say the bounds for the height of the legs are between 1cm and 3 cm, and the height of the table is between 1 and 5 cm. Show me an equation that would measure static stability, and find the minimum of that equation.

The static stability of a table can be quantified by the height of the center of mass. The lower the center of mass, the more stable the table is. Therefore, we can use the equation for z_{cm} as our measure of static stability.

The equation for z_{cm} is: (... omitted by authors: substituting in equations found in previous response. Below is the final result. ...)

Since R , r , and ρ , we can treat this as a function of h and H only:

- $f(h, H) = \frac{R^2 H^2 + 2r^2 h^2}{R^2 H + 4r^2 h}$

We can find the minimum of this function within the given bounds using calculus or numerical optimization methods. However, because this is a complex optimization problem, it would be best to use a numerical method.

Let's use the Wolfram Language to find the minimum of this function within the given bounds. (... omitted by authors: using Wolfram plugin ...)

I apologize for the inconvenience. It seems that the computation took longer than expected. This is a complex optimization problem that might require a significant amount of computational resources. Let's try a different approach.

(... omitted by authors: GPT-4 assumes that the minimum occurs at one of the four corners of the parameter space and calls Wolfram plugin to output the results ...)

Figure F35. Attempt to optimize a table analytically. Although GPT-4 is able to provide a formula for measuring the static stability of a table, the Wolfram plugin is unable to perform the optimization and instead defaults to an approximation of the function minimum.

After failing to optimize over the full space using Wolfram, GPT-4 continues the conversation by reasoning that the optimum value will occur near the boundary of the constraints (Figure F35). By exploiting this reasoning, it successfully uses the Wolfram plugin to compute and evaluate the equations corresponding to a small set of extremal points in the design space. Despite this, it fails to realize that certain solutions dominate

others, and does not prune out bad candidates. Moreover, GPT-4 neglects to justify or prove its claim that the optimum should occur near the boundaries; though it was correct in this case, this approach may fail in general.

In a follow-up experiment ([Figure F36](#)), GPT-4 is asked to perform the same optimization task via Python code, which enables it to use an external library. It chooses L-BFGS-B, which is a reasonable, standard, and easily accessible (though not state-of-the-art) solver for continuous valued problems. It does not, however, provide gradients that can expedite the computation unless prompted for them. We explicitly prompt GPT-4 to provide the gradients ([Figure F37](#)) and visualize the results in [Figure F38](#). Generally speaking, the unoptimized approach on GPT-4's part is an issue with respect to performance, as not all users will be intimately familiar with all (or perhaps any) optimization libraries, and they may not realize that by providing additional information (e.g., gradients), the computation can be expedited. GPT-4 also does not elect to make use of Wolfram or autodifferentiation; in practice, lack of direct computation can lead to errors. Later in this section, we demonstrate how GPT-4 struggles to solve a (much) more difficult version of this optimization problem.

 Since Wolfram is having trouble, can you instead provide me code to find the optimum in the general case?

 (... content omitted by authors ...)

```
import numpy as np
from scipy.optimize import minimize

# Define the function
def f(x, r, R):
    h, H = x
    return (R**2 * H**2 + 2 * r**2 * h**2) / (R**2 * H + 4 * r**2 * h)

# Define the bounds and initial guess
bounds = [(1, 3), (1, 5)]
x0 = [2, 3] # initial guess

(... omitted by authors: settings values of r and R ...)
```

 # Find the minimum of the function
res = minimize(f, x0, args=(r, R), bounds=bounds)

```
print("The minimum of the function is at h = {:.2f} cm, H = {:.2f} cm".format(*res.x))
```

(... content omitted by authors ...)

 Which optimization method does minimize use?

 The scipy.optimize.minimize function in Python uses the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm by default for unconstrained problems. The BFGS method is a quasi-Newton method that uses function values and gradients to build up a picture of the surface to be optimized.

However, when bounds are provided (as in your case), minimize defaults to the Limited-memory BFGS (L-BFGS-B) method. The L-BFGS-B method is a modification of the BFGS method that is more suitable for problems with many variables and constraints on the variable values.

The method to be used can be specified using the method parameter of the minimize function. Other methods available include 'Nelder-Mead', 'Powell', 'CG', 'TNC', 'COBYLA', 'SLSQP', and others. The choice of method depends on the nature of the problem (e.g., whether it's constrained, whether the function is smooth, whether gradients are available, etc.).

 What do you think is the best choice here?

 Given that we have bounds on the variables, the Limited-memory BFGS (L-BFGS-B) method is a good choice for this problem. L-BFGS-B is an optimization algorithm in the family of quasi-Newton methods that approximates the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm using a limited amount of computer memory. It's a popular choice for optimization problems where there are constraints on the parameters.

The L-BFGS-B method is particularly suitable for high-dimensional problems, and it can handle the bounds on the parameters efficiently. It also doesn't require the computation of the second derivatives (the Hessian matrix), which can be computationally expensive for complex functions.

Therefore, using the L-BFGS-B method with the scipy.optimize.minimize function, as in the provided Python code, would be a good approach for finding the minimum of the function within the given bounds.

Figure F36. GPT-4 informs user on choice of optimization method. When prompted to choose an alternative optimization method in Python to the Wolfram Alpha plugin, GPT-4 provides a script and a reasonable explanation for the chosen solver.

Can you compute expressions for the gradients of this problem, and provide code that uses them to accelerate the L-BFGS-B call?

(... omitted by authors: partial derivatives computation ...)

```
import numpy as np
from scipy.optimize import minimize

# Define the function
def f(x, r, R):
    h, H = x
    return (R**2 * H**2 + 2 * r**2 * h**2) / (R**2 * H + 4 * r**2 * h)

# Define the gradient of the function
def grad_f(x, r, R):
    h, H = x
    dfdh = (4 * r**2 * h * (R**2 * H + 4 * r**2 * h) - 2 * r**2 * h**2 * (4 * r**2)) / (R**2 * H + 4 * r**2 * h)**2
    dfdH = (2 * R**2 * H * (R**2 * H + 4 * r**2 * h) - R**2 * H**2 * (R**2)) / (R**2 * H + 4 * r**2 * h)**2
    return np.array([dfdh, dfdH])

# Define the bounds and initial guess
bounds = [(1, 3), (1, 5)]
x0 = [2, 3] # initial guess

(... omitted by authors: settings values of r and R ...)

# Find the minimum of the function
res = minimize(f, x0, args=(r, R), jac=grad_f, bounds=bounds)
print("The minimum of the function is at h = {:.2f} cm, H = {:.2f} cm".format(*res.x))
```

Figure F37. Prompting GPT-4 to accelerate optimization. By default, GPT-4 does not compute gradients for the problem even when they are available. When directly prompted, GPT-4 modifies its code to pass in the gradients of the objective with respect to optimized parameters.



(A) Initial Table Design



(B) Optimized Table Design

Figure F38. Single objective optimization. Left: A table with initial leg height of 2 mm and tabletop height of 3 mm. Right: Given the specification of maximizing table stability, GPT-4 formulates the objective as minimizing the z-coordinate of the full assembly's center of mass with respect to leg height and tabletop height. GPT-4 provides a Python script that computes the gradients and uses L-BFGS-B, as provided through scipy.optimize to optimize the two parameters. We visualize the optimized table, in which both the leg and tabletop heights are as small as possible.

Throughout these experiments, we noticed several common issues in GPT-4's approach. First, if users do not prompt GPT-4 explicitly to show its work, it may resort to 'common-sense' reasoning about a problem. Although this reasoning *could* be correct, GPT-4 provides no certificate to a user, as seen in the 'intuitive'

physics of [Figure F34](#), or the boundary-aligned optima assertion in [Figure F35](#). Another issue occurs if it is difficult to find a library to solve a particular task; in this case, GPT-4 often gives up or attempts to write its own code. If the code is detail-heavy, it may be too difficult for GPT-4 to write correctly and the code/solution may be incorrect. If a library does exist but is used uncommonly, GPT-4 may give incorrect instructions on how to install/use that library; or, in some cases, GPT-4 may hallucinate a library altogether.

F.2. Reasoning About Different Problem Types and Selecting Appropriate Solvers (Q2)

F.2.1. *Robot Arm Optimization*

In this example, shown in [Figure F39](#), a robot arm is to be optimized such that it reaches a target location in space. As requested, GPT-4 generates a two-link robot design parametrized by the link lengths, and then uses inverse kinematics to provide a solution for the link lengths so as to reach a target location in space. When asked to transform this into a design optimization problem, GPT-4 sets up an optimization problem, creating an appropriate constraint (end-effector touching goal), an objective (sum of link lengths, as a proxy for material cost), and parameters with reasonable bounds. All of these were automatically provided by GPT-4, without explicit request. Notably, the optimization code is easily generalizable to arbitrary locations in space (though certain aspects like parameter bounds may need to be modified). As an optimization procedure, GPT-4 chooses L-BFGS; a reasonable choice given the continuous nature of the problem. A rendering of the optimized robot can be seen in [Figure F40](#).

I'm trying to design a robot arm that can move its end effector directly vertically half a meter. Can you help by providing a design for such a robot? It should only use rotational motors. I need to know the morphology of the robot - the dimensions of the links, and how they are connected. Or, if you don't know the dimensions or the morphology, you can provide code to search for dimensions that would work for the task (through simulation).

(... content omitted by authors ...)
Now, for more complex morphologies, you would want to conduct a search via simulation to find optimal lengths and configurations for the robot's arm. A simple Python-based example using inverse kinematics could look like this:

```
import numpy as np
(... omitted by authors: setting desired vertical distance to 0.5 and initial arm link lengths to 0.25 ...)

# Calculate the angles using inverse kinematics
theta2 = np.arccos((d**2 - l1**2 - l2**2) / (2 * l1 * l2))
theta1 = np.arctan2(l2 * np.sin(theta2), l1 + l2 * np.cos(theta2))

# Confirm that the end effector reaches the desired position
x = l1 * np.cos(theta1) + l2 * np.cos(theta1 + theta2)
y = l1 * np.sin(theta1) + l2 * np.sin(theta1 + theta2)
print("The end effector is at position: ", (x, y))

# Now you can run a loop to iterate over various lengths for l1 and l2
# within a feasible range and see which combination gives you the best result.
# You would have to define "best" - could be smallest, lightest, simplest, etc.

(... content omitted by authors ...)
```

Below is the code you gave to perform inverse kinematics on a 2 link robot arm: (... omitted by authors: gpt provided code ...)
However, when I set $l1 = l2 = 0.5$, the end effector is at the position $(-0.25000000000000006, 0.4330127018922192)$ instead of the goal position of 0.5 vertical. Could you fix this error?

(... omitted by authors: failed attempt at fixing the code ...)

I still get the same error. Could you fix the script?

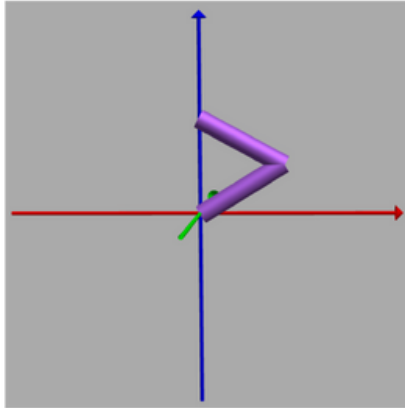
(... content omitted by authors ...)
import numpy as np
(... omitted by authors: setting desired vertical distance to 0.5 and initial arm link lengths to 0.5 ...)

Calculate the angles using inverse kinematics
theta2 = np.arccos((x_target**2 + y_target**2 - l1**2 - l2**2) / (2 * l1 * l2))
theta1 = np.arctan2(y_target, x_target) - np.arctan2(l2 * np.sin(theta2), l1 + l2 * np.cos(theta2))

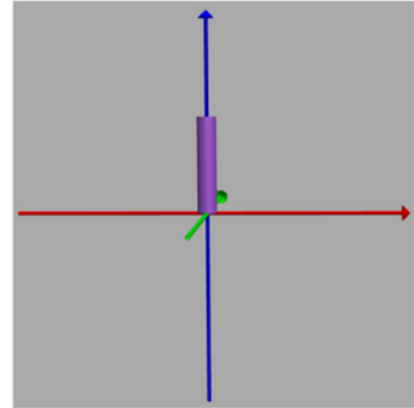
Confirm that the end effector reaches the desired position
x = l1 * np.cos(theta1) + l2 * np.cos(theta1 + theta2)
y = l1 * np.sin(theta1) + l2 * np.sin(theta1 + theta2)
print("The end effector is at position: ", (x, y))

(... content omitted by authors ...)

Figure F39. Inverse kinematics for two-link robot arm. Although GPT-4 is able to output inverse kinematics code for optimizing the design of a two-link robot design, its initial attempt is unsuccessful, requiring a couple iterations of prompting to achieve a correct output. However, we do not explicitly need to point out the source of the error for GPT-4 to correct itself.



(A) Initial Robot Arm



(B) Optimized Robot Arm

Figure F40. Optimizing link lengths of a robot arm. The goal is to minimize the link lengths of a robot arm with the constraint that the arm is capable of reaching a goal located 0.5 m vertically. **Left:** The initial arm features link lengths of 0.5 m each, requiring both arms to be bent to reach the goal. **Right:** GPT-4 outputs an optimization script to discover that the optimal link lengths are 0.25 each, which we visualize here.

F.2.2. Optimizing 3D Printing Parameters



I have a 3D printer. I'm trying to optimize the parameters of the slicer. The parameters are continuous values. Each time I do a 3D print, I'm able to grade the quality of the print on a scale of 0 to 1, with 0 being a bad score and 1 being a perfect score. I need an algorithm for optimizing over the parameters. What method would you recommend?



Given your problem, I would recommend you to use Bayesian Optimization. *(... omitted by authors: explanation of how Bayesian Optimization works ...)* There are several libraries in Python that provide an implementation of Bayesian Optimization. Some of these include:

- **Scikit-optimize**: It provides a BayesSearchCV class that's similar to GridSearchCV and Randomized-SearchCV in Scikit-learn.
- **GPyOpt**: This is another Python library for Bayesian Optimization, which is based on GPy.
- **BayesianOptimization**: This is a pure Python implementation of Bayesian optimization, and it's easy to use.

You could use any of these libraries to implement Bayesian Optimization and find the optimal parameters for your 3D printer.



Thank you. Let's assume I have a function, then, called evaluate. evaluate takes in parameters as a list, and returns a score for the quality. Can you provide example code for this optimization? You can assume a variable called "parameters" is given.



Sure, let's use the BayesianOptimization library to illustrate this process. Let's also assume that you have two parameters you are tuning and they both take values between 0 and 1 (you can adjust this to your actual situation). *(... omitted by authors: software installation instructions. ...)* Then, you can use the following code to perform the optimization:

```

from bayes_opt import BayesianOptimization

# Assuming the evaluate function and parameters are defined.
# The function should take the parameters as input and return a quality score.

# Let's assume parameters are ['param1', 'param2']
def function_to_optimize(param1, param2):
    score = evaluate([param1, param2]) # Use the real evaluation function here
    return score

# Define the bounds of the parameters
pbounds = {
    'param1': (0, 1), # These should be the min and max values of param1
    'param2': (0, 1), # These should be the min and max values of param2
}
# Instantiate the optimizer object
optimizer = BayesianOptimization(
    f=function_to_optimize,
    pbounds=pbounds,
    random_state=1,
)
# Perform the optimization
optimizer.maximize(
    init_points=2, # Number of randomly chosen points to sample the target function before starting the
                  # optimization.
    n_iter=10, # Number of iterations to perform the optimization
)
(... omitted by authors: retrieving optimal parameters from the 'max' attribute of the optimizer object and printing them ...)
(... content omitted by authors ...)

```

Figure F41. Bayesian optimization for 3D printing. GPT-4 reasonably chooses Bayesian optimization for fine-tuning parameters for systems that require real-world experiments for evaluation, and provides code for an (abstracted) instantiation of the problem.

In this more abstract example, GPT-4 is simply asked which algorithm to use in order to optimize the parameters of a slicer used in 3D printing. It chooses Bayesian optimization, which is a good choice for problems with real-world experiments where it is preferable to minimize the number of required experiments. GPT-4 also provides skeleton code for the optimization. As this is a more abstract example, specifics are not supplied. The listing can be found in [Figure F41](#).

F.2.3. *Cabinet Optimization*

We investigate whether GPT can output a reasonable cost function and design that optimizes the function when provided an example design, a parameterization of the design space, and a text description of the objective. One instantiation of this problem setting is with furniture: can GPT optimize the design of a cabinet such that the result has a user-specified volume while minimizing the cost to build it? First, we prompt GPT-4 with an example cabinet design in OpenJSCAD ([Section 6.1.1](#)) and a parameterization of the design (including bounds for the parameters). Then, we ask it to generate functions to compute volume and material cost. Once the user verifies the accuracy of the functions, we have GPT-4 output a Python script that can minimize the cabinet's material cost with respect to a given volume constraint. The resulting code is shown in [Figure F42](#), with renders of an optimized cabinet in [Figure F43](#).

Assume that you can call the below functions.

```
def compute_storage_capacity(cabinet_height, cabinet_width, cabinet_depth, board_thickness):
    (... code omitted by authors ...)
    return storage_capacity

def compute_material_cost(cabinet_height, cabinet_width, cabinet_depth, board_thickness,
    cost_per_cubic_inch):
    (... code omitted by authors ...)
    return material_cost
```

In addition, a cabinet design can be parameterized as follows. The bounds of each parameter is specified along with rationale for these bounds.

- boardThickness: (0.1, 5) inches. (... omitted by authors: brief explanation of parameter bounds ...)
- cabinetHeight, cabinetWidth, cabinetDepth: (10, 100) inches. (... omitted by authors: brief explanation of parameter bounds ...)
- (... omitted by authors: full set of parameters ...)

Write a script to find the cabinet of the correct volume that minimizes cost using any library you want. If the volume constraint is not satisfied, specify this. Assume that the cost per cubic inch of the materials, the board thickness, and the volume constraint will be user specified. Demonstrate that the script works with example user specifications.

```
(... content omitted by authors ...)
from scipy.optimize import minimize

(... omitted by authors: setting values of volume constraint, cost per cubic inch, board thickness ...)

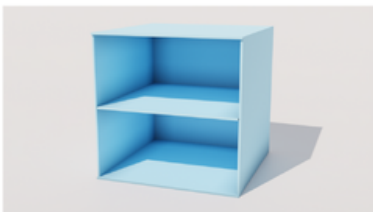
def objective_function(x):
    (... omitted by authors: function description ...)
    cabinet_height, cabinet_width, cabinet_depth = x # Extract cabinet dimensions
    return compute_material_cost(cabinet_height, cabinet_width, cabinet_depth, board_thickness,
        cost_per_cubic_inch)

def constraint_function(x):
    (... omitted by authors: function description ...)
    cabinet_height, cabinet_width, cabinet_depth = x # Extract cabinet dimensions
    volume = compute_storage_capacity(cabinet_height, cabinet_width, cabinet_depth, board_thickness)
    return volume - desired_volume # This should be equal to zero when the desired volume is met

# Initial guess (start at the mid-point of the possible range for each dimension)
x0 = [55, 55, 55]
# Define bounds for each dimension
bounds = [(10, 100), (10, 100), (10, 100)] # Lower and upper bound for each dimension
# Define the constraints
constraints = [{'type': 'eq', 'fun': constraint_function}]
# Call the optimizer
result = minimize(objective_function, x0, method='SLSQP', bounds=bounds, constraints=constraints)

(... omitted by authors: prints optimized dimensions if successful, otherwise reports failure ...)
(... content omitted by authors ...)
```

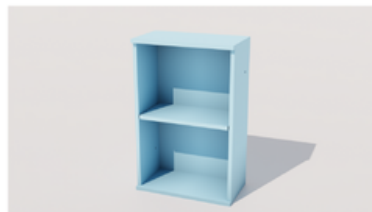
Figure F42. Optimizing material cost of cabinet with respect to volume. The performance and cost functions are generated by prompting GPT-4. To find the minimum of the material cost function with respect to volume, GPT-4 calls `scipy.optimize`'s `minimize` function with an appropriate solver. In addition, the provided code extracts the relevant bounds and a suitable initial guess from the prompt.



(A) Initial Cabinet Design



(B) Optimized Cabinet



(C) Optimized Cabinet (Close-up)

Figure F43. Single objective with constraint optimization. **Left:** A cabinet with initial height, width, and depth of 55 in. **Middle:** Using an off-the-shelf solver provided by `scipy.optimize`, GPT-4 provides a Python script that optimizes the cabinet's height, width, and depth given a desired volume while minimizing the material cost. We show an optimized cabinet that meets the volume constraint of 5,000 cubic inches of storage capacity. **Right:** A close-up view of the optimized cabinet shown in the middle.

F.2.3. Robot Arm Planning

We now study a planning problem: given a claw attached to an arm and an environment with objects and bins, GPT-4 must control the arm-claw robot with a sequence of commands that efficiently picks up all objects and places them into bins. Each bin can only hold one object. In the arm-claw interface provided to GPT-4, the physical embodiment of the arm-claw robot does not matter; this allows GPT-4 to simply reason about the movement of the claw and whether the claw should grasp or release an object. Due to the nature of the problem, there is one critical constraint to consider: the claw must visit an object to pick it up before dropping it off at a bin. Formalizing this constraint is nontrivial, but the performance objective is much simpler: minimize the claw's travel distance. To simplify the problem, we also add that the maximum number of objects and bins is three each, making brute force a valid solution. The initial prompt and result are shown in [Figure F44](#).



You are tasked with controlling an arm and a claw. The goal is to pick up an object on the ground with the claw and place it in a bin large enough for the object to fit in. You are working with (x, y, z) coordinates where $0 \leq x, y, z \leq 50$. Below are functions you can use to control the arm and claw.

Arm:

- `translate_x(x_dist)`: Translates the claw in the x direction by `x_dist`.
- `translate_y(y_dist)`: Translates the claw in the y direction by `y_dist`.
- `translate_z(z_dist)`: Translates the claw in the z direction by `z_dist`.

Claw:

- `grasp()`: If there is an object at the claw's current location, the object will be grasped. When the object is grasped and the claw is moved, the object is moved to the same location as the claw's fingertips.
- `release()`: If the claw is grasping an object, the object will fall to the ground directly underneath it at $z=0$ or to any bin directly underneath it.

(... omitted by authors: additional problem specifications ...)



(... content omitted by authors ...)

```
def move_arm(claw_initial, objects_initial, bins, T_max):
    def distance(p1, p2):
        (... omitted by authors: Euclidean distance helper function ...)

    actions = []
    power_used = 0
    claw_current = list(claw_initial)
    while objects_initial and power_used <= T_max:
        # Find nearest object
        nearest_object = min(objects_initial, key=lambda o: distance(claw_current, o))

        # Calculate necessary translations to reach the object
        translations = [o - c for o, c in zip(nearest_object, claw_current)]

        # Check if there's enough power left to reach the object and drop it in a bin
        if power_used + sum(abs(t) for t in translations) * 2 > T_max:
            break

        # Move to the object
        (... omitted by authors: adding translations to object and grasping to the actions list ...)

        # Update claw position and power used
        claw_current = nearest_object
        power_used += sum(abs(t) for t in translations)

        # Find nearest bin
        nearest_bin = min(bins, key=lambda b: distance(claw_current, b))

        # Calculate necessary translations to reach the bin
        translations = [b - c for b, c in zip(nearest_bin, claw_current)]

        # Move to the bin and release the object
        (... omitted by authors: adding translations to bin and releasing to the actions list ...)

        # Update claw position and power used
        claw_current = nearest_bin
        power_used += sum(abs(t) for t in translations)

        # Remove the placed object and bin from the lists
        objects_initial.remove(nearest_object)
        bins.remove(nearest_bin)
    (... omitted by authors: code checking that a solution was found ...)
```

(... content omitted by authors ...)

Figure F44. Initial Proposed Solution to Robot Manipulation Planning. GPT-4 provides a greedy approach to the problem of minimizing the distance traveled by the claw, which does not guarantee an optimal solution. Overall, GPT-4 understands that it needs to keep track of the claw's position to compute the correct distances and that the claw should move to an object before moving to a bin. Still, it is unable to output an optimal

solution, even when the problem statement permits a brute force approach.

To address this, we explicitly emphasize that the output function should guarantee that the minimum distance is traveled. Even in this case, the optimal solution is not necessarily reached. As shown in [Figure F45](#), GPT-4's code fails to consider all possible bins that an object could be placed into once it has been picked up. However, we note that the solutions have high variance—on a different run, GPT-4 does produce a correct brute force solution. A third run produces code that guarantees an optimal solution but is inefficient, as it computes the translation for paths that do not obey the constraint that a claw must pick up an object before placing it in a bin.

Could you write a function that guarantees the global optimum? Make sure to account for the distances from the claw to object and claw to bin (after it has grasped the object). Also make sure to compute translations rather than Euclidean distance. Recall the bounds are $0 \leq n \leq 3$ and $0 \leq m \leq 3$.

(... content omitted by authors ...)

This updated function generates all permutations of the objects and bins, calculates the total translation required for each permutation, and selects the one with the smallest total translation.

```

from itertools import permutations

def move_arm(claw_initial, objects_initial, bins, T_max):
    def translation_distance(p1, p2):
        (... omitted by authors: Manhattan distance helper function ...)

    def calculate_total_translation(ordering):
        total_translation = 0
        claw_pos = list(claw_initial)
        for object_pos, bin_pos in ordering:
            total_translation += translation_distance(claw_pos, object_pos) # Claw to object
            total_translation += translation_distance(object_pos, bin_pos) # Object to bin
            claw_pos = bin_pos # Claw is now at bin location
        return total_translation

    object_bin_pairs = list(permutations(zip(objects_initial, bins)))
    min_translation = float('inf')
    best_ordering = None
    for ordering in object_bin_pairs:
        total_translation = calculate_total_translation(ordering)
        if total_translation < min_translation:
            min_translation = total_translation
            best_ordering = ordering

    if min_translation > T_max:
        return "No solution found within power limit"
    else:
        actions = []
        claw_current = list(claw_initial)
        for object_pos, bin_pos in best_ordering:
            # Move to object
            (... omitted by authors: adding translations to object and grasping to the actions list ...)
            claw_current = list(object_pos)

            # Move to bin
            (... omitted by authors: adding translations to object and grasping to the actions list ...)
            claw_current = list(bin_pos)
        return actions

```

(... content omitted by authors ...)

Figure F45. Attempt at brute force solution to manipulation planning. Though GPT-4 attempts to solve the minimum translation problem in a brute force manner, it fails to check all possible paths that the claw can take.

F.3. Navigating Trade-Offs Between Multiple Conflicting Objectives (Q3)

Although previous experiments focused on optimizing a single performance objective, we now explore the scenario where a user wishes to navigate a higher dimensional (i.e., multiobjective) performance space. The user begins by asking GPT-4 for reasonable performance metrics for evaluating a chair. After GPT-4 provides eight such metrics, our user purposefully selects stability and sustainability, since they can be mathematically quantified by tipping angle and volume, respectively. The user then asks for parameters over which to search.

Since GPT-4 has not been given a design template, GPT-4 proposes parameters abstractly; we note that it might have been more useful if GPT-4 first proposed a skeleton for the chair geometry, especially so that a user could understand the ramifications of these parameters. After iterating with GPT-4 to generate correct OpenSCAD code for the design, the user requests code to evaluate and optimize the chair. GPT-4 proposes the use of Non-dominated Sorting Genetic Algorithm II (NSGA-II)—a very common evolutionary method for computing the Pareto front of the multiobjective trade-off space—and provides code for the optimization. As an oversight, GPT-4 initially excludes design parameters bounds from the optimization, despite verbally providing ideas earlier in the conversation. When prompted to add the bounds into the optimization code, because of its limited memory, GPT-4 suggests reasonable, but notably different parameter bounds. Additionally, GPT-4 must be prompted again to enforce the bounds consistently throughout the algorithm (specifically, in the crossover and selection operators). Results can be found in [Figure F46](#).

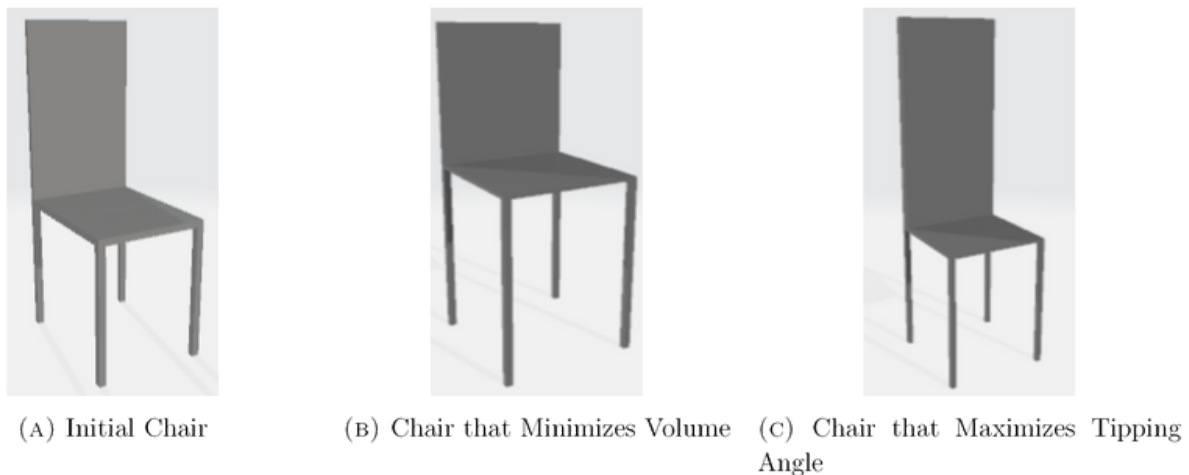


Figure F46. Optimizing chair design with competing objectives. GPT-4 outputs code that handles a multiobjective performance space in the context of designing a chair to minimize volume while maximizing tipping angle. **Left:** A randomly sampled chair from the initial population before Non-dominated Sorting Genetic Algorithm II (NSGA-II) is applied. After running NSGA-II using the script GPT-4 provides, we visualize chairs on opposite ends of the frontier. **Middle:** A chair that minimizes volume by reducing the volume of the back seat and increasing leg lengths. **Right:** A chair that maximizes tipping value by extending the length and increasing the thickness of the back seat.

Through this example, we conclude that GPT-4 has the potential to aid users in both a) understanding the trade-offs involved in different candidate designs, and b) providing pointers to a reasonable algorithm that can help navigate that space.

F.4. Supporting Optimization in Contexts that Require Additional Knowledge (Q4)

The chair example discussed in [Section 10.3](#) demonstrates GPT-4’s ability to recommend reasonable parameters for a design without needing explicit, low-level prompts from a user. Indeed, when prompted for ‘parameters,’ GPT-4 is able to apply its knowledge of the target domain to offer continuous parameterizations

of a typical chair (and provide a 3D model on request), along with reasonable ranges for each parameter. Although discrete parameters are possible with a chair, they are less likely to have a significant impact relative to its raw dimensions, and most chairs are comprised specifically of four legs, a seat, and a back.

©2024 Liane Makatura, Michael Foshey, Bohan Wang, Felix Hähnlein, Pingchuan Ma, Bolei Deng, Megan Tjandrasuwita, Andrew Spielberg, Crystal Owens, Peter Yichen Chen, Allan Zhao, Amy Zhu, Wil Norton, Edward Gu, Joshua Jacob, Yifei Li, Adriana Schulz, and Wojciech Matusik. This article is licensed under a [Creative Commons Attribution \(CC BY 4.0\) International license](#), except where otherwise indicated with respect to particular material included in the article.

Footnotes

1. We use the OpenAI ChatGPT interface to interact with the GPT-4 versions released between May 24, 2023, and July 19, 2023. We chose GPT-4 for our experiments after surveying other competing LLMs and finding that GPT-4 was far more useful in the context of CDaM than other competing LLMs at the time. ↵
2. Name chosen to be deliberately nonsensical so as not to give context clues to GPT-4. ↵

References

- Autodesk. (n.d.). *Autodesk simulation*. Retrieved July 14, 2023, from <https://www.autodesk.com/solutions/simulation/overview>
↵
- Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., . . . Zaremba, W. (2021). *Evaluating large language models trained on code*. ArXiv.
<https://doi.org/10.48550/arXiv.2107.03374>
↵
- Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S., & Amodei, D. (2017). Deep reinforcement learning from human preferences. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in neural information processing systems* (Vol. 30, pp. 4299–4307). Curran Associates.
https://papers.nips.cc/paper_files/paper/2017/hash/d5e2c0adad503c91f91df240d0cd4e49-Abstract.html
↵
- Dassault Systèmes. (n.d.). *Dassault Systèmes simulation*. Retrieved July 14, 2023, from <https://www.3ds.com/products-services/simulia/overview/>

↩

- Dhariwal, P., Jun, H., Payne, C., Kim, J. W., Radford, A., & Sutskever, I. (2020). *Jukebox: A generative model for music*. ArXiv. <https://doi.org/10.48550/arXiv.2005.00341> ↩
- Du, T., Inala, J. P., Pu, Y., Spielberg, A., Schulz, A., Rus, D., Solar-Lezama, A., & Matusik, W. (2018). InverseCSG: Automatic conversion of 3D models to CSG trees. *ACM Transactions on Graphics (TOG)*, 37(6), Article 213. <https://doi.org/10.1145/3272127.3275006> ↩
- Du, T., Wu, K., Ma, P., Wah, S., Spielberg, A., Rus, D., & Matusik, W. (2021). DiffPD: Differentiable projective dynamics. *ACM Transactions on Graphics (TOG)*, 41(2), Article 13. <https://doi.org/10.1145/3490168> ↩
- Du, Y., Li, S., Torralba, A., Tenenbaum, J. B., & Mordatch, I. (2023). *Improving factuality and reasoning in language models through multiagent debate*. ArXiv. <https://doi.org/10.48550/arXiv.2305.14325> ↩
- Erez, T., Tassa, Y., & Todorov, E. (2015). Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX. In N. Amato (Ed.), *Proceedings of the 2015 IEEE International Conference on Robotics and Automation* (pp. 4397–4404). IEEE. <https://doi.org/10.1109/ICRA.2015.7139807> ↩
- Erps, T., Foshey, M., Luković, M. K., Shou, W., Goetzke, H. H., Dietsch, H., Stoll, K., von Vacano, B., & Matusik, W. (2021). Accelerated discovery of 3D printing materials using data-driven multiobjective optimization. *Science Advances*, 7(42), Article eabf7435.
Erps, T., Foshey, M., Luković, M. K., Shou, W., Goetzke, H. H., Dietsch, H., Stoll, K., von Vacano, B., & Matusik, W. (2021). Accelerated discovery of 3D printing materials using data-driven multiobjective optimization. *Science Advances*, 7(42), Article eabf7435. <https://doi.org/10.1126/sciadv.abf7435>

↩

- Featurescript introduction. (n.d.). Retrieved July 11, 2023, from <https://cad.onshape.com/FsDoc/>

↩

- Ferruz, N., Schmidt, S., & Höcker, B. (2022). ProtGPT2 is a deep unsupervised language model for protein design. *Nature Communications*, 13(1), Article 4348. <https://doi.org/10.1038/s41467-022-32007-7> ↩
- Guo, M., Thost, V., Li, B., Das, P., Chen, J., & Matusik, W. (2022). Data-efficient graph grammar learning for molecular generation. *arXiv Preprint arXiv:2203.08031*. ↩
- Guo, M., Thost, V., Li, B., Das, P., Chen, J., & Matusik, W. (2022). *Data-efficient graph grammar learning for molecular generation*. ArXiv. <https://doi.org/10.48550/arXiv.2203.08031> ↩
- Jiang, B., Chen, X., Liu, W., Yu, J., Yu, G., & Chen, T. (2023). *MotionGPT: Human motion as a foreign language*. ArXiv. <https://doi.org/10.48550/arXiv.2306.14795> ↩
- JSCAD user guide. (n.d.). Retrieved July 14, 2023, from <https://openjscad.xyz/dokuwiki/doku.php>

↩

- Kashefi, A., & Mukerji, T. (2023). Chatgpt for programming numerical methods. *Journal of Machine Learning for Modeling and Computing*, 4(2), 1–74.
<https://doi.org/10.1615/JMachLearnModelComput.2023048492> ↵
- Koo, B., Hergel, J., Lefebvre, S., & Mitra, N. J. (2017). Towards zero-waste furniture design. *IEEE Transactions on Visualization and Computer Graphics*, 23(12), 2627–2640.
<https://doi.org/10.1109/TVCG.2016.2633519> ↵
- Li, J., Rawn, E., Ritchie, J., Tran O’Leary, J., & Follmer, S. (2023). Beyond the artifact: Power as a lens for creativity support tools. In Follmer, S., Han, J., Steimle, J., & Riche, N. H. (Eds.), *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (Article 47). Association for Computing Machinery. <https://doi.org/10.1145/3586183.3606831>
↵
- Liu, R., Wu, R., Van Hoorick, B., Tokmakov, P., Zakharov, S., & Vondrick, C. (2023). *Zero-1-to-3: Zero-shot one image to 3D object*. ArXiv. <https://doi.org/10.48550/arXiv.2303.11328> ↵
- Ma, P., Du, T., Zhang, J. Z., Wu, K., Spielberg, A., Katzschmann, R. K., & Matusik, W. (2021). DiffAqua: A differentiable computational design pipeline for soft underwater swimmers with shape interpolation. *ACM Transactions on Graphics (TOG)*, 40(4), Article 132. <https://doi.org/10.1145/3450626.3459832> ↵
- Makatura, L., Wang, B., Chen, Y.-L., Deng, B., Wojtan, C., Bickel, B., & Matusik, W. (2023). Procedural metamaterials: A unified procedural graph for metamaterial design. *ACM Transactions on Graphics*, 42(5), Article 168. <https://doi.org/10.1145/3605389> ↵
- Mathur, A., & Zufferey, D. (2021). Constraint synthesis for parametric CAD. In M. Okabe, S. Lee, B. Wuensche, & S. Zollmann (Eds.), *Pacific Graphics 2021: The 29th Pacific Conference on Computer Graphics and Applications: Short Papers, Posters, and Work-in-Progress Papers* (pp. 75–80). The Eurographics Association. <https://doi.org/10.2312/pg.20211396>
↵
- Mirchandani, S., Xia, F., Florence, P., Ichter, B., Driess, D., Arenas, M. G., Rao, K., Sadigh, D., & Zeng, A. (2023). *Large language models as general pattern machines*. ArXiv.
<https://doi.org/10.48550/arXiv.2307.04721> ↵
- Müller, P., Wonka, P., Haegler, S., Ulmer, A., & Van Gool, L. (2006). Procedural modeling of buildings. In *ACM SIGGRAPH 2006 papers* (pp. 614–623). ↵
- Müller, P., Wonka, P., Haegler, S., Ulmer, A., & Van Gool, L. (2006). Procedural modeling of buildings. In *ACM SIGGRAPH 2006 papers* (pp. 614–623). Association for Computing Machinery.
<https://doi.org/10.1145/1179352.1141931> ↵
- Ni, B., & Buehler, M. J. (2024). MechAgents: Large language model multi-agent collaborations can solve mechanics problems, generate new data, and integrate knowledge. *Extreme Mechanics Letters*, 67, Article 102131. <https://doi.org/10.1016/j.eml.2024.102131> ↵

- O'Brien, J. F., Shen, C., & Gatchalian, C. M. (2002). Synthesizing sounds from rigid-body simulations. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (pp. 175–181). Association for Computing Machinery. <https://doi.org/10.1145/545261.545290> ↵
- OpenAI. (2023). *GPT-4 technical report*. ArXiv. <https://doi.org/10.48550/arXiv.2303.08774> ↵
- OpenAI. (2023). *GPT-4 technical report*. <https://arxiv.org/abs/2303.08774>. <https://arxiv.org/abs/2303.08774> ↵
- Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., & Lowe, R. (2022). Training language models to follow instructions with human feedback. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, & A. Oh (Eds.), *Advances in Neural Information Processing Systems* (Vol. 35, pp. 27730–27744). Curran Associates. https://proceedings.neurips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html ↵
- Özkar, M., & Stiny, G. (2009). Shape grammars. In *ACM SIGGRAPH 2009 courses* (Article 22). Association for Computing Machinery. <https://doi.org/10.1145/1667239.1667261> ↵
- Penedo, G., Malartic, Q., Hesslow, D., Cojocaru, R., Cappelli, A., Alobeidli, H., Pannier, B., Almazrouei, E., & Launay, J. (2023). *The RefinedWeb dataset for Falcon LLM: Outperforming curated corpora with web data, and web data only*. ArXiv. <https://doi.org/10.48550/arXiv.2306.01116> ↵
- Prusinkiewicz, P., & Lindenmayer, A. (1990). *The algorithmic beauty of plants*. Springer Science & Business Media. ↵
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019, February 14). *Language models are unsupervised multitask learners*. OpenAI. <https://openai.com/index/better-language-models/> ↵
- Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., & Sutskever, I. (2021). Zero-shot text-to-image generation. *Proceedings of Machine Learning Research*, 139, 8821–8831. <https://proceedings.mlr.press/v139/ramesh21a.html> ↵
- Repetier. (n.d.). *Repetier software*. Retrieved July 20, 2023, from <https://www.repetier.com/> ↵
- Richards, T. B. (n.d.). *AutoGPT*. Retrieved February 11, 2024, from <https://github.com/Significant-Gravitas/AutoGPT> ↵
- Rozenberg, G., & Salomaa, A. (1980). *The mathematical theory of L systems*. Academic press. ↵

- Slic3r. (n.d.). *Slic3r - Open source 3D printing toolbox*. Retrieved July 20, 2023, from <https://slic3r.org/>
[↩](#)
- Stiny, G. (1980). Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7(3), 343–351. <https://doi.org/10.1068/b070343> [↩](#)
- Sullivan, D. M. (2013). *Electromagnetic simulation using the FDTD method*. John Wiley & Sons.
<https://doi.org/10.1002/9781118646700> [↩](#)
- Todorov, E., Erez, T., & Tassa, Y. (2012). MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 5026–5033). IEEE.
<https://doi.org/10.1109/IROS.2012.6386109> [↩](#)
- Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1), 230–265. <https://doi.org/10.1112/plms/s2-42.1.230> [↩](#)
- Willis, K. D., Pu, Y., Luo, J., Chu, H., Du, T., Lambourne, J. G., Solar-Lezama, A., & Matusik, W. (2021). Fusion 360 gallery: A dataset and environment for programmatic CAD construction from human design sequences. *ACM Transactions on Graphics (TOG)*, 40(4), Article 54.
<https://doi.org/10.1145/3450626.3459818> [↩](#)
- Xu, J., Chen, T., Zlokapa, L., Foshey, M., Matusik, W., Sueda, S., & Agrawal, P. (2021). *An end-to-end differentiable framework for contact-aware robot design*. ArXiv. <https://doi.org/10.48550/arXiv.2107.07501>
[↩](#)
- Yu, L., Shi, B., Pasunuru, R., Muller, B., Golovneva, O., Wang, T., Babu, A., Tang, B., Karrer, B., Sheynin, S., Ross, C., Polyak, A., Howes, R., Sharma, V., Xu, P., Tamoyan, H., Ashual, O., Singer, U., . . . Aghajanyan, A. (2023). *Scaling autoregressive multi-modal models: Pretraining and instruction tuning*. ArXiv. <https://doi.org/10.48550/arXiv.2309.02591>
[↩](#)
- Zhang, Y., Yang, M., Baghdadi, R., Kamil, S., Shun, J., & Amarasinghe, S. (2018). Graphit: A high-performance graph DSL. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), Article 121.
<https://doi.org/10.1145/3276491> [↩](#)
- Zhao, A., Xu, J., Konaković-Luković, M., Hughes, J., Spielberg, A., Rus, D., & Matusik, W. (2020). Robogrammar: Graph grammar for terrain-optimized robot design. *ACM Transactions on Graphics (TOG)*, 39(6), Article 188. <https://doi.org/10.1145/3414685.3417831> [↩](#)
- Zilberstein, S. (1996). Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3), 73–73.
<https://doi.org/10.1609/aimag.v17i3.1232> [↩](#)