



CAD-Assistant: Tool-Augmented VLLMs as Generic CAD Task Solvers

Dimitrios Mallis¹

dimitrios.mallis@uni.lu

Danila Rukhovich¹

danila.rukhovich@uni.lu

Ahmet Serda Karadeniz¹

ahmet.karadeniz@uni.lu

Sebastian Cavada¹

sebastian.cavada.dev@gmail.com

Niki Foteinopoulou¹

niki.foteinopoulou@uni.lu

Kseniya Cherenkova^{1,2}

kseniya.cherenkova@uni.lu

Anis Kacem¹

anis.kacem@uni.lu

Djamila Aouada¹

djamila.aouada@uni.lu

¹SnT, University of Luxembourg ²Artec3D, Luxembourg

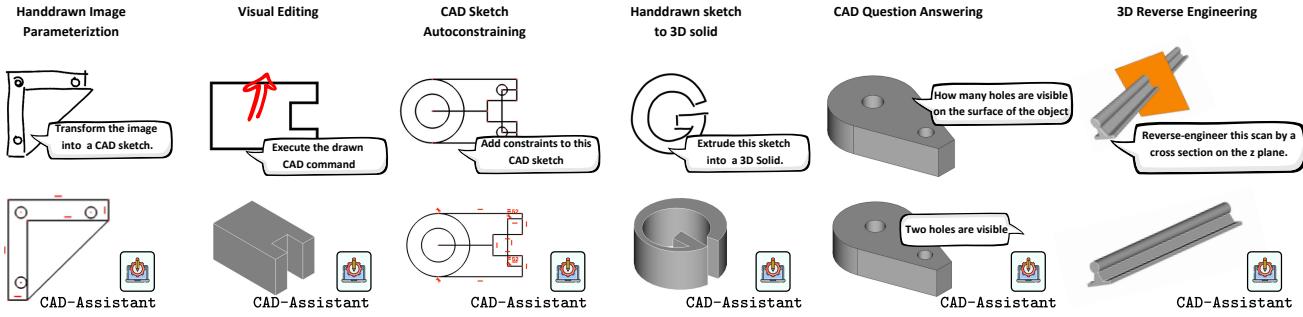


Figure 1. CAD-Assistant is a tool-augmented VLLM framework for AI-assisted CAD. Our framework generated FreeCAD code that is executed within CAD software directly and can process multimodal inputs, including textual queries, sketches, drawn commands and 3D scans. This figure showcases various examples of generic CAD queries and the responses generated by CAD-Assistant.

Abstract

We propose CAD-Assistant, a general-purpose CAD agent for AI-assisted design. Our approach is based on a powerful Vision and Large Language Model (VLLM) as a planner and a tool-augmentation paradigm using CAD-specific tools. CAD-Assistant addresses multimodal user queries by generating actions that are iteratively executed on a Python interpreter equipped with the FreeCAD software, accessed via its Python API. Our framework is able to assess the impact of generated CAD commands on geometry and adapts subsequent actions based on the evolving state of the CAD design. We consider a wide range of CAD-specific tools including a sketch image parameterizer, rendering modules, a 2D cross-section generator, and other specialized routines. CAD-Assistant is evaluated on multiple CAD benchmarks, where it outperforms VLLM baselines and supervised task-specific methods. Beyond existing benchmarks, we qualitatively demonstrate the po-

tential of tool-augmented VLLMs as general-purpose CAD solvers across diverse workflows.

1. Introduction

Computer-Aided Design (CAD) refers to the use of computer software to assist in the creation, modification, analysis, or optimization of a design [5]. CAD is crucial for enabling precise, efficient, and scalable design processes across industries. Recently, there has been a significant research interest in the automation of CAD pipelines. Examples include, 3D reverse-engineering [13, 23, 37, 56], CAD generation [47, 49, 59, 62], edge parametrization [9, 73], CAD from multiview images [17, 69], hand-drawn CAD sketch parametrization [21, 22] and text-guided CAD editing [26]. Still, most efforts to date have centered around fixed workflows, and the development of CAD agents to address generic tasks remains largely unexplored. In this work, we advocate for the creation of CAD agents capa-

ble of interacting with and supporting designers through the CAD process. This would be a transformative advancement for the CAD industry streamlining time-consuming work.

As Vision and Large Language Models (VLLMs) continue to mature [1, 3, 12, 28, 31, 32, 38, 40], they hold promise for enabling AI-assisted CAD design, particularly given that their very vast pre-training endow them with broad knowledge of design and manufacturing [36]. Despite the identified potential, their ability to be used within computational design and manufacturing workflows remains severely constrained by weaknesses in geometric reasoning and handling of mathematical concepts [19]. Indeed, VLLMs may struggle to semantically interpret the appearance of rendered objects from their corresponding CAD sequences [45]. They may also fail to recognize spatial arrangements and the varied combinations of visual concepts [50] or correctly orient primitives and generate accurate placements [36]. Their effectiveness in an agentic CAD setting is further hindered by the inherently unpredictable effects of CAD commands. High-level CAD operations, such as applying geometric constraints, fillet, chamfer, etc., can have complex and non-intuitive impacts on a model’s geometry and topology [48, 49], which is typically resolved by advanced CAD solvers. VLLMs cannot reliably predict the cumulative effects of the CAD commands they generate further limiting their practical usability in CAD workflows.

Recently, tool-augmentation has emerged as a prevailing strategy for addressing various shortcomings of foundational models and enhancing their performance in real-world applications [19, 34, 51, 55, 57]. Despite demonstrated effectiveness, VLLMs capable of composing and utilizing external tools have yet to be explored within the domain of CAD design. This work addresses this gap by introducing **CAD-Assistant**, a generic tool-augmented VLLM framework that integrates CAD-specific tools to effectively address the limitations of VLLMs in AI-assisted CAD. **CAD-Assistant** integrates a wide range of external CAD-specific modules, including a hand-drawn image parameterizer, rendering modules for multimodal CAD sequence understanding, a specialized utility for analysis of geometric constraints and a 2D cross-section generator for VLLM interaction with 3D scans.

Our framework leverages a VLLM-based planner and CAD-specific tool augmentation for generic CAD task solving. The planner generates CAD code actions, that are executed directly within the open-source CAD software FreeCAD [11], accessed via its Python API. Geometric reasoning is enhanced by dedicated CAD rendering and parameter serialization modules, enabling a more comprehensive multimodal representation of CAD models throughout the planning and reasoning process. Instead of solely relying on the effect prediction of complex CAD commands, our CAD agent inspects the evolving state of a design and refines or

corrects actions based on the current CAD geometry. CAD-specific tools facilitate the processing of multimodal inputs, from text to hand-drawn sketches, precise CAD drawings, drawn commands and 3D scans.

CAD-Assistant is a *training-free* framework that generates CAD code on an open-source CAD API, producing outputs that are both editable and highly interpretable. **CAD-Assistant** is also highly extensible and can operate across the diverse set of commands available in the FreeCAD API, requiring only a Python docstring to incorporate further capabilities. This is in contrast to the majority of CAD automation research focusing on the limited set of CAD operations captured in large-scale CAD datasets [29, 59, 63, 64]. To address the lack of benchmarks for tool use akin to specialized sets commonly used in other domains [33, 35], this work adopts an evaluation setting for generic CAD agents leveraging multiple existing CAD tasks. Evaluations are conducted for 2D and 3D CAD question answering, auto-constraining, and hand-drawn CAD sketch image parametrization. **CAD-Assistant** outperforms both VLLM baselines and supervised task-specific methods trained on large-scale datasets, despite being prompted in a zero-shot manner. Furthermore, we demonstrate the potential of **CAD-Assistant** beyond existing benchmarks by showcasing diverse use cases, including generating 3D solids from hand-drawn sketches, performing 3D reverse engineering from 3D scans via cross-section parameterization, and visual CAD design through semantically interpretable drawing commands (*e.g.* sketching an extrusion operation). Example responses of the proposed **CAD-Assistant** on diverse multimodal queries are depicted in Figure 1.

Contributions: The main contributions of this work can be summarized as follows:

1. We introduce **CAD-Assistant**, the first tool-augmented VLLM framework for generic CAD task solving. Our framework is equipped with a diverse set of CAD-specific tools and can process multimodal inputs, including hand-drawn sketches and 3D scans.
2. We demonstrate the effectiveness of tool-use for mitigating VLLMs’ limitations on AI-assisted CAD. Geometric reasoning is enhanced by incorporating comprehensive multimodal representations of CAD models and enabling direct interaction with CAD software.
3. We propose a highly extensible and training-free framework that can operate beyond the simple set of CAD commands captured on existing CAD datasets.
4. We identify an evaluation setting for generic CAD agents based on existing benchmarks. The proposed zero-shot method outperforms baselines and task-specific approaches trained on large datasets. We also qualitatively demonstrate the potential of **CAD-Assistant** on a diverse set of real-world use cases.

The rest of the paper is organized as follows; Section 2 reviews the related works. CAD-Assistant framework is described in Section 3 and Section 4 presents conducted evaluations on CAD benchmarks and provides further analyses. Conclusions are provided in Section 5.

2. Related Work

Foundation Models for CAD: Recently, there has been increasing research interest in the use of foundation models on CAD-related applications. CAD-Talk [70] introduces a framework for semantic CAD code captioning using multi-view photorealistic renderings of CAD models along with part-segmentation, powered by foundation models [8, 25]. Taking a similar path, QueryCAD [24] proposes an open-vocabulary CAD part segmentation from images leveraging segmentation foundation models and LLMs to perform CAD-related question-answering for robotic applications. CADLLM [61] proposes a T5 model [46] finetuned on the SketchGraphs [48] dataset of 2D CAD sketches for sketch auto-completion. CadVLM [60] extends CADLLM [61] to the visual domain, incorporating a visual modality for CAD sketch auto-completion, autoconstraining and image-guided generation. CADReparam [26] uses VLLMs to infer meaningful variation spaces for parametric CAD models, re-parameterizing them to enable exploration along design-relevant axes. Img2CAD [69] utilizes a VLLM to reverse engineer objects from images, predicting the specific CAD command types needed to model each part of the object accurately. Badagabettu *et al.* [4], focus on text-guided generation of CAD models as CADQuery code, while LLM4CAD [30] use a similar approach to generate 3D CAD models from text and image inputs. Related to ours is the training-free method of [2] focusing on CAD model generation. Authors introduce a verification process to ensure the validity of generated models, but do not explore tool augmentation. Our investigation diverges from these task-specific approaches as it shifts the focus on tool-augmentation for mitigating the limitation of VLLMs on AI-assisted CAD. CAD-Assistant is the first *general-purpose* framework for CAD design, able to process multimodal prompts and address diverse CAD use cases on a zero-shot manner.

Tool-augmented VLLMs: The creation and use of tools to overcome physical limitations serve as a clear demonstration of human intelligence [52]. Driven by the advanced reasoning and planning capabilities of LLMs and VLLMs, there has been growing interest in enhancing their performance via augmentation of external tools [16, 19, 34, 51, 55, 57, 67, 71]. The field is further propelled by the emergence of benchmarks, namely ScienceQA [33] and TabMWP [35], which are well-suited for evaluating the effectiveness of tool-use. Tool-use offers several ben-

efits [44], such as reducing hallucinated knowledge [53], providing real-time information [34], enhancing domain expertise [39] and producing interpretable outputs by making intermediate steps explicit [16, 55]. Planning is commonly performed via instructions in natural language [16, 34] or Python code generation [19, 55], and tool set might include search engines [27, 34, 39], calculators [10, 42], external APIs [43], vision modules [16, 55], Hugging Face models [51], Azure models [67] or LLM created tools [7]. Despite the vast potential of tool-augmented LLMs and VLLMs for CAD-related applications, the space remains unexplored. To our knowledge, this work is the first investigation on tool-augmented VLLMs for AI-assisted CAD.

VLLMs as Geometrical Reasoners: In order to advance tool-augmented VLLMs for AI-assisted CAD, it is crucial for the VLLMs planner to semantically recognize and precisely identify and manipulate individual elements within parametric geometries. This type of precision is an essential skill when interfacing with CAD software. Naturally, this raises the question: *Can large vision language models understand symbolic graphics programs?* In that direction, Yi *et al.* [68] explored incorporating symbolic structure as prior knowledge for enhancing visual question answering. More recently, Sharma *et al.* [50] examined visual program generation and recognition, showing that while shape generation often relies on memorizing prototypes from training data, shape recognition demands a deeper understanding of primitives. Qi *et al.* [45] introduced SGPBench, a question-answering benchmark designed to assess the semantic understanding and consistency of symbolic graphics programs, including CAD models. This benchmark evaluates the extent of LLMs' ability to semantically comprehend and reason about geometric structures. While [45] applied instruction tuning to improve visual program understanding, our work emphasizes general-purpose VLLMs, demonstrating that factors like serialization and parametrization strategies for formatting geometry and multimodal representation of a CAD model can significantly expand VLLMs' capacity for geometric reasoning.

3. The proposed CAD-ASSISTANT

3.1. General Framework

This section provides an overview of CAD-ASSISTANT. Our framework comprises the following three components:

Planner: The planner \mathcal{P} is modelled by a VLLM capable of advanced reasoning. Following [19], on each timestep t , the planner analyses the current context c_t and generates a plan p_t and an action a_t that implements p_t . In this work, we employ GPT-4o [40] as the core framework planner.

Environment: We utilize the Python interpreter as the primary environment \mathcal{E} for executing the generated action a_t

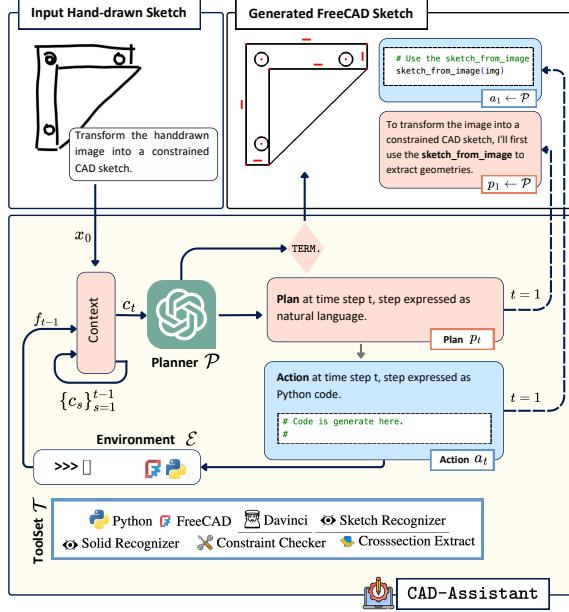


Figure 2. Overview of CAD-Assistant framework. A multimodal user request is provided as context to a VLLM planner \mathcal{P} that responds with a plan p_t and an action a_t (*python code*). The action is executed on an environment \mathcal{E} equipped with CAD software. The output generated from the execution is then concatenated with the previously generated context and fed back to the planner, enabling the generation of the next iterative step. at time t . Additionally, \mathcal{E} integrates CAD software [11] as a foundational component for AI-assisted CAD applications. On each timestep, t , the environment provides feedback e_t of the current state of the CAD design.

Tool Set: CAD-ASSISTANT utilizes a set $\mathcal{T} = \{\mathcal{T}_i\}_{i=1}^N$ of N CAD-specific tools, suitable for AI-Assisted CAD. These include standard Python libraries, modules of the FreeCAD Python API [11] to interface CAD commands, and other useful CAD-specific tools and Python routines. CAD-ASSISTANT can be formalized as follows: Given a multimodal x_0 input user query, on each timestep t , the planner \mathcal{P} generates:

$$p_t \leftarrow \mathcal{P}(x_0; c_{t-1}, \mathcal{T}), \quad (1)$$

$$a_t \leftarrow \mathcal{P}(p_t; c_{t-1}, x_0, \mathcal{T}), \quad (2)$$

where p_t is the current plan in natural language, and a_t is the current action formulated as Python code. Then, the generated action a_t is executed on the framework's environment:

$$(f_t, e_t) \leftarrow \mathcal{E}(a_t; e_{t-1}, \mathcal{T}, x_0), \quad (3)$$

where f_t is the output of the code execution, and e_t is the new state of the CAD design. Note that f_t can include both textual and visual outputs of the execution, *e.g.* list of CAD geometries in .json format or the rendering of the current state of the CAD object. Finally, the context is updated as:

$$c_{t+1} \leftarrow \text{concat}(f_t, \{c_s\}_{s=1}^t), \quad (4)$$

USER QUERY: Analyse the sketch to identify geometric relationships and apply parametric constraints.

PLAN: To analyze the sketch and apply the appropriate parametric constraints, I will first use the *sketch_recognizer* function to understand the current state of the sketch. Then I will use the *analyse_constraints* function to evaluate their validity and impact on the sketch before applying them.

ACTION: Use the *sketch_recognizer* function to analyze the current state of the sketch.

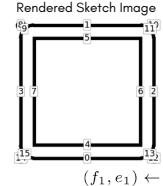
```
# Use the sketch_recognizer function to analyze the current state of the sketch
sketch_recognizer(sketch)
```

$$p_1 \leftarrow \mathcal{P}$$

ENVIRONMENT: Execution Success

The sketch contains the following elements, serialized in JSON format:

```
[{"Geometry": [{"Type": "Line", "StartPoint": "0", "EndPoint": "1", "IsConstruction": "false"}, {"Type": "Line", "StartPoint": "1", "EndPoint": "2", "IsConstruction": "false"}, {"Type": "Line", "StartPoint": "2", "EndPoint": "3", "IsConstruction": "false"}, {"Type": "Line", "StartPoint": "3", "EndPoint": "4", "IsConstruction": "false"}, {"Type": "Line", "StartPoint": "4", "EndPoint": "5", "IsConstruction": "false"}, {"Type": "Line", "StartPoint": "5", "EndPoint": "6", "IsConstruction": "false"}, {"Type": "Line", "StartPoint": "6", "EndPoint": "7", "IsConstruction": "false"}, {"Type": "Line", "StartPoint": "7", "EndPoint": "8", "IsConstruction": "false"}, {"Type": "Line", "StartPoint": "8", "EndPoint": "9", "IsConstruction": "false"}, {"Type": "Line", "StartPoint": "9", "EndPoint": "10", "IsConstruction": "false"}, {"Type": "Line", "StartPoint": "10", "EndPoint": "11", "IsConstruction": "false"}, {"Type": "Line", "StartPoint": "11", "EndPoint": "12", "IsConstruction": "false"}, {"Type": "Line", "StartPoint": "12", "EndPoint": "0", "IsConstruction": "false"}, {"Type": "Arc", "StartAngle": -0.4, "EndAngle": 0.4, "Center": "5", "Radius": 0.5, "IsConstruction": "true"}, {"Type": "Arc", "StartAngle": 0.4, "EndAngle": -0.4, "Center": "5", "Radius": 0.5, "IsConstruction": "true"}], "Constraints": []}
```



$$(f_1, e_1) \leftarrow \mathcal{E}$$

PLAN: The sketch consists of several line segments and arcs forming a square with rounded corners. To ensure the sketch is fully constrained, I will apply the following constraints:

1. Coincident constraints to connect endpoints of adjacent geometries.
2. Horizontal and Vertical constraints for the straight line segments.
3. Equal constraints for the arcs to ensure they have the same radius.
4. Tangent constraints between the arcs and the line segments.

I will first analyse and apply the Coincident constraints.

$$p_2 \leftarrow \mathcal{P}$$

ACTION: Analyze and apply Coincident constraints to connect endpoints of geometries.

```
# Define Coincident constraints for connecting endpoints
coincident_constraints = [
    Sketcher.Constraint('Coincident', 0, SubRef.END.value, 15, SubRef.START.value),
    Sketcher.Constraint('Coincident', 0, SubRef.END.value, 13, SubRef.START.value),
    Sketcher.Constraint('Coincident', 1, SubRef.START.value, 9, SubRef.END.value),
    Sketcher.Constraint('Coincident', 1, SubRef.END.value, 11, SubRef.START.value),
    Sketcher.Constraint('Coincident', 2, SubRef.END.value, 11, SubRef.START.value),
    Sketcher.Constraint('Coincident', 2, SubRef.END.value, 13, SubRef.START.value),
    Sketcher.Constraint('Coincident', 3, SubRef.START.value, 15, SubRef.START.value),
    Sketcher.Constraint('Coincident', 3, SubRef.END.value, 9, SubRef.START.value),
    Sketcher.Constraint('Coincident', 4, SubRef.START.value, 7, SubRef.END.value),
    Sketcher.Constraint('Coincident', 4, SubRef.END.value, 6, SubRef.START.value),
    Sketcher.Constraint('Coincident', 5, SubRef.START.value, 7, SubRef.START.value),
    Sketcher.Constraint('Coincident', 5, SubRef.END.value, 6, SubRef.END.value)
]
# Analyze and apply constraints
for i, constraint in enumerate(coincident_constraints):
    is_valid, moves_geometries = analyse_constraint(sketch, constraint)
    if is_valid and not moves_geometries:
        sketch.addConstraint(constraint)
sketch.recompute()
```

$$a_2 \leftarrow \mathcal{P}$$

ENVIRONMENT: Execution Success

$$e_2 \leftarrow \mathcal{E}$$

PLAN: The Coincidents have been applied, ensuring that the loops are closed. Next, I will apply Horizontal and Vertical constraints to the straight line segments to maintain their orientation.

$$p_3 \leftarrow \mathcal{P}$$

ACTION: Apply Horizontal and Vertical constraints to the line segments.

```
# Define Horizontal and Vertical constraints
# Rest of the code here.
```

$$a_3 \leftarrow \mathcal{P}$$

PLAN: The sketch is now properly constrained.

$$TERMINATE.$$

$$p_5 \leftarrow \mathcal{P}$$

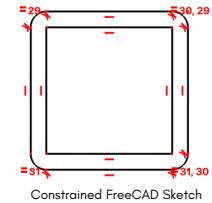


Figure 3. Execution flow for autoconstraining. CAD-Assistant utilizes the *sketch_recognizer* function for multimodal CAD understanding and generates constraints in a chain-of-thought manner.

concatenating the previous context with the current code execution output and is supplied to \mathcal{P} for plan generation of timestep $t + 1$. This process iterates for an arbitrary T number of steps until the planner \mathcal{P} concludes that the request

Module Type	Module Description
Python	Action format and logical operations.
FreeCAD	Integration with CAD software [11].
Sketch Parameterizer	Hand-drawn sketch image to CAD sketch based via [21].
Sketch Recognizer	Renders sketch and plots parameters.
Solid Recognizer	Renders a 3D CAD model and plots parameters.
Constraint Checker	Analyzes geometric constraints.
Crosssection Extract	Generates an image of a cross section from a 3D mesh.

Table 1. Overview of CAD-specific tools.

x_0 has been successfully addressed. At that point, \mathcal{P} generates p_T , a special TERMINATE plan that indicates the completion of CAD-ASSISTANT’s response. An illustration of the proposed execution flow is provided in Figure 2 and an example of the agents’ trajectory as a response to an auto-constraining request is provided in Figure 3.

3.2. CAD-specific Tool-set

CAD-ASSISTANT includes a set of N CAD-specific tools or modules. Each tool is defined by its method signature and the docstring [18] that disambiguates its use. Note that \mathcal{T}_i contains only the module signature rather than the full method code as in [19, 55] to minimize context size and dependency on module details. Modules \mathcal{T}_i are instantiated via their Python interface with arguments generated by \mathcal{P} as part of the action a_t . Notably, actions are formulated as Python code, as in [19, 55], rather than the natural-language instructions advocated by recent works [16, 34]. This design choice allows for direct use of the FreeCAD API. Moreover, the generated action a_t can access the parameters of the CAD models’ state e_t and perform logical and computational operations, which is highly advantageous for design tasks (see also section 10 of supplementary). Our CAD-specific tool set is summarized in Table 1 and a detailed overview of each tool is provided in supplementary.

4. Experiments

This section outlines the experiments conducted to validate the effectiveness of CAD-Assistant. First, we explore various strategies for effective CAD geometric reasoning capabilities of VLLMs. Next, we describe the CAD benchmarks and experimental settings used for evaluation and present the performance results of CAD-Assistant on these benchmarks. Finally, we showcase new capabilities enabled by CAD-Assistant for real-world applications.

4.1. Strategies for Effective Geometric Reasoning

Effective geometric reasoning is an essential requirement for the development of generic CAD agents. However, VLLMs have shown limited ability to geometrically comprehend and mathematically reason about CAD programs [45, 50]. Previous work has explored symbolic instruction tuning [45] for addressing this limitation. In contrast, we shift our focus on tool augmentation as a training-free alternative to enhance geometric reasoning. The study in this subsection examines CAD representations that can be derived using external tools and improve VLLMs’ understanding of CAD programs. Specifically, we study the following factors of a CAD representation:

Parametrization Strategy: Parametric geometries can be represented by different sets of parameters. For instance, a line could use start / end points or an angle and length relative to a reference. We compare the implicit parametrization approach of [45] to the point-based primitive representation of [22]. We also explore over-parametrization, where a redundant set of parameters is used per geometry. More details about this comparison are provided on supplementary.

Serialization Strategy: The serialization format used to convert the parametric geometry into text can impact the planner’s ability to understand the geometry. Motivated by recent work on text-based serialization methods for tabular data [14], we compare commonly used formats such as *CSV*, *Markdown*, *HTML*, and *JSON*.

Rendering-based Reasoning: We investigate visual representations for geometric reasoning by providing the VLLM planner with 2D renderings of the CAD sketch or 3D solid.

To examine the impact of the above strategies on CAD program understanding and geometric reasoning, we experiment on the CAD question answering benchmark SGP-Bench [45]. This benchmark comprises multiple-choice questions and captures three types of graphical programs, *i.e.*, *SVG*, CAD sketches, and 3D CAD models. For this experiment, we report accuracy on the 2D CAD subset. This subset is derived from 700 CAD sketches from Sketch-Graphs [48]. A VLLM planner (GPT-4o) is provided with a textual description of a 2D CAD sketch and tasked with answering a multiple-choice question about the design.

In Table 2 (**top**), we analyze the effect on the performance of the parametrization and serialization strategies used to parse the CAD sketch into a textual format. Firstly, we observe that schema-embedded representation like *JSON* performs better than tabular formats. Note that this is in contrast with recent work [54], where *HTML* was identified as the optimal serialization for tabular data. Secondly, GPT-4o demonstrates high sensitivity to geometry parametrization. The implicit parametrization used in SGPBench [45] significantly under-performs compared to a point-based parametrization for geometric primitives as in [22]. Overall, using a *JSON* serialization along with

2D CAD SGPBench - Sketch in Textual Format		
Serialization	Parametrization	Acc
<i>SGPBench [45] format</i>		
Serialized Graph	Implicit	0.674
<i>Standardized CAD Sketch formats</i>		
DXF [20]		0.671
OCA [15]		0.707
<i>Serialization Strategy (Tabular formats)</i>		
CSV	Point-based	0.703
Markdown	Point-based	0.706
HTML	Point-based	0.710
<i>Serialization Strategy (Schema-embedded formats)</i>		
Serialized Graph	Point-based	0.744
JSON	Point-based	0.748
<i>Parametrization Strategy</i>		
JSON	Point-based	0.748
JSON	Overparametrized	0.747

2D CAD SGPBench - Sketch as a Rendering		
CAD Sketch Image Type	Acc	
Hand-drawn Sketch	0.616	
Precise Rendering	0.754	

Table 2. Investigation of prompting strategies on geometric reasoning. We report performance for GPT-4o in terms of accuracy on the 2D partition of SGPBench [45]. (**Top**) Impact of Parametrization and serialization on CQA performance. (**Bottom**) Performance from hand-drawn and precise rendering of a CAD sketch.

the point-based parametrization from [22] leads to substantial improvements over the original SGPBench format and other text-based CAD sketch formats, such as *DXF* and *OCA*. While over-parameterizing the sketches results in a negligible drop in performance w.r.t. a point-based parameterization, we argue that it is safer to opt for over-parameterization as other tasks might benefit from it. Furthermore, as shown in Table 2 (**bottom**), rendering-based question answering surpasses the performance reported for text-based recognition. Following these findings, we equip the CAD-Assistant with a specialized recognition tools that generate an over-parameterized JSON representation of CAD models as well as renderings of 2D CAD sketch or 3D solid for comprehensive multimodal geometric reasoning.

4.2. CAD Benchmarks and Experimental Setup

As a generic framework, CAD-Assistant can be conditioned to perform a wide range of tasks related to CAD design. Given the lack of specialized evaluation benchmarks for CAD agents, this work adapts an evaluation set-

Method	Planner	2D Acc	3D Acc
SGPBench [45]	GPT-4 mini	0.594	0.737
	GPT-4 Turbo	0.674	0.762
	GPT-4o	0.686	0.782
CAD-ASSISTANT	GPT-4 mini	0.614	0.737
	GPT-4 Turbo	0.741	0.789
	GPT-4o	0.791	0.805

Table 3. Comparison for the proposed CAD-Assistant to baselines for CQA on the 2D and 3D subsets of SGPBench [45]. For CAD-Assistant performance is reported for different planners.

ting based on the following existing CAD tasks.

CAD Question Answering: As in subsection 4.1, quantitative evaluations of CAD Question Answering (CQA) is performed on the recently introduced SGPBench [45]. We do not provide the CAD code as part of the prompt as in [45]. Instead, the CAD sketch or model is pre-loaded into a FreeCAD project file, allowing CAD-Assistant to utilize the FreeCAD integration and CAD-specific tools to understand the design and answer questions. This experimental setup simulates a real-world question-answering environment where a CAD designer can ask open-ended questions about the design to support the iterative design process. We report accuracy on both the 2D and 3D CAD subsets. The 2D subset is sourced from SketchGraphs [48], while the 3D subset consists of 1000 CAD models from the DeepCAD dataset [59].

Autoconstraining: Parametric constraints are a key component of feature-based CAD modeling [37] and a widely adapted mechanism for explicit capturing of design intent [41, 72]. Given a CAD sketch of n parametric primitives $\{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n\} \in \mathcal{P}^n$ (lines, arcs, circles, points) the goal of autoconstraining is to infer a set of parametric constraints $\{\mathbf{c}_i\}_{i=1}^m \in \mathcal{C}^m$ applied on these primitives. Each constraint \mathbf{c}_i is composed of constraint type, participating primitives $\mathbf{p}_i, \mathbf{p}_j$ and subreferences (s_i, s_j) specifying the point of application (e.g. start, end, center). In contrast to the evaluation setting of [48, 49], we incorporate the application of the geometric solver (CAD software) to determine the final configuration of sketch primitives. Performance is measured in terms of Primitive F1 Score (*PF1*) and Constraint F1 Score (*CF1*) as in [66]. *PF1* defines a true positive as a primitive with the correct type and parameters within five quantization units, and for *CF1* a constraint is considered a true positive only if all associated primitives are also correctly predicted. Quantitative evaluations are performed on SketchGraphs [48]. We use the test set of [49] and evaluate on a subset of 700 CAD sketches due to the resource intensive nature GPT4-o API requests.

Hand-drawn CAD sketch Parameterization: Given a binary sketch image $\mathbf{X} \in \{0, 1\}^{h \times w}$, sketch parameteriza-

Method	Type	$PF1 \uparrow$	$CF1 \uparrow$
GPT-4o	<i>zero-shot</i>	0.693	0.274
Vitruvion [49]	<i>supervised</i>	0.706	0.238
CAD-ASSISTANT	<i>zero-shot</i>	0.979	0.484

Table 4. Evaluation on the task of autoconstraining. Performance is measured in terms of $PF1$ and $CF1$ on the SketchGraphs dataset [48].

CAD-Specific Tools		Prompting	$PF1 \uparrow$	$CF1 \uparrow$
<i>MMrecog</i>	<i>ConstrCheck</i>	Demonstr Docstr		
✗	✗	<i>0-shot</i>	✓	0.726 0.318
✓	✗	<i>0-shot</i>	✓	0.747 0.329
✓	✓	<i>0-shot</i>	✓	0.979 0.484
✓	✓	<i>5-shot</i>	✗	0.981 0.514
✓	✓	<i>5-shot</i>	✓	0.984 0.529

Table 5. Impact of CAD-specific tools and prompting strategies for CAD-Assistant on the autoconstraining task.

tion aims to recover the complete constrained CAD sketch ($\{\mathbf{p}_i\}_{i=1}^n, \{\mathbf{c}_i\}_{i=1}^m$). We report parametric accuracy computed on quantized primitive tokens as in [22, 49] after solving the CAD sketch. We also compute bidirectional Chamfer Distance (CD) on the image space. For evaluation, we use the same test split as in the autoconstraining task. For hand-drawn sketch synthesis, we follow the strategy of [49].

4.3. Results and Comparisons

We evaluate the performance of CAD-Assistant on the benchmarks described in the previous section.

CAD Question Answering: CAD-Assistant is able to interact directly with a CAD model via its integration with CAD software and is tasked with answering a question about the design. Results for CQA on SGPBench [45] are reported in Table 3. For this experiment, we also report the performance of the GPT-4 mini and GPT-4 Turbo models as planners. We observe that by leveraging available tools such as the Python interpreter and the comprehensive multimodal representation of CAD models generated via the recognizer tools, CAD-Assistant improves CQA performance for both CAD sketches and 3D CAD models, thus highlighting the potential of tool-use for CAD understanding. Notably, for the smaller GPT-4 mini, the performance gain from CAD-Assistant is marginally above (2D subset) or on-par (3D subset), emphasizing the need for pairing tool-augmented frameworks with a powerful VLM.

Autoconstraining: We evaluate our method on the task of CAD sketch autoconstraining [49]. CAD-Assistant is prompted to apply a set of parametric constraints with proper design intent to a CAD sketch preloaded into a FreeCAD project file, similar to the CQA setup. Performance is compared to a GPT-4o baseline and the constraint

Method	$Acc \uparrow$	$CD \downarrow$
Vitruvion [49]	0.659	1.586
Davinci [21]	0.789	1.184
CAD-ASSISTANT	0.784	0.680

Table 6. Evaluation on the task of hand-drawn image parameterization. Comparison against the task-specific models of [21, 49].

generation model Vitruvion [49], trained on a large-scale dataset [48]. Results are reported on Table 4. Note that the autoconstraining performance is reported after solving the predicted constraints with a CAD solver. As we are operating within CAD software, the CAD solver enforces the predicted constraints (e.g., orthogonality between two lines) on CAD sketches, adjusting the parameters of the affected primitives accordingly (e.g., modifying the parameters of the two lines). We observe that both the baseline and [49] tend to generate poorly parameterized constraints, which may lead to the arbitrary repositioning of primitives when applied by the CAD solver, as evidenced by the low $PF1$ values. In contrast, CAD-Assistant effectively utilizes tools to interact with the CAD software, assesses the impact of constraints, and preserve the integrity of the geometry. Notably, constraints generated by CAD-Assistant result in a high $CF1$ score despite *zero-shot* prompting, further underscores the broad understanding of CAD-Assistant in CAD design. In Table 5, we investigate the impact of tools relevant to auto-constraining on the effectiveness of CAD-Assistant. We find that both the multimodal sketch recognizer (*MMrecog*) and the constraint checker module (*ConstrCheck*) contribute to performance gains. Table 5 also compares prompting strategies for the proposed framework. While we primary focus on zero-shot prompting, which promotes agentic behavior by eliminating the need for CAD designers to create tailored examples for unique use cases, we find that a few high quality demonstrations can further enhance performance as shown by the results for *5-shot* prompting.

Hand-drawn CAD sketch Parameterization: Our framework utilizes the sketch parameterization tool that processes hand-drawn inputs to generate a textual description of primitives and constraints, as well as the constraint analysis module to assess the impact of constraints on CAD geometry. Performance is compared to task-specific models in Table 6. We observe that CAD-Assistant effectively translates the text-based parameterization recovered by the sketch parameterizer (based on [21]) into a FreeCAD sketch, resulting in high accuracy. Additionally, it successfully applies constraints without compromising the solved geometry, as evidenced by the reduction in CD.

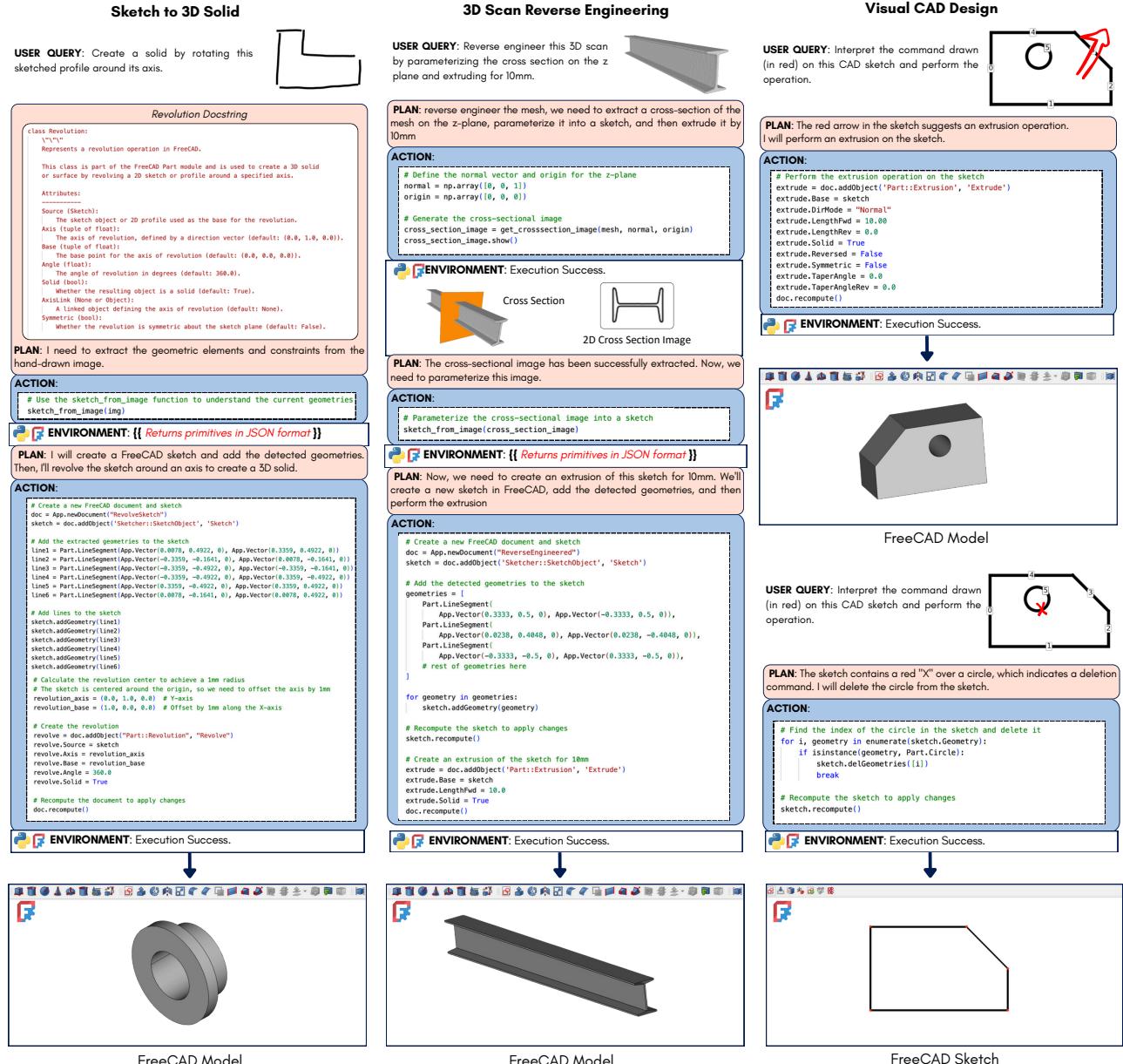


Figure 4. Real-world CAD use cases. (**Left**) The CAD-Assistant generated a 3D solid conditioned on a handdrawn sketch image. (**Center**) Our method reconstructs a 3D scan via cross-section parameterization. (**Right**) The CAD-Assistant semantically interprets the drawn operation and fulfills user requests directly without composing CAD-specific tools.

4.4. Exploring New Capabilities

This section identifies new emerging capabilities of tool-augmented CAD agents and demonstrates their potential beyond existing benchmarks on real-world use cases.

Beyond Simplified CAD Commands: Research on common CAD tasks generally focuses on the limited sets of CAD commands captured by large-scale datasets [48, 62]. As a train-free framework, CAD-Assistant can leverage the full range of commands available within the FreeCAD

API requiring only the corresponding docstring. On Figure 4 (left) and Figure 6 (supplementary) we showcase examples of our method utilizing the CAD commands *Revolution* and *Fillet* that are not included in existing datasets [62].

Real-world use cases: Tool augmentation allows interaction with multimodal inputs such as sketches and 3D scans. Figure 4 (center) showcases CAD-Assistant’s ability to process 3D scans along with textual queries to extract cross-sections, parameterize features, and reverse engineer CAD

models from scans. In Figure 4 (*right*), the VLLM planner determines to semantically interpret drawn operation directly without utilizing additional CAD-specific tools for fulfilling user requests. Note that generated FreeCAD code is interpretable, editable and easily extendable.

5. Conclusion

In this work, we introduce CAD-Assistant, a general-purpose CAD agent built on a VLLM planner and a tool-use paradigm of CAD-specific tools. Our framework responds to multimodal queries via generated actions that are executed in a python interpreter integrated with FreeCAD. We assess CAD-Assistant on diverse CAD benchmarks and qualitatively demonstrate the potential of tool-augmented VLLMs for the automation in real-world CAD workflows.

6. Acknowledgements

The present work is supported by the National Research Fund (FNR), Luxembourg, under the BRIDGES2021/IS/16849599/FREE-3D project and by Artec3D.

References

- [1] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. Flamingo: a visual language model for few-shot learning. *NeurIPS*, 2022. [2](#)
- [2] Kamel Alrashedy, Pradyumna Tambwekar, Zulfiqar Haider Zaidi, Megan Langwasser, Wei Xu, and Matthew Gombolay. Generating cad code with vision-language models for 3d designs. In *The Thirteenth International Conference on Learning Representations*. [3](#)
- [3] Anthropic. Introducing the next generation of claude. 2023. [2](#)
- [4] Akshay Badagabettu, Sai Sravan Yarlagadda, and Amir Barati Farimani. Query2cad: Generating cad models using natural language queries. *ArXiv*, 2024. [3](#)
- [5] Antoine Brière-Côté, Louis Rivest, and Roland Maranzana. Comparing 3d cad models: uses, methods, tools and perspectives. *Computer-Aided Design and Applications*, 2012. [1](#)
- [6] Francesco Buonomamici, Monica Carfagni, Rocco Furferi, Lapo Governi, Alessandro Lapini, and Yary Volpe. Reverse engineering modeling methods and tools: a survey. *Computer-Aided Design and Applications*, 2018. [1](#)
- [7] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *ArXiv*, 2023. [3](#)
- [8] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé J’egou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging properties in self-supervised vision transformers. *ICCV*, 2021. [3](#)
- [9] Kseniya Cherenkova, Elona Dupont, Anis Kacem, Ilya Arzhannikov, Gleb Gusev, and Djamil Aouada. Sepicnet: Sharp edges recovery by parametric inference of curves in 3d shapes. In *CVPRW*, 2023. [1](#)
- [10] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *ArXiv*, 2021. [3](#)
- [11] FreeCAD Community. Freecad, 2024. [2](#), [4](#), [5](#), [1](#)
- [12] Wenliang Dai, Junnan Li, Dongxu Li, Anthony Meng Huat Tiong, Junqi Zhao, Weisheng Wang, Boyang Albert Li, Pascale Fung, and Steven C. H. Hoi. Instructclip: Towards general-purpose vision-language models with instruction tuning. *ArXiv*, 2023. [2](#)
- [13] Elona Dupont, Kseniya Cherenkova, Dimitrios Mallis, Gleb Gusev, Anis Kacem, and Djamil Aouada. Transcad: A hierarchical transformer for cad sequence inference from point clouds. In *ECCV*, 2024. [1](#)
- [14] Xi Fang, Weijie Xu, Fiona Anting Tan, Jian Zhang, Ziqing Hu, Yanjun Qi, Scott Nickleach, Diego Socolinsky, Srinivasan H. Sengamedu, and Christos Faloutsos. Large language models(llms) on tabular data: Prediction, generation, and understanding - a survey. *ArXiv*, 2024. [5](#)
- [15] FreeCAD Community. The oca file format. [6](#)
- [16] Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. *CVPR*, 2022. [3](#), [5](#)
- [17] Eunji Hong, Minh Hieu Nguyen, Mikaela Angelina Uy, and Minhyuk Sung. Mv2cyl: Reconstructing 3d extrusion cylinders from multi-view images. *NeurIPS*, 2024. [1](#)
- [18] Cheng-Yu Hsieh, Sibe Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander J. Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. Tool documentation enables zero-shot tool usage with large language models. *ArXiv*, 2023. [5](#)
- [19] Yushi Hu, Weijia Shi, Xingyu Fu, Dan Roth, Mari Ostendorf, Luke S. Zettlemoyer, Noah A. Smith, and Ranjay Krishna. Visual sketchpad: Sketching as a visual chain of thought for multimodal language models. *ArXiv*, 2024. [2](#), [3](#), [5](#)
- [20] Autodesk Inc. Dxf reference, 2012. [6](#)
- [21] Ahmet Serdar Karadeniz, Dimitrios Mallis, Nesryne Mejri, Kseniya Cherenkova, Anis Kacem, and Djamil Aouada. Davinci: A single-stage architecture for constrained cad sketch inference. In *BMVC*, 2024. [1](#), [5](#), [7](#), [2](#)
- [22] Ahmet Serdar Karadeniz, Dimitrios Mallis, Nesryne Mejri, Kseniya Cherenkova, Anis Kacem, and Djamil Aouada. Picasso: A feed-forward framework for parametric inference of cad sketches via rendering self-supervision. In *WACV*, 2025. [1](#), [5](#), [6](#), [7](#), [2](#)
- [23] Mohammad Sadil Khan, Elona Dupont, Sk Aziz Ali, Kseniya Cherenkova, Anis Kacem, and Djamil Aouada. Cad-signet: Cad language inference from point clouds using layer-wise sketch instance guided attention. In *CVPR*, 2024. [1](#)
- [24] Claudius Kienle, Benjamin Alt, Darko Katic, and Rainer Jäkel. Quercad: Grounded question answering for cad models. *ArXiv*, 2024. [3](#)
- [25] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer White-

- head, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross B. Girshick. Segment anything. *ICCV*, 2023. 3
- [26] Milin Kodnongbua, Benjamin Jones, Maaz Bin Safeer Ahmad, Vladimir Kim, and Adriana Schulz. Reparamcad: Zero-shot cad re-parameterization for interactive manipulation. In *SIGGRAPH Asia*, 2023. 1, 3
- [27] Mojtaba Komeili, Kurt Shuster, and Jason Weston. Internet-augmented dialogue generation. In *Annual Meeting of the Association for Computational Linguistics*, 2021. 3
- [28] Junnan Li, Dongxu Li, Caiming Xiong, and Steven C. H. Hoi. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *ICML*, 2022. 2
- [29] Xueyang Li, Yu Song, Yunzhong Lou, and Xiangdong Zhou. CAD translator: An effective drive for text to 3d parametric computer-aided design generative modeling. In *ACM Multimedia 2024*, 2024. 2
- [30] Xingang Li, Yuewan Sun, and Zhenghui Sha. Llm4cad: Multi-modal large language models for 3d computer-aided design generation. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2024. 3
- [31] Haotian Liu, Chunyuan Li, Qingshang Wu, and Yong Jae Lee. Visual instruction tuning. In *NeurIPS*, 2023. 2
- [32] Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. Improved baselines with visual instruction tuning. In *CVPR*, 2024. 2
- [33] Pan Lu, Swaroop Mishra, Tony Xia, Liang Qiu, Kai-Wei Chang, Song-Chun Zhu, Oyvind Tafjord, Peter Clark, and Ashwin Kalyan. Learn to explain: Multimodal reasoning via thought chains for science question answering. In *NeurIPS*, 2022. 2, 3
- [34] Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. In *NeurIPS*, 2023. 2, 3, 5
- [35] Pan Lu, Liang Qiu, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, Tanmay Rajpurohit, Peter Clark, and A. Kalyan. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. *ICLR*, 2023. 2, 3
- [36] Liane Makatura, Michael Foshey, Bohan Wang, Felix Hahnelein, Pingchuan Ma, Bolei Deng, Megan Tjandrasuwita, Andrew Everett Spielberg, Crystal Elaine Owens, Peter Chen, Allan Zhao, Amy Zhu, Wil J. Norton, Edward Gu, Joshua Jacob, Yifei Li, Adriana Schulz, and Wojciech Matusik. How can large language models help humans in design and manufacturing? *ArXiv*, 2023. 2
- [37] Dimitrios Mallis, Ali Sk Aziz, Elona Dupont, Kseniya Cherenkova, Ahmet Serdar Karadeniz, Mohammad Sadil Khan, Anis Kacem, Gleb Gusev, and Djamil Aouada. Sharp challenge 2023: Solving cad history and parameters recovery from point clouds and 3d scans. overview, datasets, metrics, and baselines. In *CVPRW*, 2023. 1, 6, 2
- [38] Meta. The llama 3 herd of models. *ArXiv*, 2024. 2
- [39] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Ouyang Long, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Buttton, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback. *ArXiv*, 2021. 3
- [40] OpenAI. Gpt-4 technical report. 2023. 2, 3
- [41] Jeffrey M. Otey, Manuel Contero, and Jorge D. Camba. Revisiting the design intent concept in the context of mechanical cad education. *Computer-aided Design and Applications*, 2018. 6
- [42] Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *ArXiv*, 2022. 3
- [43] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *ArXiv*, 2023. 3
- [44] Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shi Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bo Li, Ziwei Tang, Jing Yi, Yu Zhu, Zhenning Dai, Lan Yan, Xin Cong, Ya-Ting Lu, Weilin Zhao, Yuxiang Huang, Jun-Han Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. Tool learning with foundation models. *ArXiv*, 2023. 3
- [45] Zeju Qiu, Weiyang Liu, Haiwen Feng, Zhen Liu, Tim Z. Xiao, Katherine M. Collins, Joshua B. Tenenbaum, Adrian Weller, Michael J. Black, and Bernhard Schölkopf. Can large language models understand symbolic graphics programs? *ArXiv*, 2024. 2, 3, 5, 6, 7, 1
- [46] Colin Raffel, Noam M. Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 2019. 3
- [47] Danila Rukhovich, Elona Dupont, Dimitrios Mallis, Kseniya Cherenkova, Anis Kacem, and Djamil Aouada. Cad-recode: Reverse engineering cad code from point clouds. *ArXiv*, 2024. 1
- [48] Ari Seff, Yaniv Ovadia, Wenda Zhou, and Ryan P. Adams. SketchGraphs: A large-scale dataset for modeling relational geometry in computer-aided design. In *ICMLW*, 2020. 2, 3, 5, 6, 7, 8
- [49] Ari Seff, Wenda Zhou, Nick Richardson, and Ryan P Adams. Vitruvion: A generative model of parametric cad sketches. In *ICLR*, 2022. 1, 2, 6, 7
- [50] Pratyusha Sharma, Tamar Rott Shaham, Manel Baradad, Stephanie Fu, Adrian Rodriguez-Munoz, Shivam Duggal, Phillip Isola, and Antonio Torralba. A vision check-up for language models. *CVPR*, 2024. 2, 3, 5
- [51] Yongliang Shen, Kaitao Song, Xu Tan, Dong Sheng Li, Weiming Lu, and Yue Ting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *ArXiv*, 2023. 2, 3
- [52] Robert W. Shumaker, Kristina R. Walkup, and Benjamin B. Beck. Animal tool behavior: The use and manufacture of tools by animals. 2011. 3
- [53] Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. Retrieval augmentation reduces hallucination

- in conversation. In *Conference on Empirical Methods in Natural Language Processing*, 2021. 3
- [54] Yuan Sui, Mengyu Zhou, Mingjie Zhou, Shi Han, and Dongmei Zhang. Table meets llm: Can large language models understand structured table data? a benchmark and empirical study. *International Conference on Web Search and Data Mining*, 2023. 5
- [55] D’idac Sur’sis, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. *ICCV*, 2023. 2, 3, 5
- [56] Mikaela Angelina Uy, Yen-Yu Chang, Minhyuk Sung, Purvi Goel, Joseph G Lambourne, Tolga Birdal, and Leonidas J Guibas. Point2cyl: Reverse engineering 3d objects from point clouds to extrusion cylinders. In *CVPR*, 2022. 1
- [57] Chenfei Wu, Sheng-Kai Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. Visual chatgpt: Talking, drawing and editing with visual foundation models. *ArXiv*, 2023. 2, 3
- [58] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *ArXiv*, abs/2308.08155, 2023. 1
- [59] Rundi Wu, Chang Xiao, and Changxi Zheng. Deepcad: A deep generative network for computer-aided design models. In *CVPR*, 2021. 1, 2, 6
- [60] Sifan Wu, Amir Hosein Khasahmadi, Mor Katz, Pradeep Kumar Jayaraman, Yewen Pu, Karl D.D. Willis, and Bang Liu. Cadlilm: Bridging language and vision in the generation of parametric cad sketches. *ECCV*, 2024. 3
- [61] Sifan Wu, Amir Hosein Khasahmadi, Mor Katz, Pradeep Kumar Jayaraman, Yewen Pu, Karl D.D. Willis, and Bang Liu. Cad-llm: Large language model for cad generation. 2024. 3
- [62] Peng Xu, Timothy M Hospedales, Qiyue Yin, Yi-Zhe Song, Tao Xiang, and Liang Wang. Deep learning for free-hand sketch: A survey. *IEEE TPAMI*, 2022. 1, 8, 2
- [63] Xiang Xu, Karl DD Willis, Joseph G Lambourne, Chin-Yi Cheng, Pradeep Kumar Jayaraman, and Yasutaka Furukawa. Skexgen: Autoregressive generation of cad construction sequences with disentangled codebooks. In *ICML*, pages 24698–24724. PMLR, 2022. 2
- [64] Xiang Xu, Pradeep Kumar Jayaraman, Joseph G Lambourne, Karl DD Willis, and Yasutaka Furukawa. Hierarchical neural coding for controllable cad model generation. *ICML*, 2023. 2
- [65] Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v. In *arXiv*, 2023. 1
- [66] Yuezhi Yang and Hao Pan. Discovering design concepts for cad sketches. *arXiv*, 2022. 6
- [67] Zhengyuan Yang, Linjie Li, Jianfeng Wang, Kevin Lin, Ehsan Azarnasab, Faisal Ahmed, Zicheng Liu, Ce Liu, Michael Zeng, and Lijuan Wang. Mm-react: Prompting chatgpt for multimodal reasoning and action. *ArXiv*, 2023. 3
- [68] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Joshua B. Tenenbaum. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. In *NIPS*, 2018. 3
- [69] Yang You, Mikaela Angelina Uy, Jiaqi Han, Rahul Thomas, Haotong Zhang, Suya You, and Leonidas Guibas. Img2cad: Reverse engineering 3d cad models from images through vlm-assisted conditional factorization. *ArXiv*, 2024. 1, 3
- [70] Haocheng Yuan, Jing Xu, Hao Pan, Adrien Bousseau, Niloy J. Mitra, and Changjian Li. Cadtalk: An algorithm and benchmark for semantic commenting of cad programs. *CVPR*, 2024. 3
- [71] Andy Zeng, Adrian S. Wong, Stefan Welker, Krzysztof Choromanski, Federico Tombari, Aveek Purohit, Michael S. Ryoo, Vikas Sindhwani, Johnny Lee, Vincent Vanhoucke, and Peter R. Florence. Socratic models: Composing zero-shot multimodal reasoning with language. *ArXiv*, 2022. 3
- [72] Yingzhong Zhang and Xiaofang Luo. Design intent information exchange of feature-based cad models. *2009 WRI World Congress on Computer Science and Information Engineering*, 2009. 6
- [73] Xiangyu Zhu, Dong Du, Weikai Chen, Zhiyou Zhao, Yinyu Nie, and Xiaoguang Han. Nerve: Neural volumetric edges for parametric curve extraction from point cloud. In *CVPR*, 2023. 1



CAD-Assistant: Tool-Augmented VLLMs as Generic CAD Task Solvers

Supplementary Material

This supplementary material includes various details that were not reported in the main paper due to space constraints. To demonstrate the benefit of the proposed CAD-Assistant, we also expand our qualitative evaluation.

7. CAD-specific Tool-set

This section provides a detailed discussion of the CAD-specific tool set utilised by the proposed framework. CAD-ASSISTANT is equipped with the following tools:

Hand-drawn Image Parameterizer: To enable visual sketching, we employ a task-specific model for hand-drawn image parameterization [21]. This module extracts parameters and constraints as text, allowing CAD-Assistant to reuse primitive parameters for CAD code generation.

CAD Sketch Recognizer: We equip CAD-Assistant with a CAD sketch recognition utility. This routine returns both a summary of geometries and parametric constraints in .json format, along with a visual rendering of the CAD sketch. The rendered sketch image includes numeric markers of the primitive ID overlaid on the rendered geometries. Motivated by [65], this approach enhances visual grounding for GPT-4o, *i.e.* its ability to associate visual content with the textual description of primitives.

3D Solid Recognizer: For CAD model recognition, we also incorporate a 3D solid recognizer that generates a .json summary of model parameters (for both sketch and extrusion operations) along with visual renderings of the 3D solid from four different angles, providing a multimodal representation of structure and geometry.

Constraint Checker: We include a dedicated function that evaluates the parameters of a parametric constraint to determine its validity and whether it causes movement in geometric elements. The constraint analyzer facilitates effective interaction with the CAD solver by assessing the impact of commands like parametric constraints on geometry.

Cross-section Extract: Cross-sections are critical components of CAD reverse engineering workflows [6]. CAD-Assistant includes a specialized routine for 2D cross-section images from 3D scans across 2D planes.

FreeCAD API: CAD-Assistant is integrated with the open-source FreeCAD software [11] via the FreeCAD Python API. This API enables programmatic control over the majority of commands available to designers and access to the current state of the CAD design. In this work, we consider a range of components from the Sketcher

and Part modules of the FreeCAD API, focusing on CAD sketching, the addition and manipulation of primitives, geometric constraints, and extrusion operations for constructing 3D solids. A summary of the exact classes, methods and class attributes of the FreeCAD API integrated with CAD-Assistant is provided in the supplementary.

Python: Beyond facilitating actions a_t , the planner can utilize Python as a tool to conduct essential logical and mathematical operations, such as calculating segment lengths, determining angles, and deriving parameter values.

8. System Details

CAD-Assistant's implementation is based on the Autogen [58] programming framework for Agentic AI. We report CAD-Assistant's performance with gpt-4o-mini-2024-07-18, gpt-4-turbo-2024-04-09 and gpt-4o-2024-08-06 as VLLM planners, accessed via API calls.

9. CAD Representations

In this section, we provide a formally introduction of 2D CAD sketches and 3D CAD models.

9.1. Constrained CAD Sketches

A constraint CAD sketch is commonly represented by a graph $\mathcal{G} = (\mathcal{P}^n, \mathcal{C}^m)$ comprising a set of n primitive nodes $\{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n\} \in \mathcal{P}^n$ and m edges between nodes $\{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_m\} \in \mathcal{C}^m$ denoting geometric constraints. Primitives \mathbf{p}_i are of type line \mathbf{l}_i , arc \mathbf{a}_i , circle \mathbf{c}_i or points \mathbf{d}_i . VLLM and LLM planners can be sensitive to the parameterization strategy followed for representing \mathbf{p}_i . This work conducts an investigation on the impact of sketch parameterization on visual program understanding in black-box VLLMs presented in section 4.1 where we compare the following parameterization strategies:

Implicit: This is the parameterization strategy utilized for representation of 2D CAD sketches by the SGPBench [45]. Primitives p_i are represented as follows:

$$\begin{aligned}\mathbf{a}_i &= (x_c, y_c, v_x, v_y, b_{wc}, \theta_s, \theta_e) \in \mathbb{R}^4 \times \{0, 1\} \times [0, 2\pi]^2 \\ \mathbf{c}_i &= (x_c, y_c, r) \in \mathbb{R}^3 \\ \mathbf{l}_i &= (x_p, y_p, v_x, v_y, d_s, d_e) \in \mathbb{R}^6 \\ \mathbf{d}_i &= (x_p, y_p) \in \mathbb{R}^2\end{aligned}$$

Table 7. Implicit parameterization strategy for arcs \mathbf{a}_i , circles \mathbf{c}_i , lines \mathbf{l}_i and points \mathbf{p}_i .

where and (x_c, y_c) denotes center point coordinates, (d_s, d_e) are signed start/end point distances to a point (x_p, y_p) , the unit direction vector is denoted as (v_x, v_y) , radius is denoted with r ,

(θ_s, θ_e) are the start/end angles to the unit direction vector in radians and b_{wc} is a binary flag indicating if the arc is clockwise.

Point-based: We contrast the implicit parameterization to the point-based approach from [21, 22, 49] as described on the following table.

$\mathbf{a}_i = (x_s, y_s, x_m, y_m, x_e, y_e) \in \mathbb{R}^6$
$\mathbf{c}_i = (x_c, y_c, r) \in \mathbb{R}^3$
$\mathbf{l}_i = (x_s, y_s, x_e, y_e) \in \mathbb{R}^4$
$\mathbf{d}_i = (x_p, y_p) \in \mathbb{R}^2$

Table 8. Point-based parameterization strategy for arcs \mathbf{a}_i , circles \mathbf{c}_i , lines \mathbf{l}_i and points \mathbf{p}_i .

where $(x_s, y_s), (x_m, y_m), (x_e, y_e)$ are start, middle and end point coordinates and r is the radius.

Overparameterized: This strategy is a simple combination of the implicit and point-based parameterization.

$\mathbf{a}_i = (x_c, y_c, v_x, v_y, x_s, y_s, x_m, y_m, x_e, y_e, b_{wc}, \theta_s, \theta_e) \in \mathbb{R}^{10} \times \{0, 1\} \times [0, 2\pi)^2$
$\mathbf{c}_i = (x_c, y_c, r) \in \mathbb{R}^3$
$\mathbf{l}_i = (x_p, y_p, v_x, v_y, d_s, d_e, x_s, y_s, x_e, y_e) \in \mathbb{R}^{10}$
$\mathbf{d}_i = (x_p, y_p) \in \mathbb{R}^2$

Table 9. Overparameterized parameterization strategy for arcs \mathbf{a}_i , circles \mathbf{c}_i , lines \mathbf{l}_i and points \mathbf{p}_i .

We identify the overparameterized strategy as the safest approach, as it enables the VLLM planner to leverage a broader and more diverse set of parameters, better accommodating the varying requirements of different input queries. In addition to parametric primitives \mathbf{p}_i , a CAD sketch incorporates constraints defined by CAD designers, ensuring that future modifications propagate coherently throughout the design. A constraint is defined as an undirected between primitives \mathbf{p}_i and \mathbf{p}_j . They might also include subreferences $(s_i, s_j) \in \llbracket 1..4 \rrbracket^2$, to specify whether the constraint is applied on *start*, *end*, *middle* point, or *entire* primitive for both \mathbf{p}_i and \mathbf{p}_j . Note that some constraints may involve only a single primitive \mathbf{p}_i (e.g. a vertical line); in such cases, the constraint is defined as the edge between the primitive and itself. In this work we consider the following types of constraints: *coincident*, *parallel*, *equal*, *vertical*, *horizontal*, *perpendicular*, *tangent*.

9.2. CAD Models

Following the feature-based CAD modeling paradigm [37, 62], a CAD model $\mathbf{C} \in \mathcal{C}$ is constructed as a sequence of design steps. In this work, evaluation is performed on CAD models from the 3D partition of SGPBench [45] sourced from the Deep-CAD dataset [62]. These models are constructed exclusively via a *sketch-extrude* strategy, where 2D CAD sketches \mathcal{G}_i are followed by extrusion operations that turns the sketch into a 3D volume. Extrusions include the following parameters:

Parameter Description	Parameter Notation
Sketch plane orientation	$(\theta, \phi, \gamma) \in [0, \pi] \times [0, 2\pi]^2$
Sketch plane translation	$(\tau_x, \tau_y, \tau_z) \in \mathbb{R}^3$
sketch scale	$\sigma \in \mathbb{R}$
Extrusion distance	$(d_-, d_+) \in \mathbb{R}^2$
Extrusion type	$\beta \in \{1, 2, 3, 4\}$

Table 10. Extrusion Parameters description.

where extrusion type β can be among *new*, *cut*, *join* and *intersect*.

9.3. Parameter Quantization

Unlike prior task-specific models for CAD-related tasks such as hand-drawn sketch parameterization [21, 22, 49], CAD sketch generation [49], or 3D CAD model generation [62], the CAD-Assistant does not rely on the common practice of parameter quantization. Typically, these methods use a 6-bit uniform quantization scheme to convert continuous sketch and extrusion parameters into discrete tokens, enabling prediction through transformer-based sequence architectures trained with cross-entropy loss [21, 22, 49, 62]. In contrast, the CAD-Assistant employs a VLLM planner that directly regresses primitive and extrusion parameters as continuous numerical values. We apply the 6-bit uniform quantization to the outputs of CAD-Assistant, to facilitate direct comparisons with task-specific methods for auto-constraining and hand-drawn sketch parameterization reported on section 4.2 of the main paper.

QUERY: How many vertices does the CAD object have? **A) 4 B) 5 C) 6 D) 7**

PLAN 1: I will count the distinct start and end points of these line segments.

ACTION 1

```
# Extracting unique vertices from the sketch geometry
vertices = set()
for geometry in geometries:
    vertices.add(geometry["StartPoint"]["x"], geometry["StartPoint"]["y"])
    vertices.add(geometry["EndPoint"]["x"], geometry["EndPoint"]["y"])
print(f"Number of unique vertices: {len(vertices)}")
```

 ENVIRONMENT: Execution Success. Number of unique vertices: 11

PLAN 2: The calculation of unique vertices seems **incorrect** as 11 is not one of the possible answers. I need to carefully examine the coordinates and ensure that similar points are not counted multiple times due to minor differences.

ACTION 2

```
vertices = [
    (-0.232, -0.1977), (-0.1855, -0.1977), (-0.1866, -0.1973), (-0.0122, -0.4994), (-0.011, -0.5),
    (-0.2331, -0.5), 0.2331, 0.5), (-0.0459, 0.5), (-0.0459, 0.1233), (-0.0471, 0.125), (-0.2331, -0.1972)
]

# Using a consistent precision for rounding
unique_vertices = set()
for vertex in vertices:
    unique_vertices.add((round(vertex[0], 2), round(vertex[1], 2)))

print(f"Re-evaluated number of unique vertices: {len(unique_vertices)})")
```

 ENVIRONMENT: Execution Success. Number of unique vertices: 7

Figure 5. Correction of an inaccurate answer for a CQA example.

10. Verification of Responses

The proposed workflow allows for verifying incorrect responses. The generated plan can be updated based on intermediate code execution results, including error logs (see f_t in Eq. 4). Figure 5 shows that when an error occurs (*i.e.*, answer 11 is not among the possible choices of the question), the planner detects this mistake and updates the plan accordingly. Exploring the potential of

combining tool-augmentation with more advanced planning and verification algorithms (*e.g.* [2]) is left as interesting future work.

11. Costs

The proposed CAD-Assistant utilizes a GPT-4o planner accessed through API calls. Table 11 provides a summary of the costs associated with each user query across CAD benchmarks.

12. CAD-Assistant Prompts

In this work, we use a unified prompt template, similar to [19] for all CAD-specific problems. The prompt consists of three key components: (1) a general context, (2) a list of tools provided to the VLLM planner via docstrings, and (3) a multimodal user request. A summary of the FreeCAD API commands is provided in Table 12, and the full set of docstrings supplied to the planner is presented in Section 15. Note that as the set of considered API commands increases, the input context of the VLLM planner could increase. To address this, a preprocessing step could be implemented to dynamically select relevant docstrings before execution. The general context available to the VLLM planner is shown in Figure 7.

13. Beyond Simplified CAD Commands

Extending the discussion of Sec. 4.4, we provide an additional qualitative example of the proposed CAD-Assistant. Figure 6 shows the utilization of the CAD operation *Fillet* by our method. It can be observed that CAD-Assistant computes the intersection of the lines to be able to perform the *Fillet* operation on the corners by analyzing its docstring. Moreover, we find that VLLM planner performance might vary across CAD commands. This highlights the necessity of developing CAD-specific benchmarks tailored to CAD agents. Such benchmarks are crucial for gaining deeper insights into the capabilities and limitations of VLLM planners on generic CAD task solving.

14. Qualitative Evaluation

This supplementary material presents examples of complete agent trajectories for the CAD benchmarks used in this study. Detailed examples from the 2D and 3D subsets of SGPBench are provided in subsections 14.1 and 14.2. Trajectories for the autoconstraining task are illustrated in subsection 14.3, while examples of hand-drawn parameterization are presented in subsection 14.4.

USER QUERY: Smooth out the 4 corners of this CAD sketch

Fillet Docstring

```
def fillet(self,
          index1: int,
          index2: int,
          start_point1: App.Vector,
          start_point2: App.Vector,
          radius: float,
          clockwise: bool = True,
          merge: bool = False
        ) -> None:
    """
    Creates a fillet between two lines in a sketch.

    Note:
    -----
    This function strictly requires positional arguments and does not support keyword arguments.

    Parameters:
    -----
    index1 (int):
        The index of the first line in the sketch.Geometry list.
    index2 (int):
        The index of the second line in the sketch.Geometry list.
    start_point1 (App.Vector):
        The point of the first line where the fillet starts.
    start_point2 (App.Vector):
        The point of the second line where the fillet starts.
    radius (float):
        The radius of the fillet.
    clockwise (bool):
        Whether the fillet is drawn clockwise. Always use True.
    merge (bool):
        Whether to merge the fillet with the original geometry. Always use False.

    Returns:
    -----
    None
    """

```

PLAN: To smooth the 4 corners of the rectangle, I need to first identify the indices of the lines forming the rectangle in the sketch. Then, I can use the 'fillet' method at each corner.

ACTION:

```
[ sketch_recognizer(sketch) ]
```

ENVIRONMENT: {{ Returns primitives in JSON format }}

PLAN: The rectangle is formed by four line segments. I will apply a fillet to each corner. I need to calculate the start points for the fillet on each pair of lines at the corners.

ACTION:

```
# Define the radius for the fillet
radius = 2.0

# Calculate start points for the fillet at each corner
# Corner 1: Between line 0 and line 1
intersection_point_1 = sketch.Geometry[0].EndPoint
line0_direction = (sketch.Geometry[0].EndPoint - intersection_point_1).normalize()
line1_direction = (sketch.Geometry[1].EndPoint - intersection_point_1).normalize()
start_point_1_0 = intersection_point_1 + line0_direction * radius
start_point_1_1 = intersection_point_1 + line1_direction * radius

# Corner 2: Between line 1 and line 2
intersection_point_2 = sketch.Geometry[1].EndPoint
line1_direction = (sketch.Geometry[1].EndPoint - intersection_point_2).normalize()
line2_direction = (sketch.Geometry[2].EndPoint - intersection_point_2).normalize()
start_point_2_1 = intersection_point_2 + line1_direction * radius
start_point_2_2 = intersection_point_2 + line2_direction * radius

# Corner 3: Between line 2 and line 3
intersection_point_3 = sketch.Geometry[2].EndPoint
line2_direction = (sketch.Geometry[2].EndPoint - intersection_point_3).normalize()
line3_direction = (sketch.Geometry[3].EndPoint - intersection_point_3).normalize()
start_point_3_2 = intersection_point_3 + line2_direction * radius
start_point_3_3 = intersection_point_3 + line3_direction * radius

# Corner 4: Between line 3 and line 0
intersection_point_4 = sketch.Geometry[3].EndPoint
line3_direction = (sketch.Geometry[3].EndPoint - intersection_point_4).normalize()
line0_direction = (sketch.Geometry[0].EndPoint - intersection_point_4).normalize()
start_point_4_3 = intersection_point_4 + line3_direction * radius
start_point_4_0 = intersection_point_4 + line0_direction * radius

# Apply fillet at each corner
sketch.fillet(0, start_point1_0, start_point1_1, radius, True, False)
sketch.fillet(1, start_point2_1, start_point2_2, radius, True, False)
sketch.fillet(2, start_point3_2, start_point3_3, radius, True, False)
sketch.fillet(3, start_point4_3, start_point4_0, radius, True, False)

# Recompute the document to apply changes
doc.recompute()
```

ENVIRONMENT: Execution Success.

FreeCAD Sketch

Figure 6. Example of the proposed CAD-Assistant utilizing the *Fillet* CAD command.

Task	Avg Input Tokens	Avg Output Tokens	Avg Cost per User Request
CAD Question Answering	11280	178	\$0.0299
Autoconstraining	28422	852	\$0.0795
Handdrawn sketch parameterization	31170	1081	\$0.0887

Table 11. Cost per user request for the CAD-Assistant utilizing GPT-4o as VLLM planner.

FreeCAD Class	Class Methods	Class Attributes
Sketcher.Sketch	<code>__init__()</code> , <code>recompute()</code> , <code>delGeometries(indx)</code> , <code>addConstraint(const)</code> , <code>addGeometry(geometry)</code>	<code>Name</code> , <code>Geometry</code> , <code>Constraints</code> , <code>State</code> , <code>ConstraintCount</code> , <code>GeometryCount</code> , <code>Placement</code>
Sketcher.Constraints	<code>__init__(constraintType, *args)</code>	<code>Name</code>
Part.Circle	<code>__init__(center, normal, radius)</code>	<code>Center</code> , <code>Radius</code>
Part.Point	<code>__init__(point)</code>	<code>X</code> , <code>Y</code> , <code>Z</code>
Part.ArcOfCircle	<code>__init__(circle, startParam, endParam)</code> , <code>__init__(startPoint, endPoint, midPoint)</code>	<code>Center</code> , <code>Radius</code> , <code>StartPoint</code> , <code>EndPoint</code> , <code>FirstParameter</code> , <code>LastParameter</code>
Part.LineSegment	<code>__init__(startPoint, endPoint)</code>	<code>StartPoint</code> , <code>EndPoint</code>
Part.Extrude	<code>__init__()</code>	<code>Base</code> , <code>DirMode</code> , <code>LengthFwd</code> , <code>LengthRev</code> , <code>Solid</code> , <code>Reversed</code> , <code>Symmetric</code> , <code>TaperAngle</code> , <code>TaperAngleRev</code>
Part.Solid	<code>fuse(shape)</code> , <code>cut(shape)</code> , <code>common(shape)</code>	<code>TypeId</code> , <code>Volume</code> , <code>BoundBox</code>

Table 12. Summary of FreeCAD API classes, methods, and attributes utilized by the CAD-Assistant framework. The VLLM planner is supplied with docstrings that clarify their use, including detailed descriptions, function signatures and usage examples.

Prompt Template:

You are a helpful multimodal Computer Aided Design (CAD) AI assistant.

Solve tasks using your vision, coding, and language skills.

The task can be free-form or multiple-choice questions.

You can answer the user's question. If you are not sure, you can code.

You are coding in a Python jupyter notebook environment. The environment has also access to the PYTHON FREECAD API.

You can suggest python code (in a python coding block) for the user to execute. In a dialogue, all your codes are executed with the same jupyter kernel, so you can use the variables, working states in your earlier code blocks.

Solve the task step by step if you need to.

The task may require several steps. Give your code to the user to execute. The user may reply with the text and image outputs of the code execution. You can use the outputs to proceed to the next step, with reasoning, planning, or further coding.

When using code, you must indicate the script type in the code block. The user cannot provide any other feedback or perform any other action beyond executing the code you suggest. The user can't modify your code. So do not suggest incomplete code which requires users to modify. Don't use a code block if it's not intended to be executed by the user.

Don't include multiple code blocks in one response. Do not ask users to copy and paste the result. Instead, use 'print' function for the output when relevant. Check the execution result returned by the user.

For each turn, you should first do a "PLAN", based on the images and text you see.

Here are the commands that you can use. Call them as described by the following docstrings

<DOCSTRINGS>

The jupyter notebook has already executed the following code to import the necessary packages:

```
import sys
from PIL import Image
from tools import *
import math
import copy
import numpy as np
import FreeCAD as App
import Part, Sketcher
from enum import Enum

img = Image.open('handdrawn_image.png')
```

REQUIREMENTS #:

1. The generated actions can resolve the given user request # USER REQUEST # perfectly. The user request is reasonable and can be solved. Try your best to solve the request.
2. The arguments of a tool must be the same number, modality, and format specified in # DOCSTRINGS #;
3. If you think you got the answer, use ANSWER: <your answer> to provide the answer, and ends with TERMINATE.
4. All images should be stored in PIL Image objects. The user will send generated image outputs to you.
5. Only the FreeCAD commands that are provided to you. Do not used different commands or import new modules, as they will not be accessible.
6. Use as few tools as possible. Only use the tools for the use cases written in the tool description. You can use multiple tools in a single action.
7. You must return an answer with the choice letter if the user request is a multiple-choice question.

USER REQUEST # <REQUEST HERE>

Now please generate only PLAN and ACTION in RESULT. If no action needed, also reply with you answer and ends with TERMINATE in the RESULT:#

Figure 7. Prompt template for the CAD-Assistant. A detailed docstring disambiguating the use of the FreeCAD API and CAD-specific tools is provided as part of the prompt. The docstring is shown in section 15. On this example, the VLLM planner has a handdrawn sketch image preloaded. For different usecases, loading can be 3D scans or FreeCAD project files.

14.1. More qualitative results on CAD question answering for the 2D Subset of SGPBench.

USER QUERY: You are given a FreeCAD project file that includes a 2D CAD sketch. The FreeCAD project is already loaded for you.

Question: What is the position of the smallest rectangle relative to the largest rectangle?
 A) Above B) Below
 C) left D) right

PLAN: To determine the position of the smallest rectangle relative to the largest rectangle in the sketch, I need to analyze the sketch. I will use the `sketch_recognizer` function to get a JSON representation of the sketch's geometries and constraints, as well as a rendered image of the sketch. This will help me identify the rectangles and their relative positions.

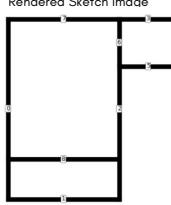
ACTION:

```
# Use the sketch_recognizer function to analyze the current state of the sketch
sketch_recognizer(sketch)
```

ENVIRONMENT: Execution Success

Rendered Sketch Image

Returns primitive parameters in JSON format here.



PLAN: The JSON data and the rendered image show two rectangles. The largest rectangle is defined by the lines with IDs 0, 1, 2, and 7. The smallest rectangle is defined by the lines with IDs 3, 4, 5, and 6. The smallest rectangle is positioned to the right of the largest rectangle.

ANSWER: D, TERMINATE.

USER QUERY: You are given a FreeCAD project file that includes a 2D CAD sketch. The FreeCAD project is already loaded for you.

Question: How many right angles are present in the CAD object?
 A) Two B) Four
 C) Six D) Eight

PLAN: To determine the number of right angles in the CAD object, I need to analyze the sketch. I will use the `sketch_recognizer` function to obtain a JSON representation of the sketch's geometries and constraints, as well as a rendered image of the sketch. This will help me identify the number of right angles present in the object.

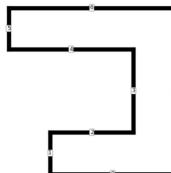
ACTION:

```
# Use the sketch_recognizer function to analyze the current state of the sketch
sketch_recognizer(sketch)
```

ENVIRONMENT: Execution Success

Rendered Sketch Image

Returns primitive parameters in JSON format here.



PLAN: The sketch consists of a series of connected line segments forming a shape. To determine the number of right angles, I need to examine the connections between these line segments. Each connection between two perpendicular lines forms a right angle.

From the JSON data and the rendered image, I can identify the following right angles:

1. Between line segments 0 and 1
 2. Between line segments 1 and 2
 3. Between line segments 2 and 3
 4. Between line segments 3 and 4
 5. Between line segments 4 and 5
 6. Between line segments 5 and 6
 7. Between line segments 6 and 7
 8. Between line segments 7 and 0
- There are eight right angles in total.

ANSWER: D, TERMINATE.

USER QUERY: You are given a FreeCAD project file that includes a 2D CAD sketch. The FreeCAD project is already loaded for you.

Question: What type of triangle is shown in the image based on its side lengths?
 A) Equilateral B) Isosceles
 C) Scalene D) Right

PLAN: To determine the type of triangle based on its side lengths, I need to examine the sketch and identify the lengths of its sides. I will use the `sketch_recognizer` function to analyze the sketch and obtain a JSON representation of its geometric elements, which will help me understand the side lengths of the triangle.

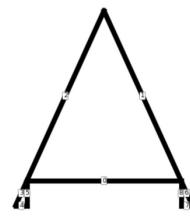
ACTION:

```
# Use the sketch_recognizer function to analyze the current state of the sketch
sketch_recognizer(sketch)
```

ENVIRONMENT: Execution Success

Rendered Sketch Image

Returns primitive parameters in JSON format here.



PLAN: The sketch shows a triangle with three main line segments: 0, 1, and 2. I need to calculate the lengths of these segments to determine the type of triangle.

ACTION:

```
def calculate_length(start, end):
    return math.sqrt((end['x'] - start['x'])**2 + (end['y'] - start['y'])**2)

# Extract the start and end points of the triangle's sides
line_0 = {"start": {"x": -0.395, "y": -0.3723}, "end": {"x": 0.3956, "y": -0.3723}}
line_1 = {"start": {"x": 0.3959, "y": -0.3728}, "end": {"x": 0.0004, "y": 0.4993}}
line_2 = {"start": {"x": 0.0001, "y": 0.5}, "end": {"x": -0.3955, "y": -0.3721}}
```

Calculate the lengths of the sides
length_0 = calculate_length(line_0['start'], line_0['end'])
length_1 = calculate_length(line_1['start'], line_1['end'])
length_2 = calculate_length(line_2['start'], line_2['end'])

length_0, length_1, length_2

ENVIRONMENT: Execution Success

Output:

(0.7906, 0.957590056, 0.9576313656)

PLAN: The lengths of the triangle's sides are approximately 0.7906, 0.9576, and 0.9576. Since two sides are equal, the triangle is isosceles.

ANSWER: B) Isosceles, TERMINATE

Figure 8. Complete agent trajectories of the CAD-Assistant for CAD Question Answering on the 2D subset of SGPBench.

14.2. More qualitative results on CAD question answering for the 3D Subset of SGPBench.

USER QUERY: You are given a FreeCAD project file that includes a 3D CAD model. The FreeCAD project is already loaded for you.
Question: What is the shape of the main body of the CAD object?
A) Cylinder B) Sphere
C) L-shape D) Cube

PLAN: To determine the shape of the main body of the CAD object, I need to analyze the 3D model in the FreeCAD project. I will use the `solid_recognizer` function to get a visual representation and JSON serialization of the model's attributes. This will help me understand the shape of the object.

ACTION:

```
# Use the solid_recognizer to analyze the state of the loaded CAD document
solid_recognizer(doc)
```

ENVIRONMENT: Execution Success

Rendered 3D Solid

Returns primitive parameters in JSON format here.



PLAN: The rendered image of the 3D model shows an L-shaped object. This matches the description of an L-shape, which is one of the options provided in the multiple-choice question.

ANSWER: C, TERMINATE.

USER QUERY: You are given a FreeCAD project file that includes a 3D CAD model. The FreeCAD project is already loaded for you.
Question: What is the primary shape of the top part of the CAD object?
A) Square B) Triangle
C) Circle D) Arch

PLAN: To answer the question about the primary shape of the top part of the CAD object, I need to examine the 3D CAD model. I will use the `solid_recognizer` function to analyze the model and obtain a rendered image of the object. This will help me identify the primary shape of the top part.

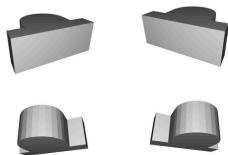
ACTION:

```
# Use the solid_recognizer to analyze the state of the loaded CAD document
solid_recognizer(doc)
```

ENVIRONMENT: Execution Success

Rendered 3D Solid

Returns primitive parameters in JSON format here.



PLAN: The rendered image of the 3D model shows that the top part of the CAD object has a curved shape, resembling an arch. The JSON data also indicates the presence of an arc in the sketch geometry, which supports this observation.

ANSWER: D) Arch TERMINATE.

USER QUERY: You are given a FreeCAD project file that includes a 3D CAD model. The FreeCAD project is already loaded for you.
Question: How many holes are visible on the flange of the CAD object?
A) Two B) Three
C) Four D) Five

PLAN: To determine the number of holes visible on the flange of the CAD object, I need to analyze the 3D model. I will use the `solid_recognizer` function to get a visual rendering and JSON serialization of the CAD model, which will help me identify the number of holes.

ACTION:

```
# Use the solid_recognizer to analyze the state of the loaded CAD document
solid_recognizer(doc)
```

ENVIRONMENT: Execution Success

Rendered 3D Solid

Returns primitive parameters in JSON format here.



PLAN: The JSON data and the rendered image show that there are four circles in the sketches, which likely correspond to holes on the flange. The image confirms the presence of four holes.

ANSWER: C, TERMINATE.

USER QUERY: You are given a FreeCAD project file that includes a 3D CAD model. The FreeCAD project is already loaded for you.
Question: What basic geometric shape forms the primary body of the CAD object?
A) Cylinder B) Cube
C) Sphere D) Pyramid

PLAN: To determine the primary geometric shape of the CAD object, I need to analyze the 3D model. I will use the `solid_recognizer` function to obtain a rendered image and JSON representation of the CAD model. This will help me identify the basic geometric shape.

ACTION:

```
# Use the solid_recognizer to analyze the state of the loaded CAD document
solid_recognizer(doc)
```

ENVIRONMENT: Execution Success

Rendered 3D Solid

Returns primitive parameters in JSON format here.



PLAN: The rendered image shows a cylindrical shape as the primary body of the CAD object. The JSON data also indicates the presence of arcs and line segments that form circular profiles, which are extruded to create a cylindrical shape.

ANSWER: A, TERMINATE.

Figure 9. Complete agent trajectories of the CAD-Assistant for CAD Question Answering on the 3D subset of SGPBench.

14.3. More qualitative results on CAD sketch autoconstraining.

USER QUERY: Analyze the provided sketch to identify geometric relationships and apply the appropriate parametric constraints, ensuring that all elements are properly constrained according to design principles. The `sketch` object is already loaded for you. Use the Coincident, Parallel, Equal, Vertical, Tangent, Perpendicular and Horizontal constraints.

PLAN: To analyze the sketch and apply the appropriate constraints, I will first use the `sketch_recognizer` function to understand the current geometries and constraints in the sketch. This will help me identify the geometric relationships and determine which constraints to apply

ACTION:

```
# Use the sketch_recognizer to analyze the state of the loaded CAD sketch
sketch_recognizer(sketch)
```

ENVIRONMENT: Execution Success

Rendered Sketch Image



Returns primitive parameters in JSON format here.

PLAN: The sketch contains multiple line segments, arcs, and points. To ensure the sketch is fully constrained, I will apply the following constraints: 1. Coincident constraints to connect endpoints of line segments and arcs. 2. Horizontal and vertical constraints for line segments that are aligned with the axes. 3. Parallel constraints for line segments that should remain parallel. 4. Perpendicular constraints for line segments that should be perpendicular. 5. Tangent constraints for arcs and line segments that should be tangent. I will analyze each constraint before applying it to ensure it is valid and does not move the geometries.

ACTION:

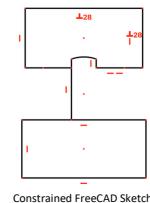
```
# Define constraints to be analyzed and potentially applied
constraints_to_analyze = [
    Sketcher.Constraint('Coincident', 0, SubRef.END.value, 1, SubRef.END.value),
    Sketcher.Constraint('Coincident', 1, SubRef.START.value, 3, SubRef.START.value),
    Sketcher.Constraint('Coincident', 3, SubRef.END.value, 2, SubRef.START.value),
    Sketcher.Constraint('Coincident', 2, SubRef.END.value, 4, SubRef.START.value),
    Sketcher.Constraint('Coincident', 4, SubRef.END.value, 8, SubRef.END.value),
    Sketcher.Constraint('Coincident', 8, SubRef.END.value, 6, SubRef.START.value),
    Sketcher.Constraint('Coincident', 6, SubRef.START.value, 9, SubRef.START.value),
    Sketcher.Constraint('Coincident', 9, SubRef.END.value, 7, SubRef.START.value),
    Sketcher.Constraint('Horizontal', 0),
    Sketcher.Constraint('Horizontal', 1),
    Sketcher.Constraint('Horizontal', 2),
    Sketcher.Constraint('Horizontal', 10),
    Sketcher.Constraint('Horizontal', 11),
    Sketcher.Constraint('Vertical', 3),
    Sketcher.Constraint('Vertical', 4),
    Sketcher.Constraint('Vertical', 6),
    Sketcher.Constraint('Vertical', 7),
    Sketcher.Constraint('Vertical', 12),
    Sketcher.Constraint('Vertical', 13),
    Sketcher.Constraint('Parallel', 0, 1),
    Sketcher.Constraint('Parallel', 2, 10),
    Sketcher.Constraint('Parallel', 4, 12),
    Sketcher.Constraint('Parallel', 4, 13),
    Sketcher.Constraint('Perpendicular', 4, 2),
    Sketcher.Constraint('Tangent', 9, 6),
    Sketcher.Constraint('Tangent', 9, 7)
]

# Analyze and apply constraints
for i, constraint in enumerate(constraints_to_analyze):
    is_valid, moves_geometries = analyse_constraint(sketch, constraint)
    if is_valid and not moves_geometries:
        sketch.addConstraint(constraint)

sketch.recompute()
```

ENVIRONMENT: Execution Success

PLAN: The constraints have been successfully analyzed and applied without moving the geometries. The sketch should now be fully constrained according to the design principles. The sketch has been analyzed and the appropriate constraints have been applied. The sketch is now fully constrained with coincident, horizontal, vertical, parallel, perpendicular, and tangent constraints. **TERMINATE.**



Constrained FreeCAD Sketch

USER QUERY: Analyze the provided sketch to identify geometric relationships and apply the appropriate parametric constraints, ensuring that all elements are properly constrained according to design principles. The `sketch` object is already loaded for you. Use the Coincident, Parallel, Equal, Vertical, Tangent, Perpendicular and Horizontal constraints.

PLAN: To analyze the sketch and apply parametric constraints, I will first use the `sketch_recognizer` function to understand the current geometries and constraints in the sketch. This will provide a JSON representation and an image rendering of the sketch, which will help in identifying the geometric relationships and determining the necessary constraints.

ACTION:

```
# Use the sketch_recognizer to analyze the state of the loaded CAD sketch
sketch_recognizer(sketch)
```

ENVIRONMENT: Execution Success

Rendered Sketch Image



PLAN: The sketch contains multiple circles and line segments. To ensure proper constraints, I will: 1. Apply Coincident constraints to ensure circles share centers where needed. 2. Apply Parallel constraints to line segments that should be parallel. 3. Apply Equal constraints to circles with the same radius. 4. Apply Vertical and Horizontal constraints to line segments that should be aligned accordingly. 5. Ensure loops are closed with Coincident constraints. I'll start by analyzing and applying Coincident constraints for shared centers and endpoints.

ACTION:

```
# Coincident constraints for shared centers
coincident_constraints = [
    Sketcher.Constraint('Coincident', 0, SubRef.CENTER.value, 1, SubRef.CENTER.value),
    Sketcher.Constraint('Coincident', 2, SubRef.CENTER.value, 5, SubRef.CENTER.value),
    Sketcher.Constraint('Coincident', 3, SubRef.CENTER.value, 4, SubRef.CENTER.value)
]

# Coincident constraints for endpoints to close loops
coincident_constraints += [
    Sketcher.Constraint('Coincident', 6, SubRef.START.value, 9, SubRef.START.value),
    Sketcher.Constraint('Coincident', 6, SubRef.END.value, 9, SubRef.END.value),
    Sketcher.Constraint('Coincident', 7, SubRef.START.value, 8, SubRef.END.value),
    Sketcher.Constraint('Coincident', 7, SubRef.END.value, 9, SubRef.END.value)
]

# Analyze and apply constraints
for i, constraint in enumerate(coincident_constraints):
    is_valid, moves_geometries = analyse_constraint(sketch, constraint)
    if is_valid and not moves_geometries:
        sketch.addConstraint(constraint)

sketch.recompute()
```

ENVIRONMENT: Execution Success

PLAN: The Coincident constraints have been successfully applied. Next, I will apply Parallel, Equal, Vertical, and Horizontal constraints. 1. Apply Parallel constraints to line segments that should be parallel. 2. Apply Equal constraints to circles with the same radius. 3. Apply Vertical and Horizontal constraints to line segments that should be aligned accordingly.

ACTION:

```
# Parallel constraints for line segments
parallel_constraints = [
    Sketcher.Constraint('Parallel', 6, 7),
    Sketcher.Constraint('Parallel', 8, 9)
]

# Equal constraints for circles with the same radius
equal_constraints = [
    Sketcher.Constraint('Equal', 0, 4),
    Sketcher.Constraint('Equal', 1, 2),
    Sketcher.Constraint('Equal', 2, 3)
]

# Vertical and Horizontal constraints
vertical_constraints = [
    Sketcher.Constraint('Vertical', 8),
    Sketcher.Constraint('Vertical', 9)
]

horizontal_constraints = [
    Sketcher.Constraint('Horizontal', 6),
    Sketcher.Constraint('Horizontal', 7)
]

# Analyze and apply constraints
for i, constraint in enumerate(parallel_constraints + equal_constraints + vertical_constraints + horizontal_constraints):
    is_valid, moves_geometries = analyse_constraint(sketch, constraint)
    if is_valid and not moves_geometries:
        sketch.addConstraint(constraint)

sketch.recompute()
```

ENVIRONMENT: Execution Success

PLAN: The constraints have been successfully applied, and the sketch is now fully constrained according to design principles. The sketch should now have proper geometric relationships and be robust against modifications. **TERMINATE.**

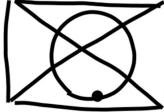


Constrained FreeCAD Sketch

Figure 10. Complete agent trajectories of the CAD-Assistant for CAD sketch autoconstraining

14.4. More qualitative results on handdrawn CAD sketch parameterization.

USER QUERY: Transform the provided hand-drawn image into a constrained CAD sketch on FreeCAD and add appropriate parametric constraints.



PLAN: I need to extract the geometric elements and constraints from the hand-drawn image using the `sketch_from_image` function. Then, I'll analyze each constraint to ensure they are valid and do not move the geometries before applying them to the FreeCAD sketch.

ACTION:

```
# Use the sketch_from_image function to understand the current geometries in the sketch
sketch_from_image(img)
```

ENVIRONMENT: Execution Success

Returns primitive and constraint parameters in JSON format here.

PLAN: I need to create a FreeCAD sketch and add the detected geometries. Then, I'll analyze each constraint to ensure they are valid and do not move the geometries before applying them.

ACTION:

```
# Create a new FreeCAD document and sketch
doc = App.newDocument("HandDrawnSketch")
sketch = doc.addObject('Sketcher::SketchObject', 'sketch')

# Add geometries to the sketch
line1 = Part.LineSegment(App.Vector(-0.4922, 0.3516, 0), App.Vector(0.4922, 0.3516, 0))
line2 = Part.LineSegment(App.Vector(-0.4922, -0.3516, 0), App.Vector(0.4922, -0.3516, 0))
circle = Part.Circle(App.Vector(0.0078, 0), App.Vector(0, 0, 0.2734))
line3 = Part.LineSegment(App.Vector(-0.4922, -0.3516, 0), App.Vector(0.4922, -0.3516, 0))
line4 = Part.LineSegment(App.Vector(0.4922, -0.3516, 0), App.Vector(0.4922, 0.3516, 0))
line5 = Part.LineSegment(App.Vector(0.4922, 0.3516, 0), App.Vector(-0.4922, 0.3516, 0))
line6 = Part.LineSegment(App.Vector(0.4922, 0.3516, 0), App.Vector(-0.4922, -0.3516, 0))

# Add geometries to the sketch
sketch.addGeometry(line1)
sketch.addGeometry(line2)
sketch.addGeometry(circle)
sketch.addGeometry(line3)
sketch.addGeometry(line4)
sketch.addGeometry(line5)
sketch.addGeometry(line6)

# Recompute the sketch to update
sketch.recompute()
```

ENVIRONMENT: Execution Success

PLAN: Now that the geometries are added to the sketch, I need to analyze each constraint to ensure they are valid and do not move the geometries before applying them.

ACTION:

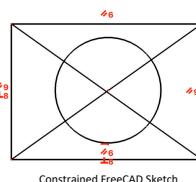
```
# Analyze and apply constraints
constraints = [
    Sketcher.Constraint('Parallel', 0, 5),
    Sketcher.Constraint('Coincident', 0, SubRef.START.value, 3, SubRef.START.value),
    Sketcher.Constraint('Coincident', 0, SubRef.END.value, 6, SubRef.END.value),
    Sketcher.Constraint('Perpendicular', 3, 5),
    Sketcher.Constraint('Parallel', 3, 6),
    Sketcher.Constraint('Horizontal', 5)
]

for i, constraint in enumerate(constraints):
    if is_valid, moves_geometries = analyse_constraint(sketch, constraint):
        if is_valid and not moves_geometries:
            sketch.addConstraint(constraint)
            sketch.recompute()
        else:
            print(f'Constraint [{i}] rejected: It causes unintended geometry movement.')
```

ENVIRONMENT: Execution Success

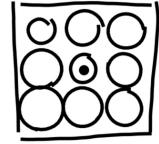
Constraint [1] rejected: It causes unintended geometry movement.

PLAN: The constraint that was rejected caused unintended geometry movement. I will proceed with the valid constraints and finalize the sketch. The hand-drawn image has been successfully transformed into a constrained CAD sketch in FreeCAD. The valid constraints were applied without causing unintended geometry movement TERMINATE.



Constrained FreeCAD Sketch

USER QUERY: Transform the provided hand-drawn image into a constrained CAD sketch on FreeCAD and add appropriate parametric constraints.



PLAN: To transform the hand-drawn image into a constrained CAD sketch, I'll first use the `sketch_from_image` function to extract geometric elements and constraints from the image. Then, I'll analyze each constraint using `analyse_constraint` to ensure they are valid and do not move the geometries before applying them.

ACTION:

```
# Use the sketch_from_image function to understand the current geometries in the sketch
sketch_from_image(img)
```

ENVIRONMENT: Execution Success

Returns primitive and constraint parameters in JSON format here.

PLAN: I have the detected geometries and constraints. Next, I'll create a FreeCAD sketch and add these geometries. Then, I'll analyze each constraint to ensure they are valid and do not move the geometries before applying them.

ACTION:

```
# Create a new FreeCAD document and sketch
doc = App.newDocument("TransformedSketch")
sketch = doc.addObject('Sketcher::SketchObject', 'sketch')

# Add geometries to the sketch
geometries = [
    Part.LineSegment(App.Vector(-0.4922, 0.4922, 0), App.Vector(0.4922, 0.4922, 0)),
    Part.LineSegment(App.Vector(0.4922, 0.4922, 0), App.Vector(0.4922, -0.4922, 0)),
    Part.Circle(App.Vector(0.0078, 0), App.Vector(0, 0, 0.2734)),
    Part.LineSegment(App.Vector(0.4922, 0.4922, 0), App.Vector(0.4922, -0.4922, 0)),
    Part.LineSegment(App.Vector(0.4922, 0.4922, 0), App.Vector(-0.4922, 0.4922, 0)),
    Part.LineSegment(App.Vector(0.4922, 0.4922, 0), App.Vector(-0.4922, -0.4922, 0)),
    Part.LineSegment(App.Vector(0.4922, -0.4922, 0), App.Vector(-0.4922, -0.4922, 0)),
    Part.LineSegment(App.Vector(0.4922, -0.4922, 0), App.Vector(-0.4922, 0.4922, 0)),
    Part.LineSegment(App.Vector(-0.4922, -0.4922, 0), App.Vector(-0.4922, 0.4922, 0)),
    Part.LineSegment(App.Vector(-0.4922, -0.4922, 0), App.Vector(0.4922, -0.4922, 0)),
    Part.LineSegment(App.Vector(0.4922, -0.4922, 0), App.Vector(0.4922, 0.4922, 0)),
    Part.LineSegment(App.Vector(-0.4922, 0.4922, 0), App.Vector(0.4922, 0.4922, 0)),
    Part.Circle(App.Vector(0.0078, 0), App.Vector(0, 0, 0.2734)),
    Part.LineSegment(App.Vector(0.4922, 0.4922, 0), App.Vector(0.4922, -0.4922, 0)),
    Part.LineSegment(App.Vector(0.4922, 0.4922, 0), App.Vector(-0.4922, 0.4922, 0)),
    Part.LineSegment(App.Vector(0.4922, 0.4922, 0), App.Vector(-0.4922, -0.4922, 0)),
    Part.LineSegment(App.Vector(0.4922, -0.4922, 0), App.Vector(-0.4922, -0.4922, 0)),
    Part.LineSegment(App.Vector(0.4922, -0.4922, 0), App.Vector(-0.4922, 0.4922, 0)),
    Part.LineSegment(App.Vector(-0.4922, -0.4922, 0), App.Vector(-0.4922, 0.4922, 0)),
    Part.LineSegment(App.Vector(-0.4922, -0.4922, 0), App.Vector(0.4922, -0.4922, 0)),
    Part.LineSegment(App.Vector(0.4922, -0.4922, 0), App.Vector(0.4922, 0.4922, 0)),
    Part.Circle(App.Vector(0.0078, 0), App.Vector(0, 0, 0.2734))
]

# Add each geometry to the sketch
for geometry in geometries:
    sketch.addGeometry(geometry)

# Recompute the sketch to apply changes
sketch.recompute()

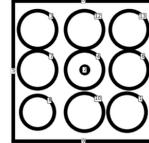
# Analyze and apply constraints
constraints = [
    Sketcher.Constraint('Parallel', 0, 5),
    Sketcher.Constraint('Coincident', 0, SubRef.START.value, 3, SubRef.END.value),
    Sketcher.Constraint('Coincident', 0, SubRef.END.value, 6, SubRef.END.value),
    Sketcher.Constraint('Parallel', 3, 5),
    Sketcher.Constraint('Horizontal', 5),
    Sketcher.Constraint('Vertical', 6, 9),
    Sketcher.Constraint('Perpendicular', 9, 11),
    Sketcher.Constraint('Horizontal', 11)
]

# Analyze and apply each constraint
for i, constraint in enumerate(constraints):
    is_valid, moves_geometries = analyse_constraint(sketch, constraint)
    if is_valid and not moves_geometries:
        sketch.addConstraint(constraint)
        sketch.recompute()
    else:
        print(f'Constraint [{i}] rejected: It causes unintended geometry movement.')
    # Recognize the sketch to verify
    sketch_recognizer(sketch)
]
```

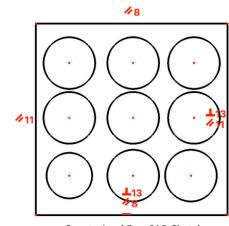
ENVIRONMENT: Execution Success

Rendered Sketch Image

Returns primitive parameters in JSON format here.



PLAN: The sketch has been successfully transformed with the geometries and constraints applied. The constraints were analyzed to ensure they are valid and do not move the geometries. The rendered image confirms the transformation. The hand-drawn image has been successfully transformed into a constrained CAD sketch in FreeCAD. All constraints were validated and applied without moving the geometries TERMINATE.



Constrained FreeCAD Sketch

Figure 11. Complete agent trajectories of the CAD-Assistant for handdrawn CAD sketch parameterization.

15. Docstrings

This section provides the complete docstring of the toolset available to the VLLM planner.

```
1 def sketch_recognizer(sketch: Sketch) -> None:
2     """Analyzes a 2D CAD sketch, providing both a JSON serialization of the Attributes for the geometry
3     and constraints and an image rendering of the sketch. Use this function to understand the current FreeCAD sketch.
4
5     This function processes the given sketch and returns a JSON representation describing its geometric
6     elements (e.g., lines, arcs, points, circles) and constraints from the `sketch.Geometry`
7     and `sketch.Constraints` lists. Attributes for Geometries and constraints are extracted directly from these lists,
8     serialized into a structured JSON format for easy interpretation.
9
10    Additionally, this function generates an image rendering of the sketch for visual examination. The sketch
11    is rendered as a black and white image. Primitives of the CAD sketch are labeled with a unique numerical ID,
12    shown by a marker that is positioned over the primitive.
13
14    Parameters:
15    -----
16    sketch (Sketch):
17        The input sketch object to be analyzed, containing the geometries and constraints.
18
19    Prints:
20    -----
21    Displays the sketch's parameters and constraints for quick review. It also returns a
22    sketch_image (PIL.Image.Image or np.ndarray) rendering of the sketch.
23
24    Usage Example:
25    -----
26    >>> sketch_recognizer(sketch)
27    The sketch contains the following geometries and constraints, serialized in JSON format:
28    {
29        "Geometry": [
30            {
31                "Index": 1,
32                "Type": "Line segment",
33                "StartPoint": {
34                    "x": ...,
35                    "y": ...
36                },
37                "EndPoint": {
38                    "x": ...,
39                    "y": ...
40                },
41                "isConstruction": ...
42            ],
43            "Constraints": [
44                { ... }
45            ]
46        }
47    Rendered image of the sketch:
48    [Image displays here]
49    """
50
51 def solid_recognizer(doc: App.Document) -> None:
52     """Analyzes a 3D CAD Model, providing both a JSON serialization of the Attributes for the geometry
53     and constraints and an image rendering of the sketch and extrude operations. Use this function to understand the current FreeCAD sketch.
54
55     This function processes the given sketch and returns a JSON representation describing its sketches and extrusions.
56     Attributes for Geometries and constraints are extracted directly from these lists,
57     serialized into a structured JSON format for easy interpretation.
58
59     Additionally, this function generates an image rendering of the 3D CAD model for visual examination. The sketch
60     is with multiple views.
61
62     Parameters:
63     -----
64     doc (App.Document):
65         The FreeCAD document including a list of Objects that can be sketch and extusion operations
66
67     Prints:
68     -----
69     Displays the sketch and extrusion parameters for quick review. It also returns a
70     cad_image (PIL.Image.Image or np.ndarray) rendering of the sketch.
71
72     Usage Example:
73     -----
74     >>> solid_recognizer(sketch)
75     The 3D CAD model contains the following sketch and extrusion operations, serialized in JSON format:
76     {
77         "Sketch0": [
78             "Geometry": [
79                 {
80                     "Index": 1,
81                     "Type": "Line segment",
82                     "StartPoint": {
83                         "x": ...,
84                         "y": ...
85                     },
86                     "EndPoint": {
87                         "x": ...,
88                         "y": ...
89                     },
90                     "isConstruction": ...
91                 ],
92                 "Constraints": [
93                     { ... }
94                 ]
95             },
96             "Extrusion1": { ... }
97         }
98     Rendered image of the 3D Model:
99     [Image displays here]
100    """
101
102
```

```

103
104 def sketch_from_image(img: PIL.Image.Image) -> None:
105     """Extracts a Sketch object from a given sketch image.
106
107     This function processes an input img and detects
108     parametric geometric primitives within the sketch (e.g., lines, circles, arcs, points) and the corresponding constraints. The function handle
109     input images as a PIL image. Detected geometric entities are printed and can be used for further analysis or manipulation.
110
111 Usage:
112 -----
113 This function is called a deep learning network that is imperfect and makes mistakes. Note that predicted constraints might be inaccurate and
114 applying them without analysing them might drastically change the sketch geometry. Use the provided 'analyse_constraint' function to make sure
115 that predicted constraints are valid and do not move geometric entities of the sketch.
116
117 Parameters:
118 -----
119 img (PIL.Image.Image or np.ndarray):
120     The input image of a handdrawn 2D CAD sketch.
121
122 Prints:
123 -----
124 A dictionary in JSON format containing the detected geometries and constraints.
125
126 Usage Example:
127 -----
128 >>> sketch_from_image(img)
129 The handdrawn parametrization tool detected the following sketch geometries and constraints, serialized in JSON format:
130 {
131     "Geometry": [
132         {
133             "Id": 1,
134             "Type": "Line segment",
135             "start_vector": {
136                 "x": ...,
137                 "y": ...
138             },
139             "end_vector": {
140                 "x": ...,
141                 "y": ...
142             },
143         },
144         "Constraints": [
145             { ... }
146         ]
147     ]
148 }
149 """
150 def get_crosssection_image(mesh: o3d.geometry.TriangleMesh, normal: np.ndarray, origin: np.ndarray) -> PIL.Image.Image
151 """
152 Generates a 2D cross-sectional image from a 3D mesh.
153
154 This function takes a 3D mesh and extracts a cross-section based on a specified plane, defined by a normal vector
155 and an origin point. The extracted cross-section is then projected onto a 2D plane and normalized to a fixed size
156 suitable for visualization. The resulting image is centered, cropped, and rescaled to 128x128 pixels, capturing the
157 silhouette of the cross-section.
158
159 Parameters:
160 -----
161 mesh : o3d.geometry.TriangleMesh
162     The 3D mesh from which the cross-section will be extracted. It should contain vertices and faces attributes.
163 normal : np.ndarray, shape (3,)
164     The normal vector defining the orientation of the cross-sectional plane.
165 origin : np.ndarray, shape (3,)
166     A point on the plane to define its position in 3D space.
167
168 Returns:
169 -----
170 img : PIL.Image.Image
171     A grayscale PIL Image object of size 128x128 representing the 2D cross-section of the mesh.
172 """
173
174 def analyse_constraint(sketch: Sketch, constraint: Constraint) -> (bool, bool):
175     """Evaluate the impact of a given constraint on a sketch without applying it,
176     and determine if it causes significant changes to the geometry.
177
178     This function returns two binary flags: one indicating whether the constraint is valid,
179     and another indicating if it would cause geometries to move. Use this function to analyze
180     the effect of constraints on CAD geometries and ensure they behave as intended before
181     adding them to the sketch.
182
183 Parameters:
184 -----
185 sketch (Sketch):
186     The original FreeCAD sketch object containing geometric elements and constraints.
187 constraint (Constraint):
188     The constraint to be evaluated.
189
190 Returns:
191 -----
192 (is_valid, moves_geometries): (bool, bool)
193     - 'is_valid': 'True' if the constraint does not introduce conflicts or invalid states;
194     - 'False' if the constraint is invalid.
195     - 'moves_geometries': 'True' if the constraint causes movement of one or more geometries;
196     - 'False' if no significant movement occurs.
197
198 Prints:
199 -----
200 Displays a summary of the effect of the constraint on the geometric entities of the Sketch.Geometry list.
201 It also displays the is_valid and moves_geometries binary flags.
202
203 """

```

```

205 Usage:
206 -----
207 Use this function to test constraints before committing them to the sketch.
208 This allows you to detect unintended movements or conflicts early in the design process.
209
210 Example:
211 -----
212 >>> # Add a coincident constraint to align the start of geometry 1 with the end point of geometry 2.
213 >>> coincident_constraint = Sketcher.Constraint('Coincident', *(1, SubRef.START.value, 2, SubRef.END.value))
214
215 # You can analyse the effect the constraint would have on the sketch geometry
216 >>> is_valid, moves_geometries = analyse_constraint(sketch, coincident_constraint) # the function automatically prints an analysis of the constraint.
217 Analysis of Constraint[0] (without applying it to the sketch):
218     Type: Coincident
219     Elements:
220         - First: 1
221         - FirstPos: START
222         - Second: 2
223         - SecondPos: END
224     Movement:
225         Sketch.Geometry[5]:
226             - START moved from: (-0.500, -0.407) to (-0.297, 0.407)
227     Moves Geometries: True
228     IsValid: True
229     >>> is_valid
230     True
231     >>> moves_geometries
232     True
233
234 Note:
235 -----
236 This function does not modify the original sketch. It only provides a preview of the
237 potential impact of the given constraint.
238 """
239
240 class Sketch:
241     """Represents a 2D sketch object in FreeCAD, used for creating and defining geometric shapes,
242     constraints, and profiles that can later be referenced in 3D operations (e.g., extrusion, revolution).
243
244     The Sketch class provides methods to add, modify, and constrain geometric elements such as
245     lines, arcs, circles, and points. Sketches serve as essential building blocks in parametric
246     modeling, allowing users to control the relationships between elements through constraints.
247
248     Attributes:
249     -----
250     Name (str):
251         The name of the sketch object.
252     Geometry (List[Union[LineSegment, ArcOfCircle, Circle, Point]]):
253         A list of geometric elements in the sketch (e.g., lines, arcs, circles, points).
254     Constraints (List[Sketcher.Constraint]):
255         A list of constraints applied to the sketch elements (e.g., coincidence, equality, tangency).
256     ConstraintCount (int):
257         The total number of constraints applied to the sketch.
258     GeometryCount (int):
259         The total number of geometric elements present in the sketch.
260     Placement (Placement):
261         Defines the position and orientation of the sketch in 3D space. This attribute allows
262         the sketch to be moved or rotated within the document, affecting how it will be aligned
263         with other objects in FreeCAD.
264     State (List[str]):
265         A list representing the current status of the sketch. Possible values include:
266         - 'Touched': The sketch has been modified since the last update.
267         - 'Untouched': The sketch has not been modified since its last valid state.
268         - 'Invalid': The sketch contains errors or unsatisfied constraints.
269
270     Usage Example:
271     -----
272     >>> import FreeCAD
273     >>> from FreeCAD import Part, Sketcher
274     >>> doc = FreeCAD.newDocument("ExampleDoc")
275     >>> sketch = doc.addObject('Sketcher::SketchObject', 'sketch')
276
277     Methods:
278     -----
279     addGeometry(self, geometry: Union[LineSegment, ArcOfCircle, Circle, Point]) -> int:
280         This method is used to adds a geometric element to the sketch.
281
282     Parameters:
283     -----
284     geometry:
285         a geometric element to be added on the sketch (e.g., lines, arcs, circles, points).
286
287     Returns:
288     -----
289     index (int):
290         The index of the added geometry on the sketch.Geometry list.
291
292     Usage Example:
293     -----
294     >>> line = Part.LineSegment( App.Vector(0.2, 0.3, 0), App.Vector(0.3, 0.2, 0))
295     >>> line_index = sketch.addGeometry(line) # A line is added.
296     >>> sketch.recompute()
297     >>> line = sketch.Geometry[line_index]
298
299     delGeometries(self, identifiers: List[int]) -> None:
300         Deletes one or more geometries from the sketch, based on their indices on the sketch.Geometry list.
301
302     Parameters:
303     -----
304     identifiers (List[int]):
305         A list of zero-based identifiers specifying which geometries to delete from the sketch.
306
307 """

```

```

308     Usage Example:
309     -----
310     >>> sketch.delGeometries([1]) # This will delete the geometry at index 1
311     >>> sketch.recompute()
312
313
314     addConstraint(self, constraint: Sketcher.Constraint) -> int:
315         Adds a constraint to the sketch.
316         Returns the index of the added constraint.
317
318     Parameters:
319     -----
320     constraint (Sketcher.Constraint):
321         a geometric constraint to be added on the sketch.
322
323     Returns:
324     -----
325     index (int):
326         The index of the added constraint on the sketch.Constraints list.
327
328     Usage Example:
329     -----
330     >>> parallel_constraint = Sketcher.Constraint('Parallel', 4, 6)
331     >>> sketch.addConstraint(parallel_constraint)
332     >>> sketch.recompute()
333
334     recompute(self) -> None:
335         Forces a recompute of the sketch to apply and update any pending changes.
336         This ensures that all modifications (such as added or deleted geometries and constraints)
337         are reflected in the document.
338
339     Parameters:
340     -----
341     None
342
343     Returns:
344     -----
345     None
346
347
348 class LineSegment:
349     """Represents a line defined by two endpoints in 3D space.
350     This class is part of the FreeCAD Part module and should be instantiated using 'Part.LineSegment'.
351
352     Constructor:
353     -----
354     __init__(self, start_vector: App.Vector, end_vector: App.Vector)
355         Initializes a LineSegment with specified start and end points in 3D space.
356
357     Parameters:
358     -----
359     start_vector (App.Vector):
360         A 3D vector representing the coordinates of the line's start point.
361     end_vector (App.Vector):
362         A 3D vector representing the coordinates of the line's end point.
363
364     Attributes:
365     -----
366     StartPoint (App.Vector):
367         The start point of the line segment.
368     EndPoint (App.Vector):
369         The end point of the line segment.
370
371     Usage Example:
372     -----
373     >>> # Create a LineSegment from start and end points.
374     >>> start_point = App.Vector(0, 0, 0)
375     >>> end_point = App.Vector(1, 1, 1)
376     >>> line = Part.LineSegment(start_point, end_point)
377     >>> line.StartPoint.x
378     0 # access the x coordinate of the start point
379
380
381 class Circle:
382     """Represents a circle in 3D space defined by a center point, a normal vector and a radius using FreeCAD's Vector objects.
383     This class is part of the FreeCAD Part module and should be instantiated using 'Part.Circle'.
384
385     Constructor:
386     -----
387     __init__(self, center_vector: App.Vector, normal_vector: App.Vector, radius: float):
388         Initializes a Circle with a specified center, normal vector, and radius.
389
390     Parameters:
391     -----
392     center_vector (App.Vector):
393         A 3D vector with the coordinates of the center point of the circle.
394     normal_vector (App.Vector):
395         A 3D vector representing the direction normal to the circle's plane.
396     radius (float):
397         The radius of the circle.
398
399     Attributes:
400     -----
401     Center : (App.Vector)
402         The center point of the circle.
403     Radius : (float)
404         The radius of the circle.
405
406     Usage Example:
407     -----
408     >>> center = App.Vector(6.0, 3.0, 0)
409     >>> normal = App.Vector(0, 0, 1)

```

```

410     >>> radius = 1.1
411     >>> circle = Part.Circle(center, normal, radius)
412     >>> circle.Center
413     Vector (6.0, 3.0, 0.0)
414     """
415
416 class Point:
417     """Represents a point in 3D space.
418     This class is part of the FreeCAD Part module and should be instantiated using 'Part.Point'.
419
420     Constructor:
421     -----
422     __init__(self, point_vector: App.Vector):
423         Constructor of the Point class
424
425     Parameters:
426     -----
427     point_vector (App.Vector):
428         A 3D vector with the coordinates of the point.
429
430     Attributes:
431     -----
432     X (float):
433         The x-coordinate of the point.
434     Y (float):
435         The y-coordinate of the point.
436     Z (float):
437         The z-coordinate of the point.
438
439     Usage Example:
440     -----
441     >>> point = Part.Point(App.Vector(1.0, 2.0, 3.0))
442     >>> point.X
443     1.0
444     """
445
446 class ArcOfCircle:
447     """Represents a circular arc derived from a given circle, defined by start and end angles in radians.
448     The arc is drawn counterclockwise from the start angle to the end angle. Angles are expressed in radians
449     where 0 radians correspond to the positive x-axis and increase counterclockwise.
450
451     This class is part of the FreeCAD Part module and should be instantiated using 'Part.ArcOfCircle'.
452
453     Constructor:
454     -----
455     __init__(self, circle: Circle, start_param: float, end_param: float):
456         Initializes an ArcOfCircle instance from a circle and specified start and end parameters.
457
458     Parameters:
459     -----
460     circle (Circle):
461         The Circle object from which the arc is derived.
462     start_param (float):
463         The starting parameter (angle in radians) on the circle's circumference that defines the beginning of the arc.
464     end_param (float):
465         The ending parameter (angle in radians) on the circle's circumference that defines the end of the arc.
466
467     Attributes:
468     -----
469     Radius : (float)
470         The radius of the circle from which the arc is derived.
471     StartPoint : (App.Vector)
472         The start point of the arc.
473     EndPoint : (App.Vector)
474         The end point of the arc.
475     Center : (App.Vector)
476         The center point of the circle from which the arc is derived.
477     FirstParameter : (float)
478         The start angle of the arc in radians.
479     LastParameter : (float)
480         The end angle of the arc in radians.
481
482     Usage Example:
483     -----
484     >>> #Create counterclockwise ArcOfCircle with center, radius, and start and end angles in radians.
485     >>> arc_center = App.Vector(0.0670, -0.0000, 0.0) # Center of the arc
486     >>> arc_radius = 0.0130 # Radius of the arc
487     >>> start_param = -1.6008 # Start parameter in radians
488     >>> end_param = -0.0000 # End parameter in radians
489     >>> arc_direction = App.Vector(0, 0, 1)
490     >>> # Create the arc using Part.ArcOfCircle
491     >>> arc = Part.ArcOfCircle(Part.Circle(arc_center, arc_radius), start_param, end_param)
492     """
493
494 class Arc:
495     """Represents an arc defined by a start point, an end point, and an intermediate point on the arc.
496
497     This class is part of the FreeCAD 'Part' module and should be instantiated using 'Part.Arc'.
498     The arc is uniquely determined by three points: the start, the end, and a point somewhere
499     on the arc (referred to as the midpoint, though it need not be the geometric middle). The
500     arc lies on the circle that passes through these three points.
501
502     After calling 'recompute()' on a FreeCAD sketch, an 'Arc' object is automatically
503     transformed into an 'ArcOfCircle' object. This is because FreeCAD optimizes the
504     geometry representation for arcs, converting them to arcs of circles after
505     the geometry is fully processed.
506
507     Constructor:
508     -----
509     __init__(self, start_vector: App.Vector, end_vector: App.Vector, mid_vector: App.Vector):
510         Initializes a Arc with specified start, end and mid points in 3D space.
511
512     """

```

```

512     parameters:
513     -----
514     start_vector (App.Vector):
515         A 3D vector representing the coordinates of the arc's start point.
516     end_vector (App.Vector):
517         A 3D vector representing the coordinates of the arc's end point.
518     mid_vector (App.Vector):
519         A 3D vector representing a point on the circumference of the arc.
520
521 Usage:
522 -----
523 Use this function to create ArcOfCircle objects from start, end and mid points.
524
525 Usage Example:
526 -----
527 >>> start_point = App.Vector(5.0, 0, 0)
528 >>> end_point = App.Vector(0, 5.0, 0)
529 >>> mid_point = App.Vector(3.54, 3.54, 0)
530 >>> arc = Part.Arc(start_point, mid_point, end_point)
531 <Arc object>
532 >>> sketch.addGeometry(arc)
533 >>> sketch.recompute()
534 >>> arc
535 <ArcOfCircle object>
536 """
537
538
539 class SubRef(Enum):
540     START = 1
541     END = 2
542     CENTER = 3
543
544
545 class Constraint:
546     """Represents a geometric constraint in a FreeCAD sketch.
547     Constraints define relationships between geometric elements (lines, arcs, circles, points), ensuring specific properties or behaviors.
548     Constraints can be created using the 'Sketch.addConstraint()' method.
549
550     Constructor:
551     -----
552     __init__(self, constraint_type: str, *args)
553         Initializes a Sketcher.Constraint instance with a specified type and parameters.
554
555     Parameters:
556     -----
557     constraint_type (str):
558         The type of constraint to apply. Supported types include:
559         - 'Coincident'
560         - 'Parallel'
561         - 'Equal'
562         - 'Vertical'
563         - 'Horizontal'
564         - 'Perpendicular'
565         - 'Tangent'
566
567     *args (varies):
568         Additional parameters specific to the constraint type. These define the geometries or points
569         to which the constraint applies and any additional constraint-specific requirements.
570
571 Usage:
572 -----
573 Supported Constraint Types and Their Arguments:
574 1. Coincident: Enforces that two points or vertices coincide (i.e., share the same location in space).
575     - args: ('Coincident', First, FirstPos, Second, SecondPos)
576         First (int):
577             The index of the first geometry.
578         FirstPos (int):
579             The vertex (1 for start, 2 for end, 3 for center) of 'Geometry_index1' to fulfill the constraint.
580         Second (int):
581             The index of the second geometry.
582         SecondPos (int):
583             The vertex (1 for start, 2 for end, 3 for center) of 'Geometry_index2' to fulfill the constraint.
584
585 2. Parallel: Ensures two lines remain parallel.
586     - args: ('Parallel', First, Second)
587         First (int):
588             The index of the first Line segment to be made parallel.
589         Second (int):
590             The index of the second Line segment to be made parallel.
591
592 3. Equal: Makes two lines or circles equal in length or radius.
593     - args: ('Equal', First, Second)
594         First (int):
595             The index of the first line segment or circle.
596         Second (int):
597             The index of the second line segment or circle.
598
599 4. Vertical: Forces a line segment to be vertical.
600     - args: ('Vertical', First)
601         First (int): The index of the Line segment.
602
603 5. Horizontal: Forces a line segment to be horizontal.
604     - args: ('Horizontal', First)
605         First : (int)
606             The index of the line segment.
607
608 6. Perpendicular: Ensures that two line segments are perpendicular.
609     - args: ('Perpendicular', Geometry_index1, Geometry_index2)
610         Geometry_index1 : (int)
611             The index of the first line segment.
612         Geometry_index2 : (int)
613             The index of the second line segment.
614

```

```

615     7. Tangent: Makes a line tangent to a curve.
616         - args: ('Tangent', Geometry_index1, Geometry_index2)
617             - Geometry_index1 : (int)
618                 The index of the first geometry.
619             - Geometry_index2 : (int)
620                 The index of the first geometry.
621
622     Usage Example:
623     -----
624     >>> coincident_constraint = Sketcher.Constraint('Coincident', *(1, SubRef.START.value, 2, SubRef.END.value))
625     >>> sketch.addConstraint(coincident_constraint)
626     >>> sketch.recompute()
627     >>> sketch.State
628     ['Touched']
629     # Check the total number of constraints applied to the sketch.
630     >>> print(f"Number of constraints: {len(sketch.Constraints)}")
631     Number of constraints: 2
632     # Remove the most recently added constraint.
633     >>> sketch.delConstraint(len(sketch.Constraints) - 1)
634     # Remove the constraint on specific index.
635     >>> sketch.delConstraint(coincident_constraint_index)
636     >>> sketch.addConstraint(Sketcher.Constraint('Horizontal', 0))
637     >>> sketch.recompute()
638     >>> sketch.State # Use the State Variable to ensure that all added constraints are valid.
639     ['Touched', 'Invalid']
640
641
642 class Extrusion:
643     """Represents an extrusion of a sketch in FreeCAD.
644     This class is part of the FreeCAD Part module and should be instantiated using a sketch object and the desired extrusion parameters.
645
646     Attributes:
647     -----
648     Base (Sketch):
649         The sketch object that is extruded into a 3D solid.
650     DirMode (str):
651         Direction mode of the extrusion (default: "Normal").
652     LengthFwd (float):
653         Forward extrusion length.
654     LengthRev (float):
655         Reverse extrusion length.
656     Solid (bool):
657         Whether the extrusion is a solid (default: True).
658     Reversed (bool):
659         Whether the extrusion direction is reversed.
660     Symmetric (bool):
661         Whether the extrusion is symmetric along the sketch plane.
662     TaperAngle (float):
663         Taper angle for the extrusion.
664     TaperAngleRev (float):
665         Reverse taper angle for the extrusion.
666
667     Usage Example:
668     -----
669     >>> extrude = doc.addObject('Part::Extrusion', 'Extrude')
670     >>> extrude.Base = sketch # extrude an existing sketch object
671     >>> extrude.DirMode = "Normal"
672     >>> extrude.DirLink = None
673     >>> extrude.LengthFwd = 10.00
674     >>> extrude.LengthRev = 0.0
675     >>> extrude.Solid = True
676     >>> extrude.Reversed = False
677     >>> extrude.Symmetric = False
678     >>> extrude.TaperAngle = 0.0
679     >>> extrude.TaperAngleRev = 0.0
680     >>> doc.recompute()
681
682
683 class Solid:
684     """Represents a 3D solid in FreeCAD, created as part of an 'Extrusion' operation.
685     This shape object holds the geometry of the extruded solid and provides access to various
686     geometric properties, as well as methods for performing transformations and boolean operations
687     with other shapes.
688
689     Attributes:
690     -----
691     TypeId (str):
692         The type of shape, typically "Part::TopoShape".
693     Volume (float):
694         The volume of the extruded solid.
695     Area (float):
696         The total surface area of the extruded solid.
697     BoundBox (BoundBox):
698         The bounding box of the shape, describing the spatial limits of the extrusion.
699
700     Methods:
701     -----
702     fuse(shape: Shape) -> Shape:
703         Performs a union operation, merging this shape with another solid to create a combined shape.
704
705     Parameters:
706     -----
707     shape (Shape):
708         The other solid shape with which to perform the union operation.
709
710     Returns:
711     -----
712     Shape:
713         A new solid object representing the union of this shape and the specified shape.
714
715     Usage Example:
716     -----
717     >>> shape1 = extrude1.Shape # Access the extruded shape

```

```

717     """ shape1 = extrude1.Shape # Access the extruded shape
718     >>> shape2 = extrude2.Shape # Access the extruded shape
719     >>> result_shape = shape2.fuse(shape1) # Union with another shape
720
721     cut(shape: Shape) -> Shape:
722         """ Performs a cut operation, subtracting the specified shape from this shape.
723
724             Parameters:
725             -----
726             shape (Shape):
727                 The solid shape to subtract from this shape.
728
729             Returns:
730             -----
731             Shape:
732                 A new solid object representing the result of subtracting the specified shape from this shape.
733
734             Usage Example:
735             -----
736             >>> result_shape = shape1.cut(shape2) # Subtract shape2 from shape1
737
738     common(shape: Shape) -> Shape:
739         """ Performs an intersection operation, keeping only the volume that is common between this shape and another.
740
741             Parameters:
742             -----
743             shape (Shape):
744                 The solid shape to intersect with this shape.
745
746             Returns:
747             -----
748             Shape:
749                 A new solid object representing the intersected volume of the two shapes.
750
751             Usage Example:
752             -----
753             >>> result_shape = shape1.common(shape2) # Intersect shape1 with shape2
754 """

```