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 equiped 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 overlayed on the rendered geometries. Motivated by [64], 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 [57] 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 $\{\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_n\} \in \mathcal{P}^n$ and m edges between nodes $\{\mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_m\} \in \mathcal{C}^m$ 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:

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

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 $(s_i, s_j) \in [\![1..4]\!]^2$, 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, 61], 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 DeepCAD dataset [61]. 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 [61], 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, 61]. 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.

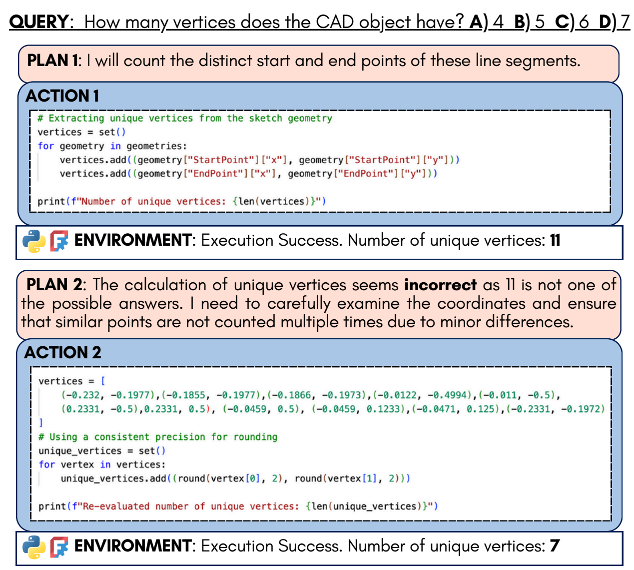


Figure 6. Correction of an inaccurate answer for a CQA example.

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 6 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 8.

13. Beyond Simplified CAD Commands

Extending the discussion of Sec. 4.4, we provide an additional qualitative example of the proposed `CAD-Assistant`. Figure 7 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.

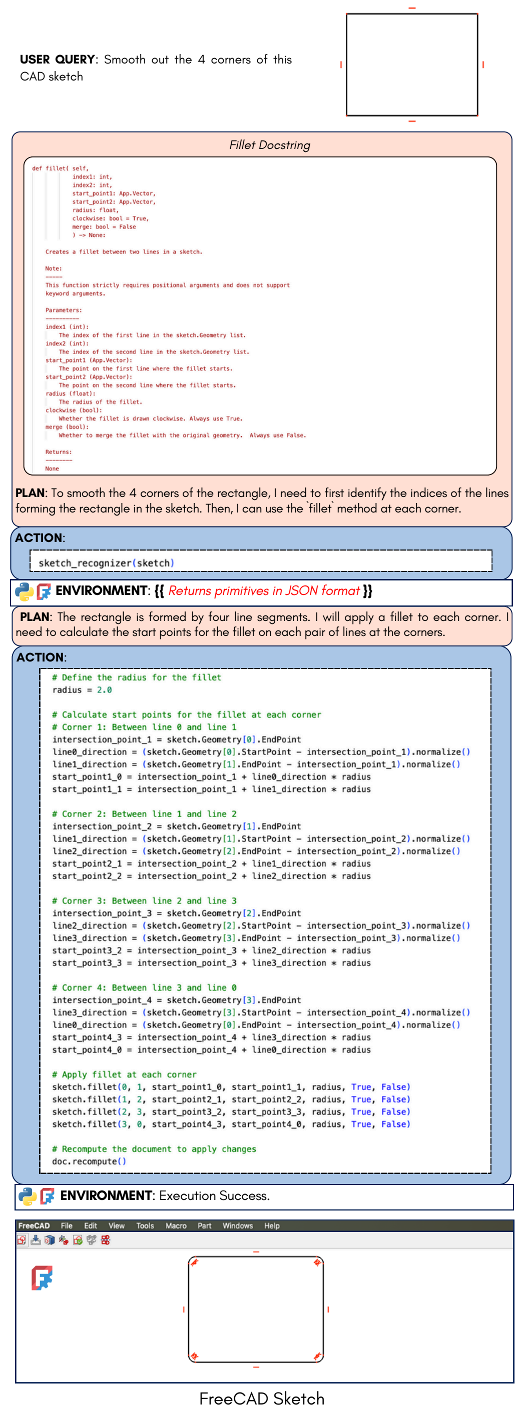


Figure 7. Example of the proposed `CAD-Assistant` utilizing the *Fillet* CAD command.

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

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:

```python
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 8. 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 inludes a 2D CAD sketch. The FreeCAD project is already loaded for you.
Question: What is the position of the smallest rectangle relative to the largets 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.

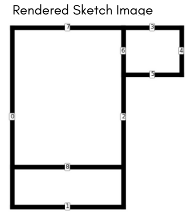
ACTION:

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

ENVIRONMENT: Execution Success

Returs primitive parameters in JSON format here.

Rendered Sketch Image



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 inludes 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

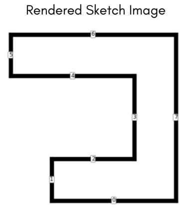
ACTION:

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

ENVIRONMENT: Execution Success

Returs primitive parameters in JSON format here.

Rendered Sketch Image



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 inludes 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.

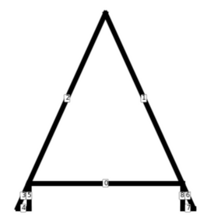
ACTION:

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

ENVIRONMENT: Execution Success

Returs primitive parameters in JSON format here.

Rendered Sketch Image



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 9. 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 inludes 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

Returs primitive parameters in JSON format here.

Rendered 3D Solid

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 inludes 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.

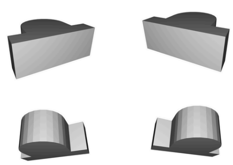
ACTION:

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

ENVIRONMENT: Execution Success

Returs primitive parameters in JSON format here.

Rendered 3D Solid



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 inludes 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

Returs 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 inludes 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.

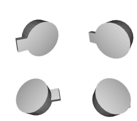
ACTION:

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

ENVIRONMENT: Execution Success

Rendered 3D Solid

Returs 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 10. 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

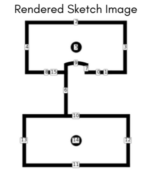
ACTION:

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

ENVIRONMENT: Execution Success

Returs primitive parameters in JSON format here.

Rendered Sketch Image



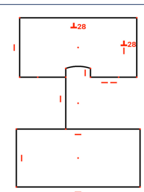
PLAN: The sketch contains multiple line segments, arcs, and points. To ensure the sketch is fully constrained, I will apply the following constraints: 1. Coincident constraints to connect endpoints of line segments and arcs. 2. Horizontal and vertical constraints for line segments that are aligned with 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.START.value, 6, SubRef.END.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('Coincident', 7, SubRef.END.value, 0, 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', 3, 12),
    Sketcher.Constraint('Parallel', 4, 13),
    Sketcher.Constraint('Perpendicular', 3, 2),
    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.

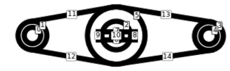
ACTION:

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

ENVIRONMENT: Execution Success

Returs primitive parameters in JSON format here.

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, 8, SubRef.START.value),
    Sketcher.Constraint('Coincident', 6, SubRef.END.value, 9, SubRef.START.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

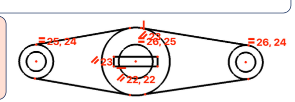
PLAN: The Coincident constraints have been successfully applied. Next, I will apply Parallel, Equal, Vertical, and Horizontal constraints. 1. Apply Parallel constraints to line segments that should be parallel. 2. Apply Equal constraints to circles with the same radius. 3. Apply Vertical and Horizontal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

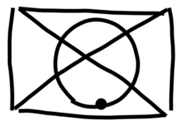


Constrained FreeCAD Sketch

Figure 11. 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))
line3 = Part.LineSegment(App.Vector(-0.4922, -0.3516, 0), App.Vector(-0.4922, 0.3516, 0))
circle = Part.Circle(App.Vector(0.0078, 0.0078, 0), App.Vector(0, 0, 1), 0.2734)
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(Part.Point(App.Vector(0.0078, -0.2734, 0)))
sketch.addGeometry(line2)
sketch.addGeometry(line3)
sketch.addGeometry(circle)
sketch.addGeometry(line4)
sketch.addGeometry(line5)
sketch.addGeometry(line6)

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

ENVIRONMENT: Execution Success

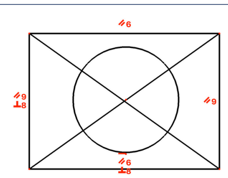
PLAN: Now that the geometries are added to the sketch, I need to analyze each constraint to ensure they are valid and do not move the geometries before applying them.

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):
    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.')
```

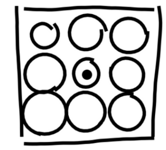
ENVIRONMENT: Execution Success
Constraint [1] rejected: It causes unintended geometry movement.

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.

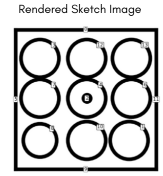
ACTION:

```
# 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.Circle(App.Vector(0.3047, -0.2891, 0), App.Vector(0, 0, 1), 0.1328),
    Part.Point(App.Vector(0.0078, 0.0078, 0)),
    Part.Circle(App.Vector(-0.3203, 0.2891, 0), App.Vector(0, 0, 1), 0.1328),
    Part.Circle(App.Vector(0.0078, 0.0078, 0), App.Vector(0, 0, 1), 0.1328),
    Part.LineSegment(App.Vector(-0.4922, -0.4922, 0), App.Vector(-0.4922, 0.4922, 0)),
    Part.Circle(App.Vector(0.3203, 0.0078, 0), App.Vector(0, 0, 1), 0.1328),
    Part.Circle(App.Vector(-0.3203, 0.0078, 0), App.Vector(0, 0, 1), 0.1328),
    Part.Circle(App.Vector(-0.3203, -0.2891, 0), App.Vector(0, 0, 1), 0.1172),
    Part.LineSegment(App.Vector(-0.4922, -0.4922, 0), App.Vector(0.4922, -0.4922, 0)),
    Part.Circle(App.Vector(0.0078, -0.2891, 0), App.Vector(0, 0, 1), 0.1328),
    Part.LineSegment(App.Vector(0.4922, -0.4922, 0), App.Vector(0.4922, 0.4922, 0)),
    Part.Circle(App.Vector(0.0078, 0.2891, 0), App.Vector(0, 0, 1), 0.1328),
    Part.Circle(App.Vector(0.3203, 0.2891, 0), App.Vector(0, 0, 1), 0.1328)
]

# 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, 9),
    Sketcher.Constraint('Coincident', 0, SubRef.START.value, 5, SubRef.END.value),
    Sketcher.Constraint('Coincident', 0, SubRef.END.value, 11, SubRef.END.value),
    Sketcher.Constraint('Parallel', 5, 11),
    Sketcher.Constraint('Coincident', 5, SubRef.END.value, 0, SubRef.START.value),
    Sketcher.Constraint('Horizontal', 9),
    Sketcher.Constraint('Perpendicular', 9, 11),
    Sketcher.Constraint('Coincident', 9, SubRef.START.value, 5, SubRef.START.value)
]

# 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)
```

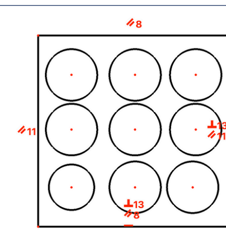
ENVIRONMENT: Execution Success

Returs primitive parameters in JSON format here.

Rendered Sketch Image



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 12. 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.

```python
def sketch_recognizer(sketch: Sketch) -> None:
    """Analyzes a 2D CAD sketch, providing both a JSON serialization of the Attributes for the geometry
    and constraints and an image rendering of the sketch. Use this function to understand the current FreeCAD sketch.

    This function processes the given sketch and returns a JSON representation describing its geometric
    elements (e.g., lines, arcs, points, circles) and constraints from the `sketch.Geometry`
    and `sketch.Constraints` lists. Attributes for Geometries and constraints are extracted directly from these lists,
    serialized into a structured JSON format for easy interpretation.

    Additionally, this function generates an image rendering of the sketch for visual examination. The sketch
    is rendered as a black and white image. Primitives of the CAD sketch are labeled with a unique numerical ID,
    shown by a marker that is positioned over the primitive.

    Parameters:
    ----------
    sketch (Sketch):
        The input sketch object to be analyzed, containing the geometries and constraints.

    Prints:
    -------
    Displays the sketch's parameters and constraints for quick review. It also returns a
    sketch_image (PIL.Image.Image or np.ndarray) rendering of the sketch.

    Usage Example:
    --------------
    >>> sketch_recognizer(sketch)
    The sketch contains the following geometries and constraints, serialized in JSON format:
    {
        "Geometry": [
            {
                "Index": 1,
                "Type": "Line segment",
                "StartPoint": {
                    "x": ...,
                    "y": ...
                },
                "EndPoint": {
                    "x": ...,
                    "y": ...
                },
                "isConstruction": ...
        ],
        "Constraints": [
            { ... }
        ]
    }
    Rendered image of the sketch:
    [Image displays here]
    """

def solid_recognizer(doc: App.Document) -> None:
    """Analyzes a 3D CAD Model, providing both a JSON serialization of the Attributes for the geometry
    and constraints and an image rendering of the sketch and extrude opeartions. Use this function to understand the current FreeCAD sketch.

    This function processes the given sketch and returns a JSON representation describing its sketches and extrusions.
    Attributes for Geometries and constraints are extracted directly from these lists,
    serialized into a structured JSON format for easy interpretation.

    Additionally, this function generates an image rendering of the 3D CAD model for visual examination. The sketch
    is with multiple views.

    Parameters:
    ----------
    doc (App.Document):
        The FreeCAD document including a list of Objects that can be sketch and extusion operations

    Prints:
    -------
    Displays the sketch and extrusion parameters for quick review. It also returns a
    cad_image (PIL.Image.Image or np.ndarray) rendering of the sketch.

    Usage Example:
    --------------
    >>> solid_recognizer(sketch)
    The 3D CAD model contains the following sketch and extrusion operations, serialized in JSON format:
    {
        "Sketch0": {
            "Geometry": [
                {
                    "Index": 1,
                    "Type": "Line segment",
                    "StartPoint": {
                        "x": ...,
                        "y": ...
                    },
                    "EndPoint": {
                        "x": ...,
                        "y": ...
                    },
                    "isConstruction": ...
            ],
            "Constraints": [
                { ... }
            ]
        },
        "Extrusion1": { ... }
    }
    Rendered image of the 3D Model:
    [Image displays here]
    """
```

```python

def sketch_from_image(img: PIL.Image.Image) -> None:
    """Extracts a Sketch object from a given sketch image.

    This function processes an input imag and detects
    parametric geometric primitives within the sketch (e.g., lines, circles, arcs, points) and the corresponding constraints. The function handle
    input images as a PIL image. Detected geometric entities are printed and can be used for further analysis or manipulation.

    Usage:
    ------
    This function is called a deep learning network that is imperfect and makes mistakes. Note that predicted constraints might be inaccurate and
    applying them without analysing them might drastically change the sketch geometry. Use the provided `analyse_constraint` function to make sure
    that predicted constraints are valid and do not move geometric entities of the sketch.

    Parameters:
    ----------
    img (PIL.Image.Image or np.ndarray):
        The input image of a handdrawn 2D CAD sketch.

    Prints:
    -------
    A dictionary in JSON format containing the detected geometies and constraints.

    Usage Example:
    --------------
    >>> sketch_from_image(img)
    The handdrawn parametarization tool detected the following sketch geometries and constraints, serialized in JSON format:
    {
        "Geometry": [
            {
                "Id": 1,
                "Type": "Line segment",
                "start_vector": {
                    "x": ...,
                    "y": ...
                },
                "end_vector": {
                    "x": ...,
                    "y": ...
                },
        ],
        "Constraints": [
            { ... }
        ]
    }
    """

def get_crosssection_image(mesh: o3d.geometry.TriangleMesh, normal: np.ndarray, origin: np.ndarray) -> PIL.Image.Image
    """
    Generates a 2D cross-sectional image from a 3D mesh.

    This function takes a 3D mesh and extracts a cross-section based on a specified plane, defined by a normal vector
    and an origin point. The extracted cross-section is then projected onto a 2D plane and normalized to a fixed size
    suitable for visualization. The resulting image is centered, cropped, and rescaled to 128x128 pixels, capturing the
    silhouette of the cross-section.

    Parameters:
    ----------
    mesh : o3d.geometry.TriangleMesh
        The 3D mesh from which the cross-section will be extracted. It should contain vertices and faces attributes.
    normal : np.ndarray, shape (3,)
        The normal vector defining the orientation of the cross-sectional plane.
    origin : np.ndarray, shape (3,)
        A point on the plane to define its position in 3D space.

    Returns:
    -------
    img : PIL.Image.Image
        A grayscale PIL Image object of size 128x128 representing the 2D cross-section of the mesh.
    """

def analyse_constraint(sketch: Sketch, constraint: Constraint) -> (bool, bool):
    """Evaluate the impact of a given constraint on a sketch without applying it,
    and determine if it causes significant changes to the geometry.

    This function returns two binary flags: one indicating whether the constraint is valid,
    and another indicating if it would cause geometries to move. Use this function to analyze
    the effect of constraints on CAD geometries and ensure they behave as intended before
    adding them to the sketch.

    Parameters:
    -----------
    sketch (Sketch):
        The original FreeCAD sketch object containing geometric elements and constraints.

    constraint (Constraint):
        The constraint to be evaluated.

    Returns:
    --------
    (is_valid, moves_geometries): (bool, bool)
        - `is_valid`: `True` if the constraint does not introduce conflicts or invalid states;
            `False` if the constraint is invalid.
        - `moves_geometries`: `True` if the constraint causes movement of one or more geometries;
            `False` if no significant movement occurs.


    Prints:
    -------
    Displays a summary of the effect of the constraint on the geometric entities of the Sketch.Geometry list.
    It also displays the is_valid and moves_geometries binary flags.

```

```python
    Usage:
    ------
    Use this function to test constraints before committing them to the sketch.
    This allows you to detect unintended movements or conflicts early in the design process.

    Example:
    --------
    >>> # Add a coincident constraint to align the start of geometry 1 with the end point of geometry 2.
    >>> coincident_constraint = Sketcher.Constraint('Coincident', *(1, SubRef.START.value, 2, SubRef.END.value))

    # You can analyse the effect the constraint would have on the sketch geometry
    >>> is_valid, moves_geometries = analyse_constraint(sketch, coincident_constraint) # the function automatically prints an analysis of the constraint.
    Analysis of Constraint[0] (without applying it to the sketch):
        Type: Coincident
        Elements:
            - First: 1
            - FirstPos: START
            - Second: 2
            - SecondPos: END
        Movement:
            Sketch.Geometry[5]:
                - START moved from: (-0.500, -0.407) to (-0.297, 0.407)
        Moves Geometries: True
        IsValid: True
    >>> is_valid
    True
    >>> moves_geometries
    True

    Note:
    -----
    This function does not modify the original sketch. It only provides a preview of the
    potential impact of the given constraint.
    """

class Sketch:
    """Represents a 2D sketch object in FreeCAD, used for creating and defining geometric shapes,
    constraints, and profiles that can later be referenced in 3D operations (e.g., extrusion, revolution).

    The Sketch class provides methods to add, modify, and constrain geometric elements such as
    lines, arcs, circles, and points. Sketches serve as essential building blocks in parametric
    modeling, allowing users to control the relationships between elements through constraints.

    Attributes:
    -----------
    Name (str):
        The name of the sketch object.
    Geometry (List[Union[LineSegment, ArcOfCircle, Circle, Point]]):
        A list of geometric elements in the sketch (e.g., lines, arcs, circles, points).
    Constraints (List[Sketcher.Constraint]):
        A list of constraints applied to the sketch elements (e.g., coincidence, equality, tangency).
    ConstraintCount (int):
        The total number of constraints applied to the sketch.
    GeometryCount (int):
        The total number of geometric elements present in the sketch.
    Placement (Placement):
        Defines the position and orientation of the sketch in 3D space. This attribute allows
        the sketch to be moved or rotated within the document, affecting how it will be aligned
        with other objects in FreeCAD.
    State (List[str]):
        A list representing the current status of the sketch. Possible values include:
        - 'Touched': The sketch has been modified since the last update.
        - 'Untouched': The sketch has not been modified since its last valid state.
        - 'Invalid': The sketch contains errors or unsatisfied constraints.

    Usage Example:
    --------------
    >>> import FreeCAD
    >>> from FreeCAD import Part, Sketcher
    >>> doc = FreeCAD.newDocument("ExampleDoc")
    >>> sketch = doc.addObject('Sketcher::SketchObject', 'sketch')

    Methods:
    --------
    addGeometry(self, geometry: Union[LineSegment, ArcOfCircle, Circle, Point]) -> int:
        This method is used to adds a geometric element to the sketch.

        Parameters:
        -----------
        geometry:
            a geometric element to be added on the sketch (e.g., lines, arcs, circles, points).

        Returns:
        --------
        index (int):
            The index of the added geometry on the sketch.Geometry list.

        Usage Example:
        --------------
        >>> line = Part.LineSegment( App.Vector(0.2, 0.3, 0),  App.Vector(0.3, 0.2, 0))
        >>> line_index = sketch.addGeometry(line)  # A line is added.
        >>> sketch.recompute()
        >>> line = sketch.Geometry[line_index]


    delGeometries(self, identifiers: List[int]) -> None:
        Deletes one or more geometries from the sketch, based on their indices on the sketch.Geometry list.

        Parameters:
        -----------
        identifiers (List[int]):
            A list of zero-based identifiers specifying which geometries to delete from the sketch.
```

```
        Usage Example:
        -------------
        >>> sketch.delGeometries([1])  # This will delete the geometry at index 1
        >>> sketch.recompute()


    addConstraint(self, constraint: Sketcher.Constraint) -> int:
        Adds a constraint to the sketch.
        Returns the index of the added constraint.

        Parameters:
        ----------
        constraint (Sketcher.Constraint):
            a geometric constraint to be added on the sketch.

        Returns:
        --------
        index (int):
            The index of the added constraint on the sketch.Constraints list.

        Usage Example:
        -------------
        >>> parallel_constraint = Sketcher.Constraint('Parallel', 4, 6)
        >>> sketch.addConstraint(parallel_constraint)
        >>> sketch.recompute()

    recompute(self) -> None:
        Forces a recompute of the sketch to apply and update any pending changes.
        This ensures that all modifications (such as added or deleted geometries and constraints)
        are reflected in the document.

        Parameters:
        ----------
        None

        Returns:
        --------
        None
     """

class LineSegment:
    """Represents a line defined by two endpoints in 3D space.
    This class is part of the FreeCAD Part module and should be instantiated using `Part.LineSegment`.

    Constructor:
    -----------
    __init__(self, start_vector: App.Vector, end_vector: App.Vector)
        Initializes a LineSegment with specified start and end points in 3D space.

        Parameters:
        ----------
        start_vector (App.Vector):
            A 3D vector representing the coordinates of the line's start point.
        end_vector (App.Vector):
            A 3D vector representing the coordinates of the line's end point.

    Attributes:
    ----------
    StartPoint (App.Vector):
        The start point of the line segment.
    EndPoint (App.Vector):
        The end point of the line segment.

    Usage Example:
    -------------
    >>> # Create a LineSegment from start and end points.
    >>> start_point = App.Vector(0, 0, 0)
    >>> end_point = App.Vector(1, 1, 1)
    >>> line = Part.LineSegment(start_point, end_point)
    >>> line.StartPoint.x
    0 # access the x coordinate of the start point
   """

class Circle:
    """Represents a circle in 3D space defined by a center point, a normal vector and a radius using FreeCAD's Vector objects.
    This class is part of the FreeCAD Part module and should be instantiated using `Part.Circle`.

    Constructor:
    -----------
    __init__(self, center_vector: App.Vector, normal_vector: App.Vector, radius: float):
        Initializes a Circle with a specified center, normal vector, and radius.

        Parameters:
        ----------
        center_vector (App.Vector):
            A 3D vector with the coordinates of the center point of the circle.
        normal_vector (App.Vector):
            A 3D vector representing the direction normal to the circle's plane.
        radius (float):
            The radius of the circle.

    Attributes:
    ----------
    Center : (App.Vector)
        The center point of the circle.
    Radius : (float)
        The radius of the circle.

    Usage Example:
    -------------
    >>> center = App.Vector(6.0, 3.0, 0)
    >>> normal = App.Vector(0, 0, 1)
```

```python
    >>> radius = 1.1
    >>> circle = Part.Circle(center, normal, radius)
    >>> circle.Center
    Vector (6.0, 3.0, 0.0)
    """

class Point:
    """Represents a point in 3D space.
    This class is part of the FreeCAD Part module and should be instantiated using `Part.Point`.

    Constructor:
    ------------
    __init__(self, point_vector: App.Vector):
        Constructor of the Point class

        Parameters:
        ----------
        point_vector (App.Vector):
            A 3D vector with the coordinates of the point.

    Attributes:
    -----------
    X (float):
        The x-coordinate of the point.
    Y (float):
        The y-coordinate of the point.
    Z (float):
        The z-coordinate of the point.

    Usage Example:
    --------------
    >>> point = Part.Point(App.Vector(1.0, 2.0, 3.0))
    >>> point.X
    1.0
    """

class ArcOfCircle:
    """Represents a circular arc derived from a given circle, defined by start and end angles in radians.
    The arc is drawn counterclockwise from the start angle to the end angle. Angles are expressed in radians
    where 0 radians correspond to the positive x-axis and increase counterclockwise.

    This class is part of the FreeCAD Part module and should be instantiated using `Part.ArcOfCircle`.

    Constructor:
    ------------
    __init__(self, circle: Circle, start_param: float, end_param: float)
        Initializes an ArcOfCircle instance from a circle and specified start and end parameters.

        Parameters:
        ----------
        circle (Circle):
            The Circle object from which the arc is derived.
        start_param (float):
            The starting parameter (angle in radians) on the circle's circumference that defines the beginning of the arc.
        end_param (float):
            The ending parameter (angle in radians) on the circle's circumference that defines the end of the arc.

    Attributes:
    -----------
        Radius : (float)
            The radius of the circle from which the arc is derived.
        StartPoint : (App.Vector)
            The start point of the arc.
        EndPoint : (App.Vector)
            The end point of the arc.
        Center : (App.Vector)
            The center point of the circle from which the arc is derived.
        FirstParameter : (float)
            The start angle of the arc in radians.
        LastParameter : (float)
            The end angle of the arc in radians.

    Usage Example:
    --------------
    >>> #Create counterclockwise ArcOfCircle with center, radius, and start and end angles in radians.
    >>> arc_center = App.Vector(0.0670, -0.0000, 0.0)  # Center of the arc
    >>> arc_radius = 0.0130  # Radius of the arc
    >>> start_param = -1.6008  # Start parameter in radians
    >>> end_param = -0.0000  # End parameter in radians
    >>> arc_direction = App.Vector(0, 0, 1)
    >>> # Create the arc using Part.ArcOfCircle
    >>> arc = Part.ArcOfCircle(Part.Circle(arc_center, arc_direction, arc_radius), start_param, end_param)
    """

class Arc:
    """Represents an arc defined by a start point, an end point, and an intermediate point on the arc.

    This class is part of the FreeCAD `Part` module and should be instantiated using `Part.Arc`.
    The arc is uniquely determined by three points: the start, the end, and a point somewhere
    on the arc (referred to as the midpoint, though it need not be the geometric middle). The
    arc lies on the circle that passes through these three points.

    After calling `recompute()` on a FreeCAD sketch, an `Arc` object is automatically
    transformed into an `ArcOfCircle` object. This is because FreeCAD optimizes the
    geometry representation for arcs, converting them to arcs of circles after
    the geometry is fully processed.

    Constructor:
    ------------
    __init__(self, start_vector: App.Vector, end_vector: App.Vector, mid_vector: App.Vector)
        Initializes a Arc with specified start, end and mid points in 3D space.

```

```
        Parameters:
        ----------
        start_vector (App.Vector):
            A 3D vector representing the coordinates of the arc's start point.
        end_vector (App.Vector):
            A 3D vector representing the coordinates of the arc's end point.
        mid_vector (App.Vector):
            A 3D vector representing a point on the curcomference of the arc.

    Usage:
    ------
    Use this function to create ArcOfCircle objects from start, end and mid points.

    Usage Example:
    --------------
    >>> start_point = App.Vector(5.0, 0, 0)
    >>> end_point = App.Vector(0, 5.0, 0)
    >>> mid_point = App.Vector(3.54, 3.54, 0)
    >>> arc = Part.Arc(start_point, mid_point, end_point)
    <Arc object>
    >>> sketch.addGeometry(arc)
    >>> sketch.recompute()
    >>> arc
    <ArcOfCircle object>
    """


class SubRef(Enum):
    START = 1
    END = 2
    CENTER = 3


class Constraint:
    """Represents a geometric constraint in a FreeCAD sketch.
    Constraints define relationships between geometric elements (lines, arcs, circles, points), ensuring specific properties or behaviors.
    Constraints can be created using the `Sketch.addConstraint()` method.

    Constructor:
    ------------
    __init__(self, constraint_type: str, *args)
        Initializes a Sketcher.Constraint instance with a specified type and parameters.

        Parameters:
        -----------
        constraint_type (str):
            The type of constraint to apply. Supported types include:
            - 'Coincident'
            - 'Parallel'
            - 'Equal'
            - 'Vertical'
            - 'Horizontal'
            - 'Perpendicular'
            - 'Tangent'

        *args (varies):
            Additional parameters specific to the constraint type. These define the geometries or points
            to which the constraint applies and any additional constraint-specific requirements.

        Usage:
        ------
        Supported Constraint Types and Their Arguments:
        1. Coincident: Enforces that two points or vertices coincide (i.e., share the same location in space).
            - args: ('Coincident', First, FirstPos, Second, SecondPos)
                - First (int):
                    The index of the first geometry.
                - FirstPos (int):
                    The vertex (1 for start, 2 for end, 3 for center) of `Geometry_index1` to fulfill the constraint.
                - Second (int):
                    The index of the second geometry.
                - SecondPos (int):
                    The vertex (1 for start, 2 for end, 3 for center) of `Geometry_index2` to fulfill the constraint.

        2. Parallel: Ensures two lines remain parallel.
            - args: ('Parallel', First, Second)
                - First (int):
                    The index of the first Line segment to be made parallel.
                - Second (int):
                    The index of the second Line segment to be made parallel.

        3. Equal: Makes two lines or circles equal in length or radius.
            - args: ('Equal', First, Second)
                - First (int):
                    The index of the first line segment or circle.
                - Second (int):
                    The index of the first line segment or circle.

        4. Vertical: Forces a line segment to be vertical.
            - args: ('Vertical', First)
                - First (int): The index of the Line segment.

        5. Horizontal: Forces a line segment to be horizontal.
            - args: ('Horizontal', First)
                - First : (int)
                    The index of the line segment.

        6. Perpendicular: Ensures that two line segments are perpendicular.
            - `args`: ('Perpendicular', Geometry_index1, Geometry_index2)
                - Geometry_index1 : (int)
                    The index of the first line segment.
                - Geometry_index2 : (int)
                    The index of the second line segment.

```

```
        7. Tangent: Makes a line tangent to a curve.
            - args: ('Tangent', Geometry_index1, Geometry_index2)
                - Geometry_index1 : (int)
                    The index of the first geometry.
                - Geometry_index2 : (int)
                    The index of the first geometry.

        Usage Example:
        --------------
        >>> coincident_constraint = Sketcher.Constraint('Coincident', *(1, SubRef.START.value, 2, SubRef.END.value))
        >>> sketch.addConstraint(coincident_constraint)
        >>> sketch.recompute()
        >>> sketch.State
        ['Touched']
        # Check the total number of constraints applied to the sketch.
        >>> print(f"Number of constraints: {len(sketch.Constraints)}")
        Number of constraints: 2
        # Remove the most recently added constraint.
        >>> sketch.delConstraint(len(sketch.Constraints) - 1)
        # Remove the constraint on specific index.
        >>> sketch.delConstraint(coincident_constraint_index)
        >>> sketch.addConstraint(Sketcher.Constraint('Horizontal', 0))
        >>> sketch.recompute()
        >>> sketch.State # Use the State Variable to ensure that all added constraints are valid.
        ['Touched', 'Invalid']
    """

class Extrusion:
    """Represents an extrusion of a sketch in FreeCAD.
    This class is part of the FreeCAD Part module and should be instantiated using a sketch object and the desired extrusion parameters.

    Attributes:
    -----------
    Base (Sketch):
        The sketch object that is extruded into a 3D solid.
    DirMode (str):
        Direction mode of the extrusion (default: "Normal").
    LengthFwd (float):
        Forward extrusion length.
    LengthRev (float):
        Reverse extrusion length.
    Solid (bool):
        Whether the extrusion is a solid (default: True).
    Reversed (bool):
        Whether the extrusion direction is reversed.
    Symmetric (bool):
        Whether the extrusion is symmetric along the sketch plane.
    TaperAngle (float):
        Taper angle for the extrusion.
    TaperAngleRev (float):
        Reverse taper angle for the extrusion.

    Usage Example:
    --------------
    >>> extrude =  doc.addObject('Part::Extrusion', 'Extrude')
    >>> extrude.Base = sketch # extrude an existing sketch object
    >>> extrude.DirMode = "Normal"
    >>> extrude.DirLink = None
    >>> extrude.LengthFwd = 10.00
    >>> extrude.LengthRev = 0.0
    >>> extrude.Solid = True
    >>> extrude.Reversed = False
    >>> extrude.Symmetric = False
    >>> extrude.TaperAngle = 0.0
    >>> extrude.TaperAngleRev = 0.0
    >>> doc.recompute()
   """

class Solid:
    """Represents a 3D solid in FreeCAD, created as part of an `Extrusion` operation.
    This shape object holds the geometry of the extruded solid and provides access to various
    geometric properties, as well as methods for performing transformations and boolean operations
    with other shapes.

    Attributes:
    -----------
    TypeId (str):
        The type of shape, typically "Part::TopoShape".
    Volume (float):
        The volume of the extruded solid.
    Area (float):
        The total surface area of the extruded solid.
    BoundBox (BoundBox):
        The bounding box of the shape, describing the spatial limits of the extrusion.

    Methods:
    --------
    fuse(shape: Shape) -> Shape:
        Performs a union operation, merging this shape with another solid to create a combined shape.

        Parameters:
        -----------
        shape (Shape):
            The other solid shape with which to perform the union operation.

        Returns:
        --------
        Shape:
            A new solid object representing the union of this shape and the specified shape.

        Usage Example:
        --------------
        >>> shape1 = extrude1.Shape  # Access the extruded shape
```

```
        >>> shape1 = extrude1.Shape  # Access the extruded shape
        >>> shape2 = extrude2.Shape  # Access the extruded shape
        >>> result_shape = shape2.fuse(shape1)  # Union with another shape

    cut(shape: Shape) -> Shape:
        Performs a cut operation, subtracting the specified shape from this shape.

        Parameters:
        ----------
        shape (Shape):
            The solid shape to subtract from this shape.

        Returns:
        --------
        Shape:
            A new solid object representing the result of subtracting the specified shape from this shape.

        Usage Example:
        -------------
        >>> result_shape = shape1.cut(shape2)  # Subtract shape2 from shape1

    common(shape: Shape) -> Shape:
        Performs an intersection operation, keeping only the volume that is common between this shape and another.

        Parameters:
        ----------
        shape (Shape):
            The solid shape to intersect with this shape.

        Returns:
        --------
        Shape:
            A new solid object representing the intersected volume of the two shapes.

        Usage Example:
        -------------
        >>> result_shape = shape1.common(shape2)  # Intersect shape1 with shape2
    """
```