AnalogCoder: Analog Circuit Design via Training-Free Code Generation

Yao Lai[1], **Sungyoung Lee**[2], **Guojin Chen**[3], **Souradip Poddar**[2],
Mengkang Hu[1], **David Z. Pan**[2], **Ping Luo**[1]
[1]The University of Hong Kong, [2]The University of Texas at Austin,
[3]The Chinese University of Hong Kong

Abstract

Analog circuit design is a significant task in modern chip technology, focusing on the selection of component types, connectivity, and parameters to ensure proper circuit functionality. Despite advances made by Large Language Models (LLMs) in digital circuit design, the complexity and scarcity of data in analog circuitry pose significant challenges. To mitigate these issues, we introduce AnalogCoder, the first training-free LLM agent for designing analog circuits through Python code generation. Firstly, AnalogCoder incorporates a feedback-enhanced flow with tailored domain-specific prompts, enabling the automated and self-correcting design of analog circuits with a high success rate. Secondly, it proposes a circuit tool library to archive successful designs as reusable modular sub-circuits, simplifying composite circuit creation. Thirdly, extensive experiments on a benchmark designed to cover a wide range of analog circuit tasks show that AnalogCoder outperforms other LLM-based methods. It has successfully designed 20 circuits, 5 more than standard GPT-4o. We believe AnalogCoder can significantly improve the labor-intensive chip design process, enabling non-experts to design analog circuits efficiently. Codes and the benchmark are provided at github.com/laiyao1/AnalogCoder.

1 Introduction

Analog circuits, essential for processing real-world signals such as temperature, pressure, sound, and light, are indispensable in modern integrated circuits. They facilitate accurate sensing, amplification, and filtering, crucial for linking digital systems with physical environments. This functionality underpins reliable data acquisition and signal processing across diverse applications, including wireless communications [1], video sensing [2], and digital medical devices [3].

The success of Large Language Models (LLMs) [4, 5] has brought new opportunities for automatic chip design [6]. Existing related research primarily focuses on two tasks: the generation and correction of Verilog codes [7–16], and the writing of design scripts [17, 18]. LLMs can convert natural language descriptions of digital circuit design tasks into Verilog code, a programming language for designing digital circuits. Once the code is generated, it can be assessed for correctness by LLMs or human experts, who attempt to fix errors by analyzing error information and simulation outputs [7, 10, 14, 19]. Due to the scant representation of Verilog in the public data for training [20], LLMs may not perform as well in generating Verilog code as they do with widely-used programming languages such as C and Python, despite ongoing improvement efforts [16]. Similarly, generating design flow scripts is another form of code generation, converting natural language descriptions of design requirements

---

This work is done while the first author is a visiting student at UT Austin.

Corresponding to: Ping Luo (pluo@cs.hku.hk), David Z. Pan (dpan@ece.utexas.edu).
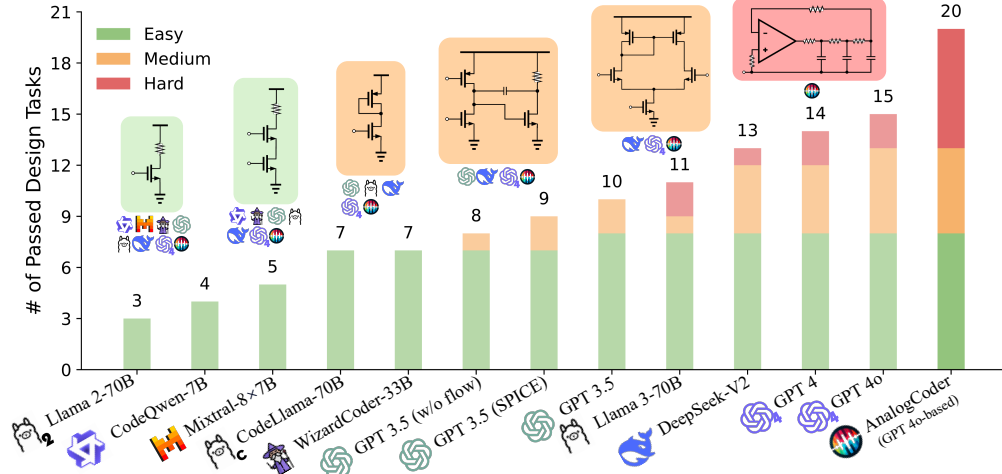
Figure 1: **Leaderboard of LLM analog circuit design.** LLMs are ranked by the number of analog circuits they design successfully. Design tasks are classified as easy, medium, or hard based on component count and connection complexity. It displays several designs from our benchmark (Task ID=1, 5, 10, 9, 11, 16) and lists the LLMs that successfully created them. Results are from Table 3 and 4.

into script files. These scripts, written in Python or Tcl, facilitate the chip design process by invoking APIs at various stages [21, 17, 18]. These design flow scripts typically implement straightforward logic to transform fundamental workflows into a series of API calls, akin to control systems used in robotics [22]. However, these works are mainly for digital circuit design, as listed in Table 1.

Analog circuit design presents significantly more challenges than digital circuit design [23–25], leaving the field less explored by LLM-aided methods. The primary challenges include: (1) *Complexity.* Unlike digital circuit design, which predominantly employs simple logic gates, analog circuits comprise diverse components such as voltage and current sources, MOSFETs, resistors, and capacitors. The complexity is further compounded by the intricate interconnections and settings required. Even minor adjustments can significantly alter the circuit's functionality, potentially leading to a combinatorial explosion due to the vast search space. (2) *Abstraction level.* Digital circuit design languages like Verilog [26] allow developers to write at a high level of abstraction, such as assigning functionality directly, without needing to specify the underlying hardware components like logic gates. In contrast, analog circuit design requires a direct representation of the physical components in the design code. It necessitates a more detailed and component-specific design process, making it more difficult to utilize LLM assistance effectively. For example, while a digital adder can be succinctly implemented in a single line of Verilog code, constructing an analog adder requires meticulous configuration and connection of approximately five MOSFETs and three resistors [27]. (3) *Corpus data volume.* Although Verilog, used for digital circuit design, constitutes a small fraction (less than 0.1%) of the repositories on GitHub, SPICE (Simulation Program with Integrated Circuit Emphasis) [28], the predominant language for analog design, is even less common. This scarcity suggests that LLMs may find it more challenging to learn the design rules for analog circuits compared to digital ones. Thus, analog circuit design is a time-intensive, challenging, and error-prone process that predominantly depends on the meticulous contributions of experienced engineers, typically necessitating several days of dedicated expert effort to meet specific functional requirements [29, 25].

To mitigate the shortcomings of traditional manual analog circuit design and to bridge the existing gap in LLM applications for such tasks, we introduce AnalogCoder, a novel training-free LLM-based agent that enables analog circuit design through the generation of Python code. Specifically, users can describe their desired analog circuit functionalities in natural language, and AnalogCoder automatically generates the corresponding Python code for the designed circuit, leveraging the LLM's strong Python programming capabilities. To further enhance the design capabilities of LLMs, we propose domain-specific prompt engineering, feedback-enhanced design flow, and the circuit tool library, greatly increasing the success rate of design.

In this work, we prioritize the correct functionality of analog circuits, avoiding extensive parameter optimization, already well-addressed by existing advanced methodologies [30–32]. Extensive experiments demonstrate that AnalogCoder can autonomously solve 20 out of 24 analog circuit challenges,

Table 1: **Comparison of works.** AnalogCoder is the first LLM-based work on analog circuit design. At the same time, AnalogCoder operates without human feedback and features automatic error correction. A comprehensive dataset is developed and provided to evaluate analog circuit design capabilities.

Method	Fully Automated [1]	Auto Fix Errors [2]	Benchmark	Open-Source	Training-Free	Circuit Type
ChipChat [7]	✗	✗	✓	✓	✓	Digital
ChipGPT [8]	✗	✗	✓	✗	✓	Digital
VeriGen [9]	✓	✗	✓	✓	✗	Digital
AutoChip [10]	✓	✓	✗	✓	✓	Digital
VerilogEval [12]	✓	✗	✓	✗	✗	Digital
RTLLM [13]	✓	✗	✓	✓	✓	Digital
RTLfixer [14]	✓	✓	✗	✓	✓	Digital
RTLCoder [15]	✓	✗	✗	✓	✗	Digital
ChipNeMo [18]	✓	✗	✗	✗	✗	Digital[3]
BetterV [16]	✓	✗	✗	✗	✗	Digital
AnalogCoder	✓	✓	✓	✓	✓	Analog

[1] Without human involvement.   [2] Automatic error fix by LLMs.   [3] Analog circuit only for QA questions.

as shown in Fig. 1, which surpasses the performance of the standard GPT-4o (15 solved) and the Llama-3 (11 solved).

This paper makes three main **contributions**: First, we introduce AnalogCoder, which, to the best of our knowledge, is the *first* LLM-based agent for analog integrated circuit design. This agent establishes a new paradigm by generating Python code to design analog circuits. Second, we develop a feedback-enhanced design flow and a circuit tool library, significantly improving the LLM's ability to design functional analog circuits. Third, we introduce the *first* benchmark specifically designed to evaluate the ability of LLMs in designing analog circuits. This benchmark comprises 24 unique circuits, three times the number included in the ChipChat benchmark [8] and offers 40% more circuits than the VeriGen benchmark [9]. It features detailed task descriptions, sample designs, and test-benches, enhancing resources for future research.

2   Preliminary

Analog Circuits.   Unlike digital circuits, which exclusively process discrete binary signals, analog circuits manage continuous-valued signals, thereby enabling a diverse array of functionalities [33]. For example, an analog amplifier, as depicted in Fig. 2, is engineered to enhance the amplitude of an input signal, expressed as $V_{out}(t) = A_v \times V_{in}(t)$, where $V_{in}(t)$ and $V_{out}(t)$ denote the time-variant behavior of the input and output voltage signals, respectively, and A_v represents the voltage gain of the amplifier. Moreover, operational amplifiers (op-amps) are high-gain voltage amplifiers with differential inputs, featuring a non-inverting input V_{inp} and an inverting input V_{inn}. The output of the op-map is expressed as $V_{out}(t) = A_v \times [V_{inp}(t) - V_{inn}(t)]$, enabling it to be configured to perform a variety of analog signal operations, including integration, differentiation, addition, and subtraction. When configured as an adder, for instance, the operational amplifier can implement the function $V_{out}(t) = -[V_{in1}(t) + V_{in2}(t)]$. Moreover, when set up as an integrator, it can integrate the input voltage signal, yielding an output given by $V_{out}(t) = -\int V_{in}(t) \, dt/\tau$, where τ represents the time constant associated with the resistance and capacitance values within the circuit. These analog circuits demonstrate how analog operations transform input signals into output signals, accomplishing computations more efficiently than clock-dependent digital circuits. Testing an analog circuit involves applying a specific input and verifying that the output aligns with expected standards to ensure its correct operation. After simulation, attributes such as gain, common-mode gain, and phase difference are identified as specifications. These specifications are essential for validating the circuit's performance against its intended design requirements.

Code Representation for Circuits.   To facilitate the description and simulation of analog circuit designs, SPICE [28] has been introduced. Often used as a programming language, this tool allows designers to specify the complex interconnections between electronic components within a circuit. With SPICE codes, the behavior of circuits can be accurately simulated and analyzed, with each component, such as resistors, capacitors, and voltage or current sources, carefully itemized and connected in a notation recognized industry-wide. In the SPICE syntax, the fundamental constructs are elements and nodes (see Fig. 2). The elements refer to various electronic components like resistors and transistors, while nodes denote the points at which these elements are interconnected. As shown in the circuit diagram in Fig. 2, the amplifier comprises four elements: two voltage sources, V_{dd} and
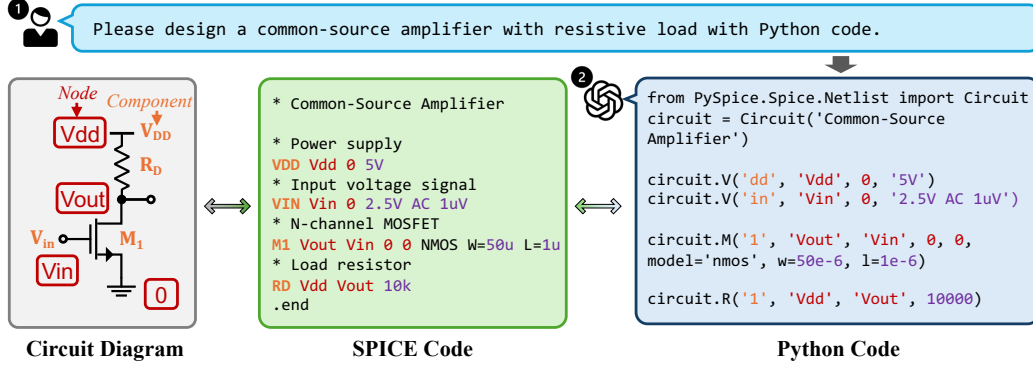
Figure 2: **Representations of designed circuit.** We input the design task description into the LLM, which outputs the design in Python. Three interconvertible representations of the designed circuit: (1) Circuit diagrams, typically take human experts several days to design due to the intricate and demanding process of selecting and connecting components. (2) SPICE code, which represents circuits through formatted netlists. (3) Python code with the PySpice library, achieving circuits equivalent to those generated by SPICE code. However, due to the coding's complexity and non-intuitive nature, experts typically sketch circuit diagrams manually and then generate the codes using design tools. More code samples are in Appendix Sec. A.4.

V_{in}, for the power supply and signal input, respectively; one N-channel MOSFET, M_1, for signal amplification; and one resistive load, R_D. Four lines in the SPICE code describe these elements. Each line in the SPICE code starts with the element name, followed by the names of the nodes to which the element is connected. For instance, the resistor R_D is connected between nodes V_{dd} and V_{out}. The corresponding SPICE code line is 'RD Vdd Vout 10k', where '10k' denotes $10\,\mathrm{k}\Omega$. Specifically, since a MOSFET has four connection nodes, the corresponding code line will include four node labels delineating the drain, gate, source, and bulk connections. PySpice [34] integrates SPICE code with the Python programming language, leveraging Python's user-friendly syntax and robust ecosystem to simplify circuit simulation and data processing, as demonstrated in the Python code in Fig. 2. This integration allows for more accessible and efficient design workflows, broadening the usability of SPICE. Since LLMs excel at Python programming [35, 36], we chose Python with the PySpice library to automate the creation of circuits, replacing the tedious manual process.

3 Our Approach

Method Overview. AnalogCoder is an LLM-based agent that interprets task descriptions in natural language to automatically generate Python code, representing functionally correct analog circuits. To enhance the design capabilities of the agent, we implemented a comprehensive methodology as shown in Fig. 3, including prompt engineering, a feedback-enhanced design flow, and a circuit tool library. Prompt engineering enhances the agent's design thinking through strategic, problem-solving prompts. The feedback-enhanced design flow uses multiple checks to provide error feedback to the agent, facilitating the correction of failed designs by LLMs. The circuit tool library, a modular sub-circuit repository, systematically organizes designed circuits as tools, enabling straightforward retrieval and reuse by LLMs for complex circuit designs.

Prompt Engineering. We initially established a well-crafted design prompt to maximize the design capabilities of LLMs. Our approach to prompt engineering encompasses three main aspects: (1) *programming language selection*, (2) *in-context learning* [37], and (3) *Chain-of-Thought* [38]. Despite the capability of LLMs to generate code in multiple programming languages, their performance in Python surpasses that in most others [39, 36]. Additionally, many prominent code-generating LLMs, such as CodeLlama [40] and WizardCoder [41], are primarily fine-tuned on Python datasets, indicating a bias towards Python. Conversely, the training datasets for most LLMs, which are based on GitHub, do not contain sufficient data on SPICE code [20]. Therefore, to mitigate this limitation, we directly prompt the LLM to generate executable Python code compatible with the PySpice library. Furthermore, we integrate in-context learning [37] to enhance circuit design, providing a detailed example of a two-stage amplifier with active and resistor loads as one-shot learning [42]. This example facilitates the LLM's learning and imitation and standardizes its output, minimizing errors. All design tasks are distinct from the provided example to maintain evaluation fairness. Additionally,
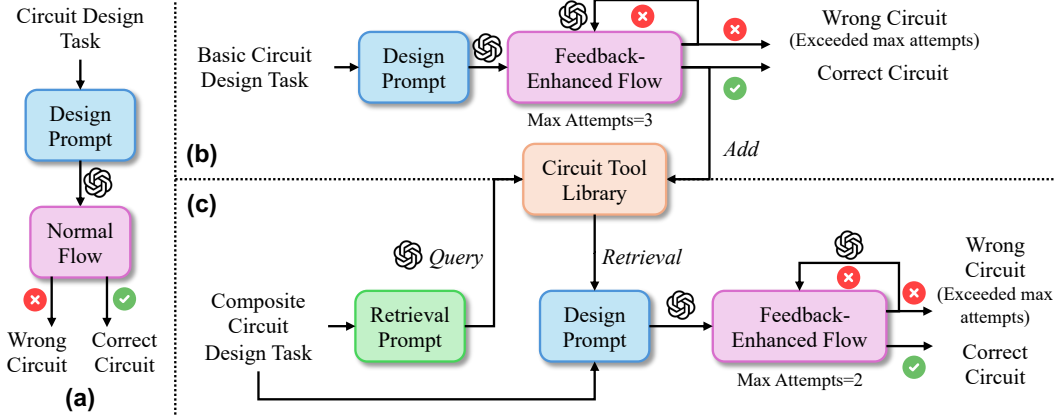
Figure 3: **Method Overview. (a) Standard method.** By converting circuit design tasks into prompts and inputting the LLM's outputs into the normal flow, the correctness of the circuit is verified. **(b) Our method for basic circuit design.** Input the design prompts to the feedback-enhanced design flow, enabling the automated error fix with LLMs. Successfully designed circuits are added to the circuit tool library, while failed designs are returned to the LLM for automatic fixing. **(c) Our method for composite circuit design.** The process adds a step of querying the library to retrieve invocation methods for subcircuits, which are then integrated into the design prompt to facilitate the design of composite circuits.

the Chain-of-Thought strategy [38] involves prompting the LLM to generate a detailed design plan, including necessary components and their interconnections. This plan subsequently guides the generation of the corresponding design code, simplifying the design task significantly. The complete prompt template can be seen in Appendix Sec. A.2.

Feedback-Enhanced Design Flow. Various errors are often observed in the code generated by LLMs. Consequently, guiding LLMs to correct the generated codes based on error messages is crucial. Numerous studies [43–45, 14, 9] have suggested that providing LLMs with relevant error information helps LLMs fix faulty code. However, for the analog circuit design, besides the runtime errors that may occur when executing SPICE simulations, additional verification of circuit-related information is necessary to ensure the correctness of the design. In analog circuit design, when a design fails, we return either runtime errors from the Python code or circuit-specific test errors to the LLM, as illustrated in Fig. 4. We divide the feedback-enhanced flow into four stages: (1) *requirement check*, (2) *simulation and operating point check*, (3) *DC sweep check*, and (4) *function check*. The *requirement check* is to verify whether the generated code meets the basic design requirements, such as the presence of requisite inputs and outputs, and the inclusion of essential circuit components. The *simulation and operating point check* initially assesses whether the generated analog circuit can successfully execute simulations, aiming to identify issues such as floating nodes and other potential errors. Once the simulation passes, the static operating point voltages of nodes are achieved. Examining these operating point voltages ensures that the MOSFET transistors are in their correct operational states. The *DC sweep check* performs a direct current (DC) analysis by changing the voltage at the input nodes and observing the corresponding changes at the output nodes to verify the integrity of the signal path from input to output. This method also helps identify the optimal bias voltage, increasing the success rate of the design. The *function check* simulates specific input waveforms and observes the outputs to verify the analog circuit's fundamental functionalities as Appendix Table 6. The simulation may involve DC, AC (alternating current), or transient analyses depending on the circuit types. For any errors occurring in these checks, the relevant error information is returned to the LLM, which then regenerates a circuit design. Due to limitations in the LLMs' code repair capabilities, we allow up to three code generations, which means up to two retries.

Circuit Tool Library. As analog circuit design tasks become more complex and the implementation code grows more intricate, it becomes increasingly challenging for LLMs to generate correct circuits. To address this complexity, basic circuits can be encapsulated into subcircuit modules in the SPICE code, facilitating their integration into more composite assemblies. Building on this modular approach and inspired by the tool-based LLM studies [46, 47], we adopted a circuit tool library that stores correctly designed subcircuits for easy reuse in more complex designs. As illustrated in Fig. 5, our approach involves two main processes: adding circuits to the library (top) and retrieving circuits
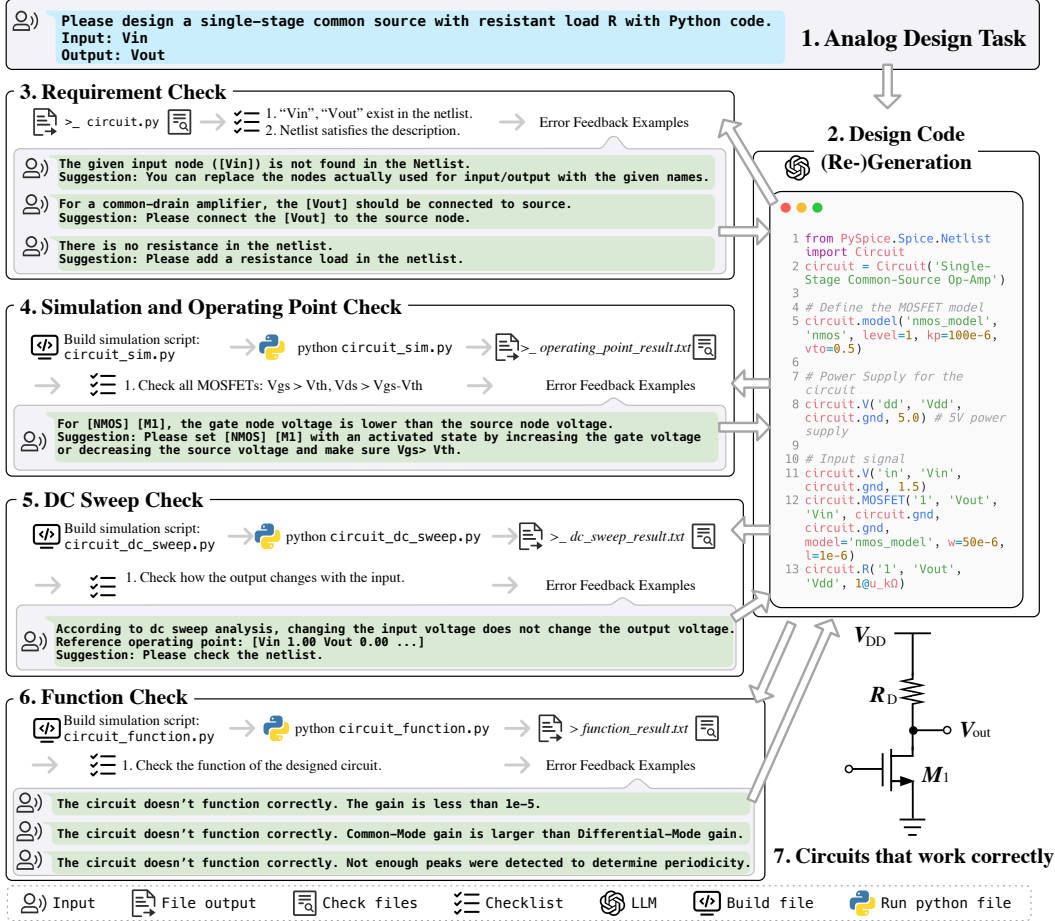
Figure 4: **Feedback-Enhanced Design Flow.** The flow facilitates autonomous error correction in designs by the LLM agent without human intervention. Error messages identified during the checking process are returned to the LLM to assist in refining the design. The entire flow is adaptable to nearly all categories of analog circuits.

from the library (bottom). After an LLM-based agent completes a basic circuit design task, we add the circuit codes and the specifications from the simulation results to the circuit tool library. If a circuit task has been successfully completed multiple times, store the optimal circuit design based on the key specification, such as gain. The task descriptions and circuit information are stored as keys for queries, while the codes and calling methods are stored as values. In composite circuit design, the task description is used to formulate a query prompt, enabling the retrieval of the requisite subcircuit tools by LLMs. The agent initially retrieves the indices of the required subcircuits and then uses these indices to fetch all corresponding specifications and calling methods. This information is then integrated with the task description and automatically re-entered into the LLM to design the circuit. At this stage, the agent uses the retrieved subcircuits' calling methods to directly integrate them into the code, thereby designing composite circuits. As shown in Fig. 5, when designing an op-amp integrator, the LLM queries and retrieves the index corresponding to the required subcircuit, a single-stage op-amp. Subsequently, the task description, along with the pertinent information of this subcircuit, is input into the LLM, which then generates the design code for the op-amp integrator.

Fine-tuning. Due to the scarcity of datasets for analog circuits and inspired by GPT-assisted data generation [48], we collected samples of successful circuit designs created by GPT-3.5, GPT-4o, and Llama-3 to fine-tune GPT-3.5 by the provided API. We gathered successful designs for each task and clustered them into three categories using text vectorization TF-IDF [49]. One design from each category was selected and paired with the input prompt to form initial pairs, then refined through text filtering to create the fine-tuning data. Cross-validation techniques [50] are applied to ensure that tasks used in fine-tuning were excluded from the testing set. Further details are provided in Appendix Sec. A.6.
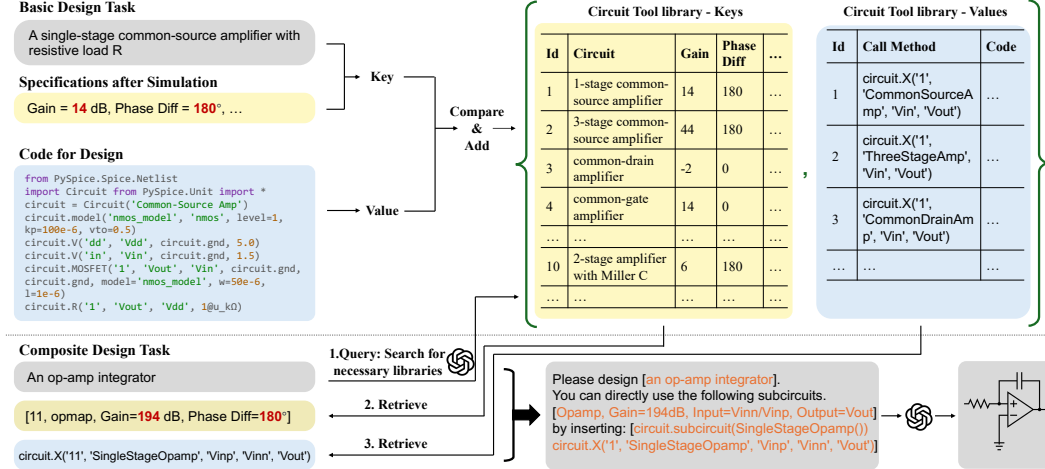
Figure 5: **Circuit Tool library. Top:** Addition of new tools derived from successfully designed basic circuits. Here, descriptions and specifications are keys, while design codes are stored as values. **Bottom:** Retrieval of tools from the library for designing composite circuits. The process begins with the LLM querying necessary tools using the task description. Subsequently, the keys and values of the retrieved tools, with the task description, are employed as prompts for circuit design.

4 Experiments

We extensively evaluate the capability of LLMs in analog circuit design, including Mixtral-7×8B [51], CodeLlama-70B-Instruct [40], Wizardcoder-33B-V1.1 [41], Llama3-70B [52], DeepSeek-V2 [53], GPT-3.5-turbo [42], GPT-4-turbo [5] and GPT-4o. CodeLlama and WizardCoder are code generation LLMs, fine-tuned on Llama2 [54] and StarCoder [55], respectively. Llama-3 and DeepSeek-V2 are the newest open-source general LLMs. WizardCoder, DeepSeek-V2, and Llama-3 are LLMs that outperformed GPT-3.5 on the HumanEval [56] coding tasks [57]. For additional models, see Appendix Sec. A.7. Open-source models were evaluated on 4 Nvidia A100 GPUs.

Metrics. We use 'Pass@k' [58] (k=1, 5), a metric widely used in code generation tasks [40, 59, 41, 57, 20], as the main evaluation metric. It is defined as the ratio of correct generations within k independent trials, with higher values being better. We conduct n trials ($n \geq k$), and compute $Pass@k = 1 - \binom{n-c}{k}/\binom{n}{k}$, where c is the number of successful trials. For open-source LLMs and GPT-3.5, we set $n = 30$; for fine-tuned GPT-3.5, GPT-4, and GPT-4o, $n = 15$. 'Number of solved' refers to the count of distinct tasks for which a circuit design is successfully achieved at least once in n trials.

Benchmark. We have developed a comprehensive benchmark of analog circuit design tasks, detailed in Table 2, to fill the gap in open-source benchmarks for this field. The difficulty of these tasks is determined by the number of components and the complexity of their connections. Tasks 1-15 are basic circuits, while 16-24 are composite circuits. Details are available in Appendix Sec. A.1.

Table 2: **Benchmark Descriptions.** All analog circuit design tasks are listed with their corresponding types. Different difficulties are distinguished by background colors (easy, medium, and hard).

Id	Type	Circuit Description	Id	Type	Circuit Description
1	Amplifier	Common-source amp. with R load	13	Opamp	Common-source op-amp with R loads
2	Amplifier	3-stage common-source amplifier with R loads	14	Opamp	2-stage op-amp with active loads
3	Amplifier	Common-drain amp. with R load	15	Opamp	Cascode op-amp with cascode loads
4	Amplifier	Common-gate amp. with R load	16	Oscillator	Wien Bridge oscillator
5	Amplifier	Cascode amp. with R load	17	Oscillator	RC Shift oscillator
6	Inverter	NMOS inverter with R load	18	Integrator	Op-amp integrator
7	Inverter	Logical inverter with NMOS and PMOS	19	Differentiator	Op-amp differentiator
8	Current Mirror	NMOS constant current source with R load	20	Adder	Op-amp adder
9	Amplifier	Common-source amp. with diode-connected load	21	Subtractor	Op-amp subtractor
10	Amplifier	2-stage amplifier with Miller compensation C	22	Schmitt trigger	Non-inverting Schmitt trigger
11	Opamp	Op-amp with active current mirror loads	23	VCO	Voltage-Controlled Oscillator
12	Current Mirror	Cascode current mirror	24	PLL	Phase-Locked Loop

Table 3: **Main results.** All LLMs, except GPT-4o, have been enhanced by prompt engineering, design flow feedback, and the circuit tool library. To highlight the impact of the circuit tool library, GPT-4o was evaluated without it. AnalogCoder can be seen as the GPT-4o enhanced with the circuit tool library.

Model Task ID	CodeLlama-70B		WizardCoder-33B		DeepSeek-V2		Llama3-70B		GPT3.5		GPT4o (w/o tool)		AnalogCoder	
	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5
1	20.0	70.2	93.3	100.0	100.0	100.0	93.3	100.0	86.7	100.0	100.0	100.0	100.0	100.0
2	3.3	16.7	13.3	53.8	93.3	100.0	20.0	70.2	70.0	99.9	100.0	100.0	100.0	100.0
3	0.0	0.0	0.0	0.0	83.3	100.0	90.0	100.0	3.3	16.7	100.0	100.0	100.0	100.0
4	3.3	16.7	10.0	43.3	70.0	99.9	83.3	100.0	50.0	97.9	100.0	100.0	100.0	100.0
5	3.3	16.7	13.3	53.8	76.7	100.0	20.0	70.2	10.0	43.3	100.0	100.0	100.0	100.0
6	23.3	76.4	13.3	53.8	100.0	100.0	100.0	100.0	73.3	100.0	100.0	100.0	100.0	100.0
7	10.0	43.3	6.7	31.0	100.0	100.0	100.0	100.0	76.7	100.0	100.0	100.0	100.0	100.0
8	13.3	53.8	20.0	70.2	96.7	100.0	93.3	100.0	66.7	99.8	100.0	100.0	100.0	100.0
9	0.0	0.0	0.0	0.0	93.3	100.0	0.0	0.0	30.0	85.7	100.0	100.0	100.0	100.0
10	0.0	0.0	0.0	0.0	100.0	100.0	83.3	100.0	46.7	96.9	100.0	100.0	100.0	100.0
11	0.0	0.0	0.0	0.0	3.3	16.7	0.0	0.0	0.0	0.0	100.0	100.0	100.0	100.0
12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	13.3	57.1	13.3	57.1
13	0.0	0.0	0.0	0.0	3.3	16.7	0.0	0.0	0.0	0.0	100.0	100.0	100.0	100.0
14	0.0	0.0	0.0	0.0	6.7	31.0	0.0	0.0	0.0	0.0	73.3	100.0	73.3	100.0
15	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	13.3	57.1	13.3	57.1
16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	6.7	33.3
17	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
18	0.0	0.0	0.0	0.0	0.0	0.0	3.3	16.7	0.0	0.0	0.0	0.0	100.0	100.0
19	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	60.0	99.8
20	0.0	0.0	0.0	0.0	0.0	0.0	3.3	16.7	0.0	0.0	0.0	0.0	100.0	100.0
21	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	20.0	73.6
22-24	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Avg	3.2	12.2	7.1	16.9	38.6	44.3	28.8	36.4	21.4	35.0	54.2	58.9	**66.1**	**75.9**
# Solved	7	7	7	7	13	13	11	11	10	10	15	15	**20**	**20**

Table 4: **Ablation study and fine-tuning.** A series of ablation studies on the GPT-3.5 model validate the efficacy of the proposed method by systematically removing components of our approach. Fine-tuned GPT-3.5 improves the success rate of designs but does not increase the number of successful circuit designs.

Model Task ID	GPT3.5 (SPICE)		GPT3.5 (w/o context)		GPT3.5 (w/o CoT)		GPT3.5 (w/o flow)		GPT3.5		GPT3.5 (fine-tune)	
	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5
1	50.0	97.9	10.0	43.3	**100.0**	**100.0**	70.0	99.9	86.7	**100.0**	100.0	100.0
2	46.7	96.9	3.3	16.7	**93.3**	**100.0**	70.0	99.9	70.0	99.9	86.7	**100.0**
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.3	16.7	**40.0**	**95.8**
4	23.3	76.4	26.7	81.5	0.0	0.0	46.7	96.9	50.0	97.9	**80.0**	**100.0**
5	10.0	43.3	0.0	0.0	3.3	16.7	6.7	31.0	10.0	43.3	**20.0**	**73.6**
6	53.3	98.6	86.7	**100.0**	83.3	**100.0**	53.3	98.6	73.3	**100.0**	100.0	100.0
7	83.3	**100.0**	26.7	81.5	76.7	**100.0**	40.0	94.0	76.7	**100.0**	86.7	**100.0**
8	60.0	99.4	33.3	89.1	53.3	98.6	10.0	43.3	66.7	99.8	**93.3**	**100.0**
9	3.3	16.7	0.0	0.0	**53.3**	**98.6**	0.0	0.0	30.0	85.7	26.7	84.6
10	3.3	16.7	6.7	31.0	3.3	16.7	10.0	43.3	**46.7**	**96.9**	40.0	95.8
11-24	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Avg	13.9	26.9	8.1	18.5	19.4	26.3	12.8	25.3	21.4	35.0	**28.1**	**39.6**
# Solved	9	9	7	7	8	8	8	8	10	10	**10**	**10**

Main Results. Table 3 compares our LLM agent, AnalogCoder, which is based on GPT-4o and incorporates prompt engineering, flow feedback, and a circuit tool library with other LLM-based methods. To ensure a fair comparison and highlight the tool library's impact, we applied our strategies across all LLMs but specifically excluded the circuit tool library from GPT-4o to isolate its effects. The results indicate that Llama-3 and DeepSeek-V2, the latest open-source models, demonstrate a marginally superior capability in circuit design compared to GPT-3.5. However, other open-source models still exhibit a certain gap compared to GPT-3.5, although some surpassed GPT-3.5 in normal Python coding tasks [57]. This is primarily because circuit design requires both coding skills and specific background knowledge; hence, general LLMs tend to perform better. GPT-4o is still the best LLM for analog circuit design, generally consistent with other findings on its performance in coding tasks [41, 60, 57, 20]. Benefiting from the circuit tool library, GPT-4o and Llama-3 can further utilize existing circuits to design more challenging composite circuits, significantly enhancing their design capabilities. More results can be seen in Appendix Sec. A.7.

Ablations. We evaluated the effectiveness of various components within our approach using the GPT-3.5 model, with results presented in Table 4. Specifically, "GPT-3.5 (SPICE)" refers to the GPT-3.5 in which the LLM is prompted to generate SPICE codes rather than Python. The variants "GPT-3.5 (w/o context)" and "GPT-3.5 (w/o CoT)" explore the impact on performance when omitting in-context information and Chain-of-Thought reasoning from the prompts, respectively. Furthermore, "GPT-3.5 (w/o flow)" indicates a setup in which our proposed design flow was not utilized, and only the first generated codes were applied for functional testing. The findings consistently show that removing these components leads to a decrease in design performance.
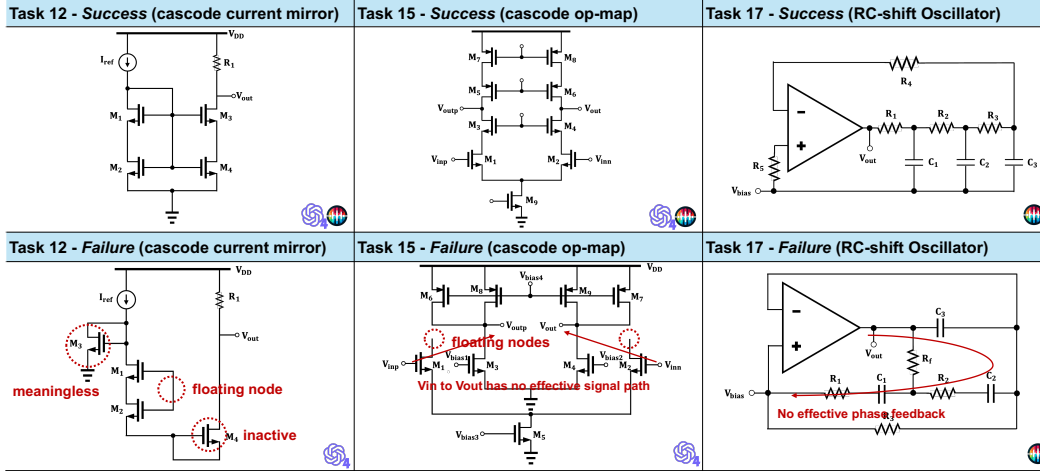
Figure 6: **Visualization for successful and failed designs.** The LLM model source utilized for this circuit's design is detailed in the lower right corner.

Fine-tuning. We employed a 3-fold cross-validation for fine-tuning evaluation, using two subsets of design tasks for fine-tuning and the remaining one for testing. Fine-tuning was conducted using the API of GPT-3.5 with two epochs. The results are shown in Table 4. Fine-tuned GPT-3.5 generally performs better on design tasks, as fine-tuning helps standardize design outputs through correct examples and reduces common syntax and design errors. However, due to the inherent limitations of the GPT-3.5 base model, fine-tuned models struggle to design additional circuits when data is limited.

Visualization. Several successful and failed circuit design diagrams are in Fig. 6, with icons at the bottom of the figures indicating the corresponding source LLMs. It can be observed that even the slightest discrepancy can render a circuit non-functional. More results are shown in Appendix Fig. 8.

Attempt Times. The number of attempts is a hyper-parameter to maximize the benefit-cost ratio. We tested three tasks (Task ID=7, 8, 10) and conducted 50 trials with a maximum of 5 attempts per trial using GPT-3.5. Fig. 7 shows the distribution of successful design attempts across 50 trials. Results show that most designs are completed within three attempts. If a design is not achieved within this limit, the LLM's further attempts are unlikely to succeed, and token consumption continues to rise. Consequently, we have set the default number of attempts to three. The number of design attempts for composite circuits has been limited to two because they involve more complex and challenging fixes and have high generation costs.
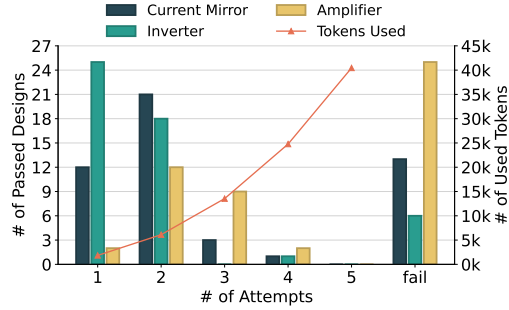

Figure 7: **Attempt Times.** 50 trials for each task with a maximum of 5 attempts across design tasks, the findings indicate that success probability significantly drops after the third attempt.

5 Conclusion

This paper proposes AnalogCoder, a training-free LLM agent for automatic analog circuit design. It innovatively transforms the analog circuit design tasks into the generation of Python code, significantly reducing the complexity faced by LLMs. At the same time, it is equipped with crafted prompts, a feedback-enhanced design flow, and a circuit tool library, effectively enhancing the success rate of the designs. Also, we provide an open-source analog design benchmark for future research. This work facilitates the complex, time-consuming, and error-prone process of analog circuit design, enabling individuals with limited design experience to easily create analog circuits.

9

Limitation and Societal Impact. Currently, LLMs lack the capability to design highly complex analog circuits, but future advancements may address this. Additionally, their use is constrained by costs and the availability of computational resources. As LLMs improve, there is a risk of AI replacing parts of human roles, a challenge common to all projects involving LLMs.

References

[1] L. Zhang, M. Z. Chen, W. Tang, J. Y. Dai, L. Miao, X. Y. Zhou, S. Jin, Q. Cheng, and T. J. Cui, "A wireless communication scheme based on space-and frequency-division multiplexing using digital metasurfaces," *Nature electronics*, vol. 4, no. 3, pp. 218–227, 2021.

[2] B. Chatterjee, A. Datta, M. Nath, G. Kumar, N. Modak, and S. Sen, "A 65nm 63.3 μw 15mbps transceiver with switched-capacitor adiabatic signaling and combinatorial-pulse-position modulation for body-worn video-sensing ar nodes," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65. IEEE, 2022, pp. 276–278.

[3] Q. Zheng, Q. Tang, Z. L. Wang, and Z. Li, "Self-powered cardiovascular electronic devices and systems," *Nature Reviews Cardiology*, vol. 18, no. 1, pp. 7–21, 2021.

[4] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[5] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[6] R. Zhong, X. Du, S. Kai, Z. Tang, S. Xu, H.-L. Zhen, J. Hao, Q. Xu, M. Yuan, and J. Yan, "Llm4eda: Emerging progress in large language models for electronic design automation," *arXiv preprint arXiv:2401.12224*, 2023.

[7] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-chat: Challenges and opportunities in conversational hardware design," in *ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2023, pp. 1–6.

[8] K. Chang, Y. Wang, H. Ren, M. Wang, S. Liang, Y. Han, H. Li, and X. Li, "Chipgpt: How far are we from natural language hardware design," *arXiv preprint arXiv:2305.14019*, 2023.

[9] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2023.

[10] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg, and R. Karri, "Autochip: Automating hdl generation using llm feedback," *arXiv preprint arXiv:2311.04887*, 2023.

[11] Y. Fu, Y. Zhang, Z. Yu, S. Li, Z. Ye, C. Li, C. Wan, and Y. C. Lin, "Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[12] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "Verilogeval: Evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–8.

[13] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2024, pp. 722–727.

[14] Y. Tsai, M. Liu, and H. Ren, "Rtlfixer: Automatically fixing rtl syntax errors with large language models," *arXiv preprint arXiv:2311.16543*, 2023.

[15] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie, "Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution," *arXiv preprint arXiv:2312.08617*, 2023.

[16] Z. Pei, H.-L. Zhen, M. Yuan, Y. Huang, and B. Yu, "Betterv: Controlled verilog generation with discriminative guidance," *arXiv preprint arXiv:2402.03375*, 2024.

[17] H. Wu, Z. He, X. Zhang, X. Yao, S. Zheng, H. Zheng, and B. Yu, "Chateda: A large language model powered autonomous agent for eda," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2024.

[18] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayrak-taroglu *et al.*, "Chipnemo: Domain-adapted llms for chip design," *arXiv preprint arXiv:2311.00176*, 2023.

[19] X. Yao, H. Li, T. H. Chan, W. Xiao, M. Yuan, Y. Huang, L. Chen, and B. Yu, "Hdldebugger: Streamlining hdl debugging with large language models," *arXiv preprint arXiv:2403.11671*, 2024.

[20] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[21] J. K. Ousterhout, "An introduction to tcl and tk," 1993.

[22] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[23] D. A. Johns and K. Martin, *Analog integrated circuit design*. John Wiley & Sons, 2008.

[24] B. Razavi, *Design of Analog CMOS Integrated Circuits*. McGraw-Hill, Inc., 2000.

[25] P. E. Allen, R. Dobkin, and D. R. Holberg, *CMOS analog circuit design*. Elsevier, 2011.

[26] D. Thomas and P. Moorby, *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[27] H. Chaoui, "Cmos analogue adder," *Electronics Letters*, vol. 31, no. 3, pp. 180–181, 1995.

[28] A. Vladimirescu, *The SPICE book*. John Wiley & Sons, Inc., 1994.

[29] Z. Dong, W. Cao, M. Zhang, D. Tao, Y. Chen, and X. Zhang, "Cktgnn: Circuit graph neural network for electronic design automation," in *The Eleventh International Conference on Learning Representations (ICLR)*, 2022.

[30] H. Wang, K. Wang, J. Yang, L. Shen, N. Sun, H.-S. Lee, and S. Han, "Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[31] W. Lyu, F. Yang, C. Yan, D. Zhou, and X. Zeng, "Batch bayesian optimization via multi-objective acquisition ensemble for automated analog circuit design," in *International conference on machine learning (ICML)*. PMLR, 2018, pp. 3306–3314.

[32] D. Krylov, P. Khajeh, J. Ouyang, T. Reeves, T. Liu, H. Ajmal, H. Aghasi, and R. Fox, "Learning to design analog circuits to meet threshold specifications," in *International Conference on Machine Learning (ICML)*. PMLR, 2023, pp. 17 858–17 873.

[33] B. Razavi, *Design of Analog CMOS Integrated Circuits*, 1st ed. USA: McGraw-Hill, Inc., 2000.

[34] F. Salvaire, "Pyspice," 2021.

[35] M. A. M. Khan, M. S. Bari, X. L. Do, W. Wang, M. R. Parvez, and S. Joty, "xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval," *arXiv preprint arXiv:2303.03004*, 2023.

[36] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2023, pp. 5673–5684.

[37] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, and Z. Sui, "A survey on in-context learning," *arXiv preprint arXiv:2301.00234*, 2022.

[38] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems (NeurIPS)*, vol. 35, pp. 24 824–24 837, 2022.

[39] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman *et al.*, "Multipl-e: a scalable and polyglot approach to benchmarking neural code generation," *IEEE Transactions on Software Engineering (TSE)*, 2023.

[40] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[41] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," *arXiv preprint arXiv:2306.08568*, 2023.

[42] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems (NeurIPS)*, vol. 33, pp. 1877–1901, 2020.

[43] S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou *et al.*, "Metagpt: Meta programming for multi-agent collaborative framework," *arXiv preprint arXiv:2308.00352*, 2023.

[44] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[45] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama, "Is self-repair a silver bullet for code generation?" in *The Twelfth International Conference on Learning Representations (ICLR)*, 2023.

[46] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, "Voyager: An open-ended embodied agent with large language models," in *Intrinsically-Motivated and Open-Ended Learning Workshop@ NeurIPS2023*, 2023.

[47] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian *et al.*, "Toolllm: Facilitating large language models to master 16000+ real-world apis," *arXiv preprint arXiv:2307.16789*, 2023.

[48] H. Liu, C. Li, Q. Wu, and Y. J. Lee, "Visual instruction tuning," *Advances in neural information processing systems (NeurIPS)*, vol. 36, 2024.

[49] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[50] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the 14th international joint conference on Artificial intelligence-Volume 2 (IJCAI)*, 1995, pp. 1137–1143.

[51] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, 2024.

[52] AI@Meta, "Llama 3 model card," 2024.

[53] DeepSeek-AI, "Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model," 2024.

[54] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[55] R. Li, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, L. Jia, J. Chim, Q. Liu *et al.*, "Starcoder: may the source be with you!" *Transactions on Machine Learning Research (TMLR)*, 2023.

[56] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[57] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 36, 2024.

[58] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. S. Liang, "Spoc: Search-based pseudocode to code," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.

[59] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[60] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang *et al.*, "Qwen technical report," *arXiv preprint arXiv:2309.16609*, 2023.

[61] M. A. team, "Codestral: Hello, world!" 2024.

[62] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[63] E. B. Wilson, "Probable inference, the law of succession, and statistical inference," *Journal of the American Statistical Association*, vol. 22, no. 158, pp. 209–212, 1927.

[64] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman *et al.*, "Do as i can, not as i say: Grounding language in robotic affordances," *arXiv preprint arXiv:2204.01691*, 2022.

[65] P. J. Ashenden, *The designer's guide to VHDL.* Morgan kaufmann, 2010.

[66] G. Chen, W. Chen, Y. Ma, H. Yang, and B. Yu, "DAMO: Deep agile mask optimization for full chip scale," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020.

[67] G. Chen, Z. Yu, H. Liu, Y. Ma, and B. Yu, "DevelSet: Deep neural level set for instant mask optimization," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021.

[68] G. Chen, Z. Pei, H. Yang, Y. Ma, B. Yu, and M. Wong, "Physics-informed optical kernel regression using complex-valued neural fields," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.

[69] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi *et al.*, "A graph placement methodology for fast chip design," *Nature*, vol. 594, no. 7862, pp. 207–212, 2021.

[70] Y. Lai, Y. Mu, and P. Luo, "Maskplace: Fast chip placement via reinforced visual representation learning," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, pp. 24 019–24 030, 2022.

[71] Y. Lai, J. Liu, Z. Tang, B. Wang, J. Hao, and P. Luo, "Chipformer: Transferable chip placement via offline decision transformer," in *International Conference on Machine Learning (ICML)*. PMLR, 2023, pp. 18 346–18 364.

[72] Y. Lai, J. Liu, D. Z. Pan, and P. Luo, "Scalable and effective arithmetic tree generation for adder and multiplier designs," *arXiv preprint arXiv:2405.06758*, 2024.

[73] B. Han, X. Wang, Y. Wang, J. Yan, and Y. Tian, "New interaction paradigm for complex eda software leveraging gpt," *arXiv preprint arXiv:2307.14740*, 2023.

[74] M. DeLorenzo, A. B. Chowdhury, V. Gohil, S. Thakur, R. Karri, S. Garg, and J. Rajendran, "Make every move count: Llm-based high-quality rtl code generation using mcts," *arXiv preprint arXiv:2402.03289*, 2024.

[75] W. Fang, M. Li, M. Li, Z. Yan, S. Liu, H. Zhang, and Z. Xie, "Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms," *arXiv preprint arXiv:2402.00386*, 2024.

[76] M. Li, W. Fang, Q. Zhang, and Z. Xie, "Specllm: Exploring generation and review of vlsi design specification with large language model," *arXiv preprint arXiv:2401.13266*, 2024.

A Appendix

A.1 Benchmark Details

We list all details of the analog circuit design benchmark details in Table 5.

Table 5: Benchmark Details

Id	Type	Design Task Description	Input	Output	Composite Circuit
1	Amplifier	a single-stage common-source amplifier with resistive load R	Vin	Vout	No
2	Amplifier	a three-stage amplifier with single input and output (each stage is common-source with resistive load)	Vin	Vout	No
3	Amplifier	a common-drain amplifier (a.k.a. a source follower) with resistive load R (output Vout at the source)	Vin	Vout	No
4	Amplifier	a single-stage common-gate amplifier with resistive load R (input signal Vin must be applied at the source terminal)	Vin, Vbias	Vout	No
5	Amplifier	a single-stage cascode amplifier with two NMOS transistors provides a single-ended output through a resistive load R	Vin, Vbias	Vout	No
6	Inverter	a NMOS inverter with resistive load R	Vin	Vout	No
7	Inverter	a logical inverter with 1 NMOS and 1 PMOS	Vin	Vout	No
8	CurrentMirror	a simple NMOS constant current source with resistive load R	Vbias	Vout	No
9	Amplifier	a single-stage amplifier (common-source with PMOS diode-connected load (gate and drain are shorted))	Vin	Vout	No
10	Amplifier	a two-stage amplifier with a Miller compensation capacitor	Vin	Vout	No
11	Opamp	a differential opamp with an active PMOS current mirror load, a tail current source, and two outputs	Vinp, Vinn, Vbias	Voutp, Vout	No
12	CurrentMirror	A cascode current mirror with 4 mosfets (2 stacked at input side with diode-connected, 2 stacked at output side), reference current source input Iref (connected to Vdd) and resistive load R	Iref	Iout	No
13	Opamp	a single-stage differential common-source opamp with dual resistive loads, tail current, and a single output	Vinp, Vinn	Vout	No
14	Opamp	a two-stage differential opamp (first stage: common-source with an active load and a tail current, second stage: common-source with an active load)	Vinp, Vinn, Vbias1, Vbias2, Vbias3	Voutp, Vout	No
15	Opamp	a single-stage telescopic cascode opamp with two outputs (4 nmos as cascode input pair, 4 pmos as cascode loads, and 1 tail current)	Vinp, Vinn, Vbias1, Vbias2, Vbias3, Vbias4	Voutp, Vout	No
16	Oscillator	an RC phase-shift oscillator	-	Vout	Yes
17	Oscillator	a Wien Bridge oscillator	-	Vout	Yes
18	Integrator	an Opamp integrator with resistor R1 and capacitor Cf	Vin	Vout	Yes
19	Differentiator	an Opamp differentiator with resistor Rf and capacitor C1	Vin	Vout	Yes
20	Adder	an Opamp adder to make Vout=-(Vin1+Vin2)	Vin1, Vin2	Vout	Yes
21	Subtractor	an Op-amp subtractor to make Vout=Vin2-Vin1	Vin1, Vin2	Vout	Yes
22	Schmitt Trigger	a non-inverting Schmitt trigger with positive feedback op-amp	Vin	Vout	Yes
23	VCO	a voltage-controlled oscillator	Vin	Vout	Yes
24	PLL	a phase-locked loop	CLK_{ref}	CLK_p	Yes

The functional testing criteria for all categories of analog circuit design are enumerated in Table 6.

A.2 Complete Prompts

We provide the complete prompts as follows.

User

You are an analog integrated circuits expert.

Assistant (Design prompt for basic circuits)

You aim to design a topology for a given circuit described in the text.
Please ensure your designed circuit topology works properly and achieves the design requirements.
Here is an example:

Question

Design a 2-stage amplifier (first stage: a common-source stage with current-source load, second stage: a common-source stage with resistor load).
Input node name: Vin, Vbias.
Output node name: Vout.

Answer

Task 1

Components Needed
- **NMOS Transistors**: M1 and M3
- **PMOS Transistors**: M2 (used as the current source in the first stage)
- **Resistors**: R1 for the second stage load
- **Power Supply**: Vdd for DC supply
- **Input Signal Source**: Vin, Vbias for biasing and signal input
- **Capacitors**: Not specified but can be included for coupling and bypass applications if required

Stage 1: Common-Source Amplifier with Current Source Load
1. **Transistor Setup**:
   - **M1** (NMOS) as the main amplifying transistor.
   - Gate of **M1** is connected to the input node **Vin**.
   - Source of **M1** connected to the ground.
   - Drain of **M1** connected to the drain of **M2**.
2. **Biasing**:
   - **Vin** provides the input signal.
   - **Vbias** is used to bias **M2** (PMOS), ensuring it operates as a current source.
3. **Current Source Load (M2)**:
   - **M2**, a PMOS transistor, is configured as a current source.
   - The source of **M2** is connected to **Vdd**, and its gate is connected to **Vbias**.
   - Drain of **M2** is connected to the drain of **M1**, providing a high-impedance load.

Stage 2: Common-Source Amplifier with Resistor Load
1. **Transistor Setup**:
   - **M3** (NMOS) as the main amplifying transistor for the second stage.
   - Gate of **M3** connected to the drain of **M1**.
   - Source of **M3** connected to the ground.
   - Drain of **M3** connected to **Vout** through resistor **R1**.
2. **Load and Coupling**:
   - **R1** connects the drain of **M3** to **Vdd**. This resistor converts the current through **M3** into an output voltage.

Task 2

```python
from PySpice.Spice.Netlist import Circuit
from PySpice.Unit import *

circuit = Circuit('Two-Stage Amplifier')
# Define the MOSFET models
circuit.model('nmos_model', 'nmos', level=1, kp=100e-6, vto=0.5)
circuit.model('pmos_model', 'pmos', level=1, kp=50e-6, vto=-0.5)

# Power Supplies
circuit.V('dd', 'Vdd', circuit.gnd, 5.0) # 5V power supply
circuit.V('in', 'Vin', circuit.gnd, 1.0) # 1V input for bias voltage
circuit.V('bias', 'Vbias', circuit.gnd, 4.0) # 4V input for bias voltage

# First Stage: Common-Source with Active Load
circuit.MOSFET('1', 'Drain1', 'Vin', circuit.gnd, circuit.gnd,
model='nmos_model', w=50e-6, l=1e-6)
circuit.MOSFET('2', 'Drain1', 'Vbias', 'Vdd', 'Vdd',
model='pmos_model', w=100e-6, l=1e-6)

# Second Stage: Common-Source with Resistor Load
circuit.MOSFET('3', 'Vout', 'Drain1', circuit.gnd,
circuit.gnd, model='nmos_model', w=100e-6, l=1e-6)
circuit.R('1', 'Vout', 'Vdd', 1@u_kOhm)
# Analysis Part
simulator = circuit.simulator()
```

As you have seen, the output of your designed topology should consist of two tasks:

1. Give a detailed design plan about all devices and their interconnectivity nodes and properties.
2. Write a complete Python code, describing the topology of integrated analog circuits according to the design plan.

Please make sure your Python code is compatible with PySpice.
Please give the runnable code without any placeholders.
Do not write other redundant codes after `simulator = circuit.simulator()`.

Tips

There are some tips you should remember all the time:

1. For the MOSFET definition `circuit.MOSFET(name, drain, gate, source, bulk, model, w=w1, l=l1)`, be careful about the parameter sequence.
2. You should connect the bulk of a MOSFET to its source.
3. Please use the MOSFET threshold voltage, when setting the bias voltage.
4. Avoid giving any AC voltage in the sources, just consider the operating points.
5. Make sure the input and output node names appear in the circuit.
6. Avoid using subcircuits.
7. Use nominal transistor sizing.
8. Assume the Vdd = 5.0 V.

Question

Design [TASK].
Input node name: [INPUT].
Output node name: [OUTPUT].

Answer

I have the following implemented subcircuits you can directly call them by Python code with the Pyspice library, we list all basic information on them.

[TABLE]

Now, you need to design [TASK]. Please maximize the design success rate, thus the circuit with two inputs and the highest possible gain has greater flexibility. Please choose the subcircuits from the above table you will use and make the number of chosen subcircuits as few as possible.

Please give out the IDs of the subcircuits that you choose, and enumerate them in a Python list like [0].

You aim to design a topology for a given circuit described in the text. Please ensure your designed circuit topology works properly and achieves the design requirements. To make the task easier, I provide you with some existing subcircuits you can directly use by calling functions in Python with the PySpice library. Now I would like you to help me design a complex analog circuit based on them.
Here is an example:

Question

Design an opamp with 470 ohm resistance load.
Input node name: in
Output node name: out
You can directly use the following subcircuits.

Subcircuits Info

Id	Circuit Type	Gain/Differential-mode gain	Common-mode gain	Input	Output
-	Opamp	10.00×10^0	0.00×10^0	Vin	Vout

Call Info

To use them, please insert the following codes.

```
from example_lib import *
# declare the subcircuit
circuit.subcircuit('BasicOperationalAmplifier())
# create a subcircuit instance
circuit.X('1', 'BasicOperationalAmplifier', 'Vin', 'Vout')
```

Answer

```
from PySpice.Spice.Netlist import Circuit
from PySpice.Unit import *
from example_lib import *

circuit = Circuit('Operational Amplifier')

# Define the MOSFET models
circuit.model('nmos_model', 'nmos', level=1, kp=100e-6, vto=0.5)
circuit.model('pmos_model', 'pmos', level=1, kp=50e-6, vto=-0.5)


circuit.V('input', 'in', circuit.gnd, 2.5@u_V)
circuit.subcircuit(BasicOperationalAmplifier())
circuit.X('op', 'BasicOperationalAmplifier', 'in', circuit.gnd, 'out')
R = 470
circuit.R('load', 'out', circuit.gnd, R)

simulator = circuit.simulator()
```

As you have seen, the output of your designed topology should be in a complete Python code, describing the topology of integrated analog circuits according to the design plan.
Please make sure your Python code is compatible with PySpice. Please give the runnable code without any placeholders. Do not write other redundant codes after `simulator = circuit.simulator()`.

1. For the MOSFET definition circuit: `MOSFET(name, drain, gate, source, bulk, model, w=w1, l=l1)`, be careful about the parameter sequence.

2. You should connect the bulk of a MOSFET to its source.

3. Please use the MOSFET threshold voltage when setting the bias voltage.

4. Avoid giving any AC voltage in the sources, just consider the operating points.

5. Make sure the input and output node names appear in the circuit.

6. Assume the Vdd = 5.0 V. Do not need to add the power supply for subcircuits.

Question

Design [TASK].
Input node name: [INPUT].
Output node name: [OUTPUT].
You can directly use the following subcircuits.

Subcircuits Info

[SUBCIRCUITS_INFO]

Note

[NOTE_INFO]

Call Info

To use them, please insert the following codes.
[CALL_INFO]

Answer

In the prompts, the sections enclosed in square brackets serve as placeholders and will be replaced with specific content pertinent to the designated task. The placeholders [TASK], [INPUT], and [OUTPUT] are replaced with information regarding the design of circuits as specified in Table 5.

The placeholder [TABLE] is used to display key information relevant to the current circuit tool library, with a specific example provided in Table 7.

The placeholder [SUBCIRCUITS_INFO] gives the basic info selected subcircuits' information in one table. An example is as follows.

SUBCIRCUITS_INFO

Id	Circuit Type	Gain/Differential-mode gain	Common-mode gain	Input	Output
11	Opamp	193.98	-173.70	Vinp, Vinn	Vout

The placeholder [NOTE_INFO] gives supplementary information for using subcircuits, which is built based on the circuit tool library as shown in Table 7 and the text template. An example is as follows, wherein the *SingleStageOpamp* is the function name.

NOTE_INFO

The Vinn of SingleStageOpamp is the inverting input.
The Vinp of SingleStageOpamp is the non-inverting input.
The DC operating voltage for Vinn/Vinp is 2.5 V.

(only for the Oscillator circuit)
Due to the operational range of the op-amp being 0 to 5V, please connect the nodes that were originally grounded to a 2.5V DC power source.
Please increase the gain as much as possible to maintain oscillation.

The placeholder [CALL_INFO] provides the function invocation methods when using the subcircuits. An example is as follows.

The implement of the subcircuit *SingleStageOpamp* is saved in the circuit tool library as values as shown in Fig. 5, which is automatically transformed into the SubCircuit class through scripting. An example of the implementation is as follows.

```
1  from PySpice.Unit import *
2  from PySpice.Spice.Netlist import SubCircuitFactory
3  class SingleStageOpamp(SubCircuitFactory):
4      NAME = ('SingleStageOpamp')
5      NODES = ('Vinp', 'Vinn', 'Vout')
6      def __init__(self):
7          super().__init__()
8          # Define the MOSFET models
9          self.model('nmos_model', 'nmos', level=1, kp=100e-6, vto=0.5)
10         self.model('pmos_model', 'pmos', level=1, kp=50e-6, vto=-0.5)
11         # Power Supplies
12         self.V('dd', 'Vdd', self.gnd, 5.0)  # 5V power supply
13         self.V('bias', 'Vbias', self.gnd, 1.5)  # Bias voltage for the tail current source M3
14         # Input Voltage Sources for Differential Inputs
15         # Differential Pair and Tail Current Source
16         self.MOSFET('1', 'Voutp', 'Vinp', 'Source3', 'Source3', model='nmos_model', w=50e-6, l=1e-6)
17         self.MOSFET('2', 'Vout', 'Vinn', 'Source3', 'Source3', model='nmos_model', w=50e-6, l=1e-6)
18         self.MOSFET('3', 'Source3', 'Vbias', self.gnd, self.gnd, model='nmos_model', w=100e-6, l=1e-6)
19         # Active Current Mirror Load
20         self.MOSFET('4', 'Voutp', 'Voutp', 'Vdd', 'Vdd', model='pmos_model', w=100e-6, l=1e-6)
21         self.MOSFET('5', 'Vout', 'Voutp', 'Vdd', 'Vdd', model='pmos_model', w=100e-6, l=1e-6)
```

We also give the prompts of 'GPT-3.5 (SPICE)', 'GPT-3.5 (w/o context)', and 'GPT-3.5 (w/o CoT)' in our ablation studies. 'GPT-3.5 (SPICE)' makes the LLMs generate SPICE code rather than Python code, whereas NgSPICE is a widely-used open-source version of SPICE. 'GPT-3.5 (w/o context)' disregards the context example in the prompt, meaning that a demonstrative design question and answer should not be provided. 'GPT-3.5 (w/o CoT)' omits the Chain-of-Thought component, meaning it allows the language model to generate code directly without first enumerating the required components and their connections. We have omitted portions of the content identical to the original prompt.

```
* parameters: name, drain, gate, source, bulk, model, w, l
M1 Drain1 Vin 0 0 nmos_model w=50e-6 l=1e-6
M2 Drain1 Vbias Vdd Vdd pmos_model w=100e-6 l=1e-6

* Second Stage: Common-Source with Resistor Load
M3 Vout Drain1 0 0 nmos_model w=100e-6 l=1e-6
R1 Vout Vdd 1k

.end
```

As you have seen, the output of your designed topology should consist of two tasks:

1. Give a detailed design plan about all devices and their interconnectivity nodes and properties.
2. Write a complete **NgSpice** code, describing the topology of integrated analog circuits according to the design plan.

Please give the runnable code without any placeholders.
Do not write other redundant codes after `.end`.

... (same with Assistant (Design prompt for basic circuits))

Assistant (Design prompt for basic circuits / without in-context learning)

You aim to design a topology for a given circuit described in the text.
Please ensure your designed circuit topology works properly and achieves the design requirements.
The output of your designed topology should consist of two tasks:

1. Give a detailed design plan about all devices and their interconnectivity nodes and properties.
2. Write a complete Python code, describing the topology of integrated analog circuits according to the design plan.

Please make sure your Python code is compatible with PySpice.
Please give the runnable code without any placeholders.
Do not write other redundant codes after `simulator = circuit.simulator()`.
For importing libraries, you can use:

```
from PySpice.Spice.Netlist import Circuit
from PySpice.Unit import *
```

For the mosfet, you can refer to the following code:

```
circuit.model('nmos_model', 'nmos', level=1, kp=100e-6, vto=0.5)
circuit.model('pmos_model', 'pmos', level=1, kp=50e-6, vto=-0.5)
circuit.MOSFET('1', 'Vout', 'Vin', circuit.gnd, circuit.gnd,
model='nmos_model', w=50e-6, l=1e-6)
```

For the resistor and the voltage source, you can can refer to the following code:

```
circuit.R('1', 'Vout', 'Vdd', 1000)
circuit.V('dd', 'Vdd', circuit.gnd, 5.0)
```

There are some tips you should remember all the time:

... (same with Assistant (Design prompt for basic circuits))

Assistant (Design prompt for basic circuits / without CoT)

You aim to design a topology for a given circuit described in the text.
Please ensure your designed circuit topology works properly and achieves the design requirements.
Here is an example:

Question

Design a 2-stage amplifier (first stage: a common-source stage with current-source load, second stage: a common-source stage with resistor load).
Input node name: Vin, Vbias.

Output node name: Vout.

Answer

```python
from PySpice.Spice.Netlist import Circuit
from PySpice.Unit import *

circuit = Circuit('Two-Stage Amplifier')
# Define the MOSFET models
circuit.model('nmos_model', 'nmos', level=1, kp=100e-6, vto=0.5)
circuit.model('pmos_model', 'pmos', level=1, kp=50e-6, vto=-0.5)

# Power Supplies
circuit.V('dd', 'Vdd', circuit.gnd, 5.0) # 5V power supply
circuit.V('in', 'Vin', circuit.gnd, 1.0) # 1V input for bias voltage
circuit.V('bias', 'Vbias', circuit.gnd, 4.0) # 4V input for bias voltage

# First Stage: Common-Source with Active Load
circuit.MOSFET('1', 'Drain1', 'Vin', circuit.gnd, circuit.gnd,
model='nmos_model', w=50e-6, l=1e-6)
circuit.MOSFET('2', 'Drain1', 'Vbias', 'Vdd', 'Vdd',
model='pmos_model', w=100e-6, l=1e-6)

# Second Stage: Common-Source with Resistor Load
circuit.MOSFET('3', 'Vout', 'Drain1', circuit.gnd,
circuit.gnd, model='nmos_model', w=100e-6, l=1e-6)
circuit.R('1', 'Vout', 'Vdd', 1@u_kOhm)
# Analysis Part
simulator = circuit.simulator()
```

As you have seen, the output of your designed topology should be in a complete Python code, describing the topology of integrated analog circuits according to the design plan.
Please make sure your Python code is compatible with PySpice.
Please give the runnable code without any placeholders.
Do not write other redundant codes after `simulator = circuit.simulator()`.
There are some tips you should remember all the time:

... (same with Assistant (Design prompt for basic circuits))

A.3 Circuit Tool Library Overview

As discussed in the circuit tool library, all successfully implemented basic circuits will be saved into the tool library. Information such as the attributes of the circuits will be stored in the library as keys and values. The key information of the tool library is listed in Table 7.

A.4 Examples of Generated Circuits

We give two examples of the circuits generated by the AnalogCoder: the basic circuit (Task ID=15, a cascade op-map) and the composite circuit (Task ID=16, an RC shift Oscillator).

Here is the basic circuit example.

```python
1  from PySpice.Spice.Netlist import Circuit
2  from PySpice.Unit import *
3  circuit = Circuit('Single-Stage Telescopic Cascode Opamp')
4  # Define the MOSFET models
5  circuit.model('nmos_model', 'nmos', level=1, kp=100e-6, vto=0.5)
6  circuit.model('pmos_model', 'pmos', level=1, kp=50e-6, vto=-0.5)
7  # Power Supply
8  circuit.V('dd', 'Vdd', circuit.gnd, 5.0) # 5V power supply
9  # Input and Bias Voltages
10 circuit.V('inp', 'Vinp', circuit.gnd, 1.49)
11 circuit.V('inn', 'Vinn', circuit.gnd, 1.49)
```

```
12    circuit.V('bias1', 'Vbias1', circuit.gnd, 1.5@u_V) # Bias for NMOS cascode
13    circuit.V('bias2', 'Vbias2', circuit.gnd, 1.5@u_V) # Bias for NMOS cascode
14    circuit.V('bias3', 'Vbias3', circuit.gnd, 3.5@u_V) # Bias for PMOS cascode
15    circuit.V('bias4', 'Vbias4', circuit.gnd, 3.5@u_V) # Bias for PMOS cascode
16    circuit.V('biasTail', 'VbiasTail', circuit.gnd, 1.0@u_V) # Bias for the tail current source
17    # NMOS Transistors
18    circuit.MOSFET('1', 'Drain1', 'Vinp', 'Source5', circuit.gnd, model='nmos_model', w=50e-6, l=1e-6)
19    circuit.MOSFET('2', 'Drain2', 'Vinn', 'Source5', circuit.gnd, model='nmos_model', w=50e-6, l=1e-6)
20    circuit.MOSFET('3', 'Voutp', 'Vbias1', 'Drain1', circuit.gnd, model='nmos_model', w=50e-6, l=1e-6)
21    circuit.MOSFET('4', 'Vout', 'Vbias2', 'Drain2', circuit.gnd, model='nmos_model', w=50e-6, l=1e-6)
22    circuit.MOSFET('5', 'Source5', 'VbiasTail', circuit.gnd, circuit.gnd, model='nmos_model', w=50e-6, l=1e-6)
23    # PMOS Transistors
24    circuit.MOSFET('6', 'Voutp', 'Vbias3', 'Vdd', 'Vdd', model='pmos_model', w=100e-6, l=1e-6)
25    circuit.MOSFET('7', 'Voutp', 'Vbias4', 'Vdd', 'Vdd', model='pmos_model', w=100e-6, l=1e-6)
26    circuit.MOSFET('8', 'Vout', 'Vbias3', 'Vdd', 'Vdd', model='pmos_model', w=100e-6, l=1e-6)
27    circuit.MOSFET('9', 'Vout', 'Vbias4', 'Vdd', 'Vdd', model='pmos_model', w=100e-6, l=1e-6)
28    # Analysis Part
29    simulator = circuit.simulator()
```

Here is the composite circuit example, which uses the subcircuit *SingleStageOpamp*.

```
1     from PySpice.Spice.Netlist import Circuit
2     from PySpice.Unit import *
3     from p11_lib import *
4     circuit = Circuit('RC Phase-Shift Oscillator')
5     # Define the power supply for non-grounded nodes
6     circuit.V('mid', 'mid', circuit.gnd, 2.5@u_V)  # 2.5V for biasing
7     # Define the operational amplifier from the library
8     circuit.subcircuit(SingleStageOpamp())
9     circuit.X('opamp', 'SingleStageOpamp', 'non_inv_input', 'inv_input', 'Vout')
10    # Feedback network
11    # RC Phase Shift Network
12    R_value = 10@u_kOhm
13    C_value = 10@u_nF
14    # First RC stage
15    circuit.R('1', 'Vout', 'n1', R_value)
16    circuit.C('1', 'n1', 'mid', C_value)
17    # Second RC stage
18    circuit.R('2', 'n1', 'n2', R_value)
19    circuit.C('2', 'n2', 'mid', C_value)
20    # Third RC stage
21    circuit.R('3', 'n2', 'inv_input', R_value)
22    circuit.C('3', 'inv_input', 'mid', C_value)
23    # Non-inverting input connected to midpoint bias
24    circuit.V('non_inv_input', 'non_inv_input', 'mid', 0@u_V)
25    simulator = circuit.simulator()
```

A.5   Details of Feedback-Enhanced Design Flow

According to the related method paragraph, the feedback-enhanced design flow includes 4 main steps, each step inspects a specific part of the circuit designed by the Language Model. Table 8 lists the specific checks conducted for each part.

A.6   Experimental Settings

Semiconductor Devices.   We focus on the analog integrated circuits in this work, which means all of our circuits include the (MOSFETs). We do not specify the MOS SPICE model to be used by the LLM, but we provide a simple example in the prompt, as demonstrated. This is a Level-1 model [24], which is the simplest form of the SPICE model designed to significantly reduce the complexity of the LLM's design task by using basic approximations to simulate semiconductor device behavior. The *kp* parameter represents the transconductance parameter of the transistor, typically measured in amperes per volt squared (A/V^2). The *vto* parameter is the threshold voltage, measured in volts (V).

All Python codes for circuits are tested and passed in the Python 3.10 and PySpice 1.5 versions.

LLM Settings. When employing LLMs for inference, we set temperature= 0.5 and top_p= 1.0 in all models to generate varied results in each trial.

Fine-tuning Settings. We have categorized the design tasks (basic circuits) that can be completed by GPT-3.5, GPT-4, and Llama-3 into three groups. We utilized the circuits designed successfully in two of these groups as data for fine-tuning. The remaining group was used for testing purposes, with other settings identical to GPT-3.5. The task grouping is detailed in Table 9. We tested using all three models for tasks not included in the fine-tuning training set (Task ID=12, 15), and none could correctly accomplish the design.

When collecting fine-tuning data, we replace all lines with `PySpice.Unit import` with `PySpice.Unit import *` to ensure the importation of all applicable units. Concurrently, we eliminate all content containing `u_V` and lines featuring `circuit.I`, due to their low frequency of occurrence in the fine-tuning dataset. As a result, the outcomes post-fine-tuning may exhibit instability.

A.7 Supplementary Experiments

Additional Results. We also tested the capabilities of other representative LLMs in designing analog circuits, including: Mixtral-8×7B [51], CodeLlama-7B, 13B, 34B [40], Llama2-70B [54], QwenCode-7B [60], DeepSeek-Coder-33B [20], and Codestral-22B [61]. The results can be seen in Table 10.

We also compared the results between the GPT-4 with circuit tool library and AnalogCoder (GPT-4o with circuit tool library), and the results are in Table 11. Based on the results, the capabilities of GPT-4 and GPT-4o in designing analog circuits are similar; however, GPT-4o successfully designed one additional circuit. Considering the overall costs, we have decided to implement our AnalogCoder based on GPT-4o.

Due to the model size and corresponding training data, these models could only successfully design no more than four analog circuits. Some other models not shown in Table 10, including Mistral-7B [62] (2 solved), Llama 3-8B [52] (1 solved), Qwen-1.5-110B [60] (2 solved), Llama 2-7B, 13B [54] (0 solved).

Additional Visualization. Additional visualizations are presented in Fig. 8.

We also present a waveform generated during the simulation of an RC-shift oscillator implemented by the AnalogCoder, as shown in Fig. 9.

A.8 Error Bar Analysis

In our experiments, the primary evaluation metric utilized is *Pass@k*. According to the formula, our calculations confirm that the *Pass@k* is unbiased, as discussed in [56].

Moreover, we provide confidence intervals for estimating the theoretical value p using the experimental values of the design success rate (Pass@p), where $n = 15, 30$ represents the sample sizes used in our experiments, and the confidence level $CI = 90\%$. Since the experiments are conducted n times independently, the number of successful designs, c, can be regarded as following a binomial distribution, donated as $c \sim \text{Bin}(n, p)$, where $P(c = k) = \binom{n}{k} p^k (1 - p)^{n-k}$. Therefore, we can employ the Wilson score interval [63] to estimate the confidence interval. The results can be seen in Fig. 10. The results indicate that increasing the number of sampling iterations can reduce the width of the confidence interval to a certain extent. Moreover, smaller values of *Pass@1* tend to underestimate the theoretical value of p; conversely, larger values of *Pass@1* are likely to overestimate the theoretical p. The number of trials in our experiment was determined by considering factors such as financial costs and computational resources.
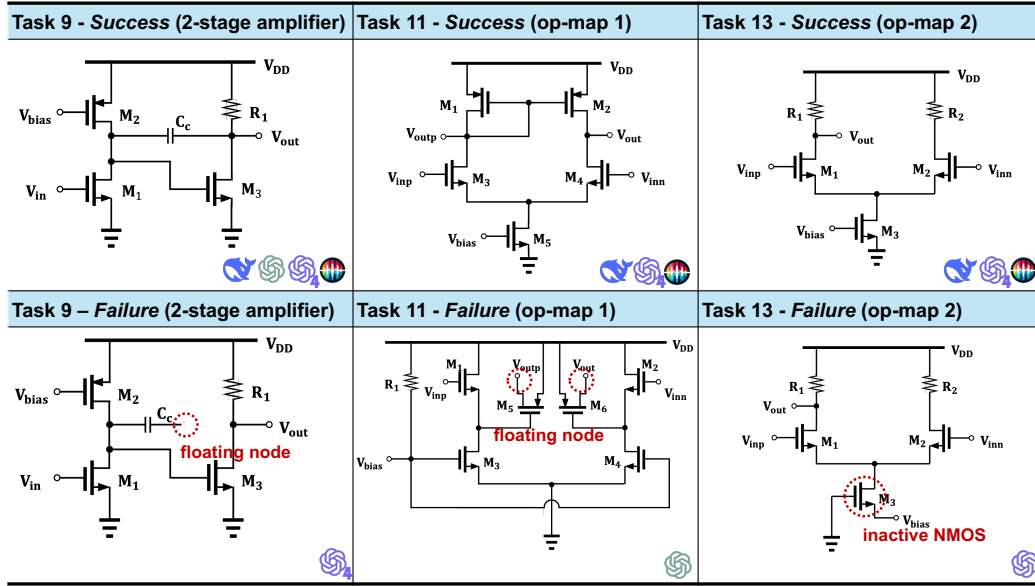
Figure 8: **Additional visualization.** The LLM model used to design this circuit is detailed in the lower right corner.
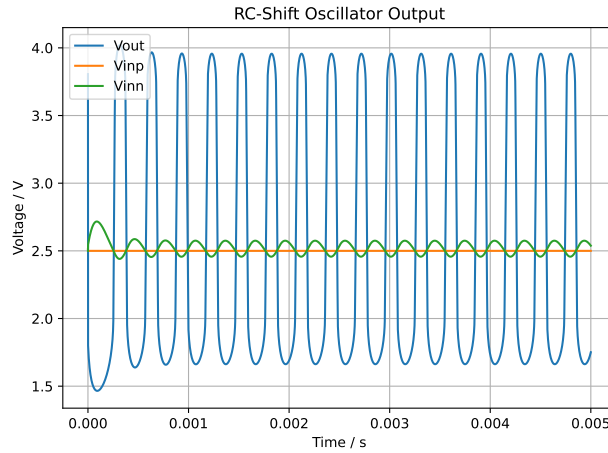

Figure 9: **RC Shift Oscillator Simluation Result.** The simulation results of the RC shift oscillator, derived from a successful design by AnalogCoder, are shown in Fig. 8.

Table 6: **Function Correctness Criteria**

Type	Criteria for Functional Correctness
Amplifier	Gain $A_v > 0$; Drain Current $I_D > 0$
CurrentMirror	Drain Current $I_D > 0$; Adjust the load resistor R from 100Ω to $1k\Omega$, $\delta I_D < 1 \times 10^{-5}$ for at least one interval.
Inverter	$V_{out} \leq 2.5$ when $V_{in} = 0$; $V_{out} \geq 2.5$ when $V_{in} = V_{dd}$; $\delta V_{out} \leq 1.0V$.
Opamp	Drain Current $I_D > 0$; Differential-mode gain $A_{DM} > 0$ when $f = 100$Hz; Differential-mode gain greater than common-mode gain $A_{DM} > A_{CM}$ when $f = 100$Hz.
Oscillator	Within 10 ms analysis, Number of oscillation peaks: $N > 3$; Oscillation amplitude: $A > 1 \times 10^{-6}$; Oscillation period variability: $\frac{\Delta T}{T} \leq 20\%$.
Integrator	Upon inputting a square wave, ensure that the slope k of the resulting triangular wave satisfies $\|k - k'\|/k' \leq 0.3$, where k' is derived from the standard RC time constant; The linear fit of the triangular wave's rising edge should have $R^2 > 0.9$; The number of peaks in the output triangular wave > 2; The circuit configuration must not constitute a passive integrator.
Differentiator	Upon inputting a triangular wave, ensure that the number of peaks $N > 0$; The symmetry of the square wave's peaks and troughs relative to the base voltage V_0 should be maintained, i.e., if V_{peak} and V_{trough} represent the peak and trough voltages respectively, their deviations from V_0 should satisfy $\|V_{peak} - V_0\| = \|V_0 - V_{trough}\|$; The fidelity of the square waveform should be such that the measured peak and trough voltages reach at least 90% of their expected values; The circuit configuration must not constitute a passive differentiator.
Adder	Upon sweeping V_{in1} from the base voltage V_0 to $V_0 + 0.5V$, ensure that the error between the output voltage V_{out} and the negative sum of the inputs $-(V_{in1} + V_{in2})$ remains within 20%, formally, the error ϵ is defined as: $$\epsilon = \left\| \frac{V_{out} + (V_{in1} + V_{in2})}{V_{in1} + V_{in2}} \right\| \leq 0.2$$
Subtractor	Upon sweeping V_{in1} from $2V_0 - 2.25$ to $2V_0 - 1.75$, with V_{in2} set at $2V_0$ (V_0 is bias voltage), ensure that the error between the output voltage V_{out} and the difference between V_{in2} and V_{in1} ($V_{in2} - V_{in1}$) remains within 20%, formally, the error ϵ is defined as: $$\epsilon = \left\| \frac{V_{out} - (V_{in2} - V_{in1})}{V_{in2} - V_{in1}} \right\| \leq 0.2$$
Schmitt Trigger	Ensure V_{out} crosses $V_{dd}/2$ at least once when V_{in} is swept from 0 to V_{dd} and back to 0; The difference in V_{in} at which V_{out} reaches $V_{dd}/2$ during the upward and downward sweeps should exceed 0.05V, formally $\|V_{in,high} - V_{in,low}\| > 0.05$; V_{out} should be a monotonic function of V_{in} throughout each sweep.
VCO	Applying V_{in} of 0.7V, 0.8V, and 0.85V, measure the corresponding output periods T_{out} of V_{out}, denoted as $T_{0.7}$, $T_{0.8}$, and $T_{0.85}$, and satisfying either $\|T_{0.7} - T_{0.8}\| > \epsilon$ and $\|T_{0.8} - T_{0.85}\| > \epsilon$ and $\|T_{0.7} - T_{0.85}\| > \epsilon$., where $\epsilon = 10^{-6}$.
PLL	Apply a 10 MHz clock to CLK_{ref} and observe whether the output frequency of CLK_p deviates from 10 MHz by no more than 5%: $$\left\| \frac{f_{CLK_p} - f_{CLK_{ref}}}{f_{CLK_{ref}}} \right\| \leq 0.05$$

Table 7: **Circuit Tool Library Overview**

Task Id	Type	Gain (dB)	CMG (dB)[*]	# of inputs	# of outputs	Input to V_{out}/I_{out} Phase
1	Amplifier	13.98	NA	1	1	inverting
2	Amplifier	44.13	NA	1	1	inverting
3	Amplifier	-1.58	NA	1	1	non-inverting
4	Amplifier	13.98	NA	1	1	non-inverting
5	Amplifier	13.98	NA	1	1	inverting
6	Inverter	NA	NA	1	1	NA
7	Inverter	NA	NA	1	1	NA
8	Current Mirror	NA	NA	1	1	NA
9	Amplifier	75.94	NA	1	1	-90 degree
10	Amplifier	6.02	NA	1	1	inverting
11	Opamp	193.98	-173.7	2	2	non-inverting, inverting
12	Current Mirror	NA	NA	1	1	NA
13	Opamp	13.98	-163.29	2	1	non-inverting, inverting
14	Opamp	-37.06	-54.72	2	2	non-inverting, inverting
15	Opamp	-28.15	-72.72	2	2	non-inverting, inverting

[*]Common Mode Gain (CMG). The ratio by which an amplifier increases identical input signals, ideally zero in op-amps.

Table 8: **Details of Feedback-Enhanced Flow**

Stage	Check Process	Simulation
Requirement Check	1. Check the presence of required input and output nodes. 2. Check whether the circuit meets basic requirements (e.g., the Vin of a common-gate amplifier should be connected at the source level).	-
Simulation & Operating Point Check	1. Check that no errors occur during the simulation, *e.g.,* floating nodes. 2. Check each MOSFET is active: Vgs>Vth; Vds> Vgs-Vth.	OP
DC Sweep Check	1. Check whether the output varies when the input changes from 0 to Vdd. 2. Substitute the original input voltage with the Vin that produces an output closest to Vdd/2.	DC
Requirement Check	1. Conduct functional checks corresponding to the types of circuits. (See Table 6)	AC/DC/Transient

Table 9: **Fine-tuning task grouping.**

Group	A	B	C
Task IDs	1, 2, 7, 10, 11	3, 5, 8, 14	4, 6, 9, 13

Table 10: **Additional model results.**

Model / Task Id	Mixtral-8×7B Pass@1	Pass@5	CodeLlama-7B Pass@1	Pass@5	CodeLlama-13B Pass@1	Pass@5	CodeLlama-34B Pass@1	Pass@5	Llama 2-70B Pass@1	Pass@5	QwenCode-7B Pass@1	Pass@5	DeepSeek-Coder-33B Pass@1	Pass@5	Codestral-22B Pass@1	Pass@5
1	6.7	33.3	23.3	76.4	0.0	0.0	23.3	76.4	76.7	100.0	0.0	0.0	60.0	99.4	93.3	100.0
2	0.0	0.0	0.0	0.0	0.0	0.0	13.3	53.8	0.0	0.0	6.7	33.3	10.0	43.3	56.7	99.1
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	46.7	96.9
4	6.7	33.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	43.3	95.7
5	0.0	0.0	0.0	0.0	10.0	43.3	3.3	16.7	0.0	0.0	6.7	33.3	0.0	0.0	13.3	53.8
6	86.7	100.0	13.3	53.8	3.3	16.7	0.0	0.0	10.0	43.3	6.7	33.3	0.0	0.0	73.3	100.0
7	20.0	73.6	6.7	31.0	0.0	0.0	36.7	91.8	0.0	0.0	6.7	33.3	6.7	31.0	53.3	98.6
8	13.3	57.1	13.3	53.8	0.0	0.0	6.7	31.0	0.0	0.0	0.0	0.0	20.0	70.2	0.0	0.0
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	13.3	53.8
11-24	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Avg	5.6	12.4	2.4	9.0	0.6	2.5	1.9	7.4	5.1	9.8	1.1	5.6	4.0	10.2	16.4	29.1
# Solved	5	5	4	4	2	2	4	4	3	3	4	4	4	4	8	8

Table 11: **Comparasion between GPT-4 and GPT-4o.**

| Model	GPT-4 (w/ tool lib.)		AnalogCoder	
Task ID	Pass@1	Pass@5	Pass@1	Pass@5
1	**100.0**	**100.0**	**100.0**	**100.0**
2	**100.0**	**100.0**	**100.0**	**100.0**
3	**100.0**	**100.0**	**100.0**	**100.0**
4	**100.0**	**100.0**	**100.0**	**100.0**
5	**100.0**	**100.0**	**100.0**	**100.0**
6	**100.0**	**100.0**	**100.0**	**100.0**
7	**100.0**	**100.0**	**100.0**	**100.0**
8	**100.0**	**100.0**	**100.0**	**100.0**
9	73.3	**100.0**	**100.0**	**100.0**
10	**100.0**	**100.0**	**100.0**	**100.0**
11	66.7	**100.0**	**100.0**	**100.0**
12	0.0	0.0	**13.3**	**57.1**
13	73.3	**100.0**	**100.0**	**100.0**
14	**86.7**	**100.0**	73.3	**100.0**
15	**26.7**	**84.6**	13.3	57.1
16	**60.0**	**99.8**	6.7	33.3
17	0.0	0.0	0.0	0.0
18	60.0	99.8	**100.0**	**100.0**
19	40.0	95.8	**60.0**	**99.8**
20	80.0	**100.0**	**100.0**	**100.0**
21	**26.7**	**84.6**	20.0	73.6
22-24	0.0	0.0	0.0	0.0
Avg	62.2	**77.7**	**66.1**	75.9
# Solved	19	19	**20**	**20**
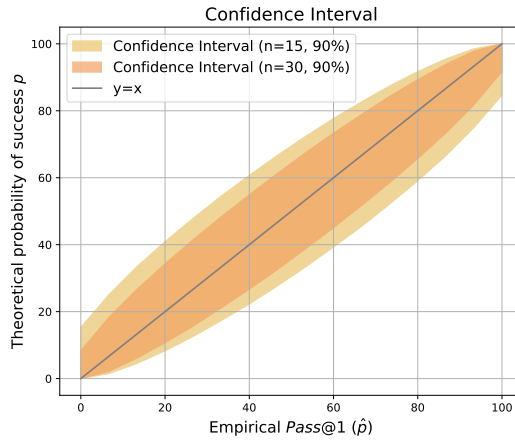
Figure 10: **Confidence Interval Estimation.**

B Related Work

LLM for Code Generation. Analogous to natural language, code generation can similarly be construed as a specific sequence generation task, one that can be effectively learned through language models. These models, trained on vast amounts of code data, have demonstrated remarkable capabilities in understanding and generating code across various programming languages [56, 59, 55, 41]. However, one of the challenges that persist in this domain is the imbalance in the representation of different programming languages and domains within the training data. According to Guo et al. [20], popular languages like Python, Java, JavaScript, and C account for around 60% dataset. This imbalance can lead to suboptimal performance when LLMs are tasked with generating code for underrepresented domains like robot control systems or circuit design. To address this challenge, researchers have proposed the inclusion of domain-specific code samples during the training process. Ahn et al. [64] incorporates robotics control codes into the training data. Liu et al. [18] domain-adaptive pre-trained the Llama2 model with Nvidia's internal design code data, including Verilog and VHDL [65]. Instead of training LLMs inflexibly, our work proposes a training-free, agent-based framework to enhance the circuit design capabilities of LLMs.

LLM for Electronic Design Automation. Advancements in Electronic Design Automation (EDA) technologies have driven progress in the semiconductor industry [66–68], effectively assisting in chip design [6, 69–72]. The main applications facilitated by LLMs include assistance chatbots [18, 73], HDL and script code generation [7, 8, 11, 9, 10, 12, 13, 15, 16], and code verification and analysis [19, 14]. Assistant chatbots can help engineers with different hardware tasks by employing extensive hardware-related data for pre-training or fine-tuning. However, these models have not been made open source now due to the training involving proprietary data. HDL and script code generation is currently one of the most extensively studied directions. In the early stages of LLMs, due to the limited capabilities, most methods required interaction with human experts [7, 8]. With the advent of more advanced LLMs such as GPT-4, researchers have begun to explore the fully automated generation of HDL or script code [13, 9, 11]. Some works have also enhanced the Verilog code generation capabilities by fine-tuning based on existing design data [15] or by employing techniques such as Generative Discriminators [16], and Monte Carlo Tree Search [74]. To provide a fair test dataset, researchers have introduced VerilogEval [12] and RTLLM [13], open-source datasets aimed at assessing the capabilities in Verilog Code. For code verification and analysis tasks, AssertLLM [75] and SpecLLM [76] use LLM for generating design specifications from natural languages. HDLdebugger [19] and RTLFixer [14] focus on the auto-fix error Verilog codes. However, these approaches remain applicable solely to digital circuits, while designing analog circuit components on chips continues to necessitate manual intervention.