pyngspice: A High-performance Python Binding for Ngspice

Jihyeon Park
Department of Electronic Engineering
Hanyang University
Seoul, Korea
falon18@hanyang.ac.kr

Mintae Kim
Department of Electronic Engineering
Hanyang University
Seoul, Korea
alsxo326@hanyang.ac.kr

Jaeduk Han
Department of Electronic Engineering
Hanyang University
Seoul, Korea
jdhan@hanyang.ac.kr

***Abstract*—This paper introduces pyngspice, a Python binding for ngspice that overcomes the performance limitations of existing solutions. Implemented as a Python C extension, pyngspice achieves near-native execution speeds while integrating seamlessly with Python's ecosystem, including libraries such as NumPy and pandas. Benchmarks show significant improvements, with pyngspice reducing simulation time by approximately 50% for simple circuits compared to PySpice, and reducing overall training time in reinforcement learning applications by more than 20%, and nearly 50% compared to the ngspice executable method. These results highlight the effectiveness of the pyngspice framework for time-sensitive simulations and its potential to bridge the gap between high-performance circuit simulations and modern data-driven workflows.**

Keywords—SPICE, circuit simulation, ngspice, pyngspice, Python, reinforcement learning

I. Introduction

Circuit simulations play a pivotal role in the design and verification of electronic circuits. By enabling designers to model and analyze circuit behavior prior to physical fabrication, simulations help in identifying critical issues early in the design process, thereby reducing development costs and time-to-market. Among the various tools available for circuit simulation, SPICE (Simulation Program with Integrated Circuit Emphasis) simulators have been the standard integrated circuit simulator for decades [1]. Ngspice, an open-source descendant of SPICE, has become particularly popular due to its flexibility, comprehensive feature set, and the active support it receives from the community. Ngspice is widely used in both academia and industry, facilitating a range of activities from educational exercises to advanced research and industrial design projects.

Despite ngspice's extensive capabilities and widespread adoption, there have been continuing demands on integrating it with modern programming environments such as Python. Python has become the de facto language for data analysis, machine learning, and automation, owing to its simplicity, readability, and the vast ecosystem of libraries it offers. For electronic design automation (EDA), where tasks often involve complex data processing, iterative simulations, and optimization routines, Python's strengths are particularly valuable for enhancing productivity.

There are two primary approaches to integrating ngspice with Python: the shared library method and the executable file method. The shared library method involves interfacing Python directly with ngspice's shared library using mechanisms such as Python's ctypes or CFFI (C Foreign Function Interface) libraries. This approach allows Python code to call ngspice functions directly, offering fine-grained control over simulations and enabling seamless integration with Python's data processing libraries. However, this method introduces overhead due to the constant switching between Python and C, which can lead to performance bottlenecks, especially in applications requiring rapid or repeated simulations.

On the other hand, the executable file method involves invoking the ngspice executable from within Python scripts. This approach typically uses Python's subprocess module to run ngspice as an external process, with input and output handled via files or standard input/output streams. While this method is simpler to implement and avoids some of the performance issues associated with shared libraries, it suffers from its own drawbacks. The overhead of process creation and stream I/O can be substantial, and the need to parse and interpret ngspice's output in Python further complicates the workflow. Additionally, the executable file method lacks the tight integration with Python's ecosystem, making it less suitable for complex data processing or automation tasks.

Both of these approaches have been implemented in various Python libraries to integrate ngspice. PySpice [2] and ngspicepy [3] use the shared library method, which offer a more Pythonic interface and better integration with Python's data handling capabilities, but at the cost of increased overhead and slower performance in simulation-heavy tasks. Conversely, spicelib [4] relies on the executable file method and provides a more straightforward implementation with lower setup complexity, but often at the expense of flexibility and execution speed.

Given these limitations, there is a clear demand for a more efficient and integrated solution to using ngspice within Python. This paper introduces pyngspice, a Python binding for ngspice that is designed to address the performance issues associated with existing solutions. Pyngspice is implemented as a Python C extension, which allows it to interface directly with ngspice at a low level, thereby minimizing overhead and achieving near-native execution speeds. This approach not only preserves the full functionality of ngspice but also allows users to leverage Python's extensive ecosystem without sacrificing performance.

The objective of this research is to demonstrate that pyngspice provides a significant performance improvement over existing tools, making it a more suitable choice for time-sensitive simulation tasks. The following sections of this paper will detail the design and implementation of pyngspice, provide benchmark comparisons with PySpice and ngspice executable method, and explore its application in reinforcement learning for circuit optimization. By achieving

pyngspice is available at: https://github.com/LeunPark/pyngspice a seamless integration of ngspice with Python, pyngspice aims to bridge the gap between high-performance circuit simulation and modern data-driven design workflows.

II. Pyngspice

A. Design Principles

The development of pyngspice is anchored in three key design principles: performance, compatibility, and extensibility. These principles guide the architecture of the tool to ensure it meets the demands of modern circuit simulation tasks, both in terms of speed and integration with existing Python-based workflows.

- Performance: Pyngspice is implemented as a Python C extension, allowing it to directly interface with the ngspice shared library without the overhead introduced by intermediate libraries like ctypes. This direct interaction enables pyngspice to achieve near-native execution speeds, which is critical for both small-scale simulations. Additionally, pyngspice employs several low-level optimizations, including the inlining of frequently called functions and minimizing data copying between Python and C. These optimizations enhance its performance, reducing latency and making it ideal for computationally intensive tasks.

- Compatibility: Pyngspice is designed to function seamlessly across multiple operating systems, including Linux, macOS, and Windows. It automatically detects and links the appropriate ngspice shared libraries and code models for each environment, ensuring smooth installation and operation regardless of the user's platform. This cross-platform compatibility ensures that pyngspice can be easily adopted by a wide range of users.

- Extensibility: Finally, pyngspice is built with extensibility in mind. The tool is intended to serve as a drop-in replacement for PySpice, allowing users to transition effortlessly to pyngspice without rewriting their existing codebases. Furthermore, pyngspice integrates smoothly with popular Python libraries like NumPy and pandas, enabling users to leverage powerful data processing and analysis tools within their circuit simulation workflows. This makes pyngspice a flexible and future-proof tool, capable of evolving with the growing needs of electronic design automation (EDA) tasks.

B. Integration with Existing Tools

One of the challenges in developing pyngspice was ensuring that it integrates smoothly with the broader Python ecosystem. To this end, pyngspice includes a comprehensive API that mirrors the functionality of ngspice while adhering to Python's design conventions. This API supports all major ngspice features, including loading circuits, running analyses, and supporting various commands.

Additionally, pyngspice is fully compatible with PySpice, allowing users to retain PySpice's functionalities while utilizing pyngspice for running simulations. For instance, users can switch from PySpice to pyngspice with just a small modification to the import statement, as demonstrated below:

```
# from
from PySpice.Spice.Netlist import Circuit
# to
from pyngspice.pyspice import Circuit
```

This simple adjustment ensures that existing PySpice code can be used with pyngspice, allowing users to benefit from the performance improvements without needing to rewrite their simulations.

C. Enhanced Data Processing

One of the key advantages of using Python in circuit simulation is its powerful data processing capabilities. Pyngspice leverages this by providing native support for NumPy, allowing users to perform complex numerical operations on simulation data with minimal overhead. For example, users can apply NumPy's vectorized operations to large datasets, enabling efficient filtering, transformation, and analysis of simulation results.

In addition to NumPy, pyngspice also supports integration with pandas, which is widely used for data manipulation and analysis in Python. This allows users to store and manipulate simulation results as DataFrames, facilitating more sophisticated data analysis workflows. For instance, users can easily compare the results of multiple simulations, perform statistical analysis, or visualize data trends directly within Python.

III. Benchmarks

A. Methodology

To thoroughly evaluate the performance of pyngspice, we conducted a series of benchmarks comparing it against both PySpice and the ngspice executable method in Python. The benchmarks measured performance at three key stages: initialization, execution, and getting plots.

- The *initialization* phase focused on the time and resources needed to set up the simulation environment and load circuits across each library.

- The *execution* phase evaluated the actual simulation time and resource usage for each circuit, providing insights into how each tool handles the computational load of the simulation process.

- Finally, the *getting plots* stage measured the efficiency of retrieving simulation results, a crucial aspect for real-time analysis and data-driven workflows.

Once the stage-based analysis framework was established, we applied it to a series of benchmark circuits. These circuits ranged from simple designs, like a single-stage amplifier (frequency response) and a level shifter (DC transfer curve), to a more complex circuit, C3540, derived from the ISCAS85 benchmark suite [5]. This selection ensured that the tests covered a wide range of real-world use cases, from basic analog components to intricate digital systems.

Each circuit was simulated under identical conditions using pyngspice, PySpice, and the ngspice executable. We measured key performance metrics, including simulation time and peak memory usage (consumed by the Python process),

TABLE I. PERFORMANCE COMPARISON OF THE INITIALIZATION

Library	***Elapsed Time***	***Peak Memory***
Executable	-	-
PySpice	49.336	1.570
pyngspice	2.595	0.000

TABLE II. PERFORMANCE COMPARISON OF THE EXECUTION AND GETTING PLOTS OF THE CIRCUITS

Circuit	**Library**	***Execution***		***Getting Plots***	
		Elapsed Time	***Peak Memory***	***Elapsed Time***	***Peak Memory***
Single-stage Amplifier	Executable	20.507	-	44.143	-
	PySpice	15.577	0.002	0.437	0.368
	pyngspice	8.747	0.000	0.247	0.367
Level shifter	Executable	6.553	-	1.552	-
	PySpice	1.748	0.004	0.197	0.004
	pyngspice	1.026	0.000	0.039	0.001
C3540	Executable	111303.473	-	76.767	-
	PySpice	106813.294	0.639	854.712	267.947
	pyngspice	106606.488	0.000	91.860	8.965

[a] Elapsed Time (ms) and Peak Memory Usage (MB)

across the different stages. Benchmarks were conducted on a NHN Cloud Compute Instance (m2.c8m16), configured with 8 vCPUs, 16 GB of RAM, and running Ubuntu 20.04 LTS. All simulations were repeated 500 times to ensure statistical significance, with results averaged to reduce the effect of transient system variations.

B. *Performance Results*

The benchmark results in Table 1 and 2 clearly demonstrate the performance advantages of pyngspice over PySpice. In simple circuits, such as a single-stage amplifier and level shifter, pyngspice consistently outperformed PySpice by reducing simulation time by approximately 50%. For more complex circuits, such as those based on the ISCAS85 benchmarks, the performance gap between pyngspice and PySpice narrows, but pyngspice still maintains a slight edge, delivering marginally faster results in nearly every case. While the difference is less pronounced, pyngspice continues to show subtle yet consistent improvements in efficiency.

Additionally, the executable method does not have a separate initialization stage, as initialization is integrated into the execution phase. As a result, the total execution time for the executable method is longer compared to pyngspice and PySpice, where initialization is a distinct phase.

Peak memory usage was also measured to assess the overall resource efficiency of pyngspice. The results show that pyngspice has a lower memory footprint compared to PySpice, which can be attributed to the optimized data handling.

C. *Implications of Performance Gains*

The performance improvements achieved by pyngspice have significant implications for circuit design workflows. In scenarios where simulations need to be run frequently, such as during reinforcement learning or optimization tasks, the time savings provided by pyngspice can be substantial. For instance, in a setup where millions of parametric sweeps are required to optimize a circuit design or evaluate its robustness under different conditions, a 50% reduction in simulation time can translate to hours or even days of saved computation time.

IV. APPLICATION IN REINFORCEMENT LEARNING

A. *Reinforcement Learning Integration*

Reinforcement learning (RL) has emerged as a powerful tool for optimizing circuit designs, particularly in situations where traditional design methodologies are insufficient [6, 7]. In an RL framework, an agent iteratively improves a circuit design by interacting with the simulation environment and receiving feedback in the form of a reward signal. This process often requires running thousands or even millions of simulations to explore the design space and converge on an optimal solution.

pyngspice is ideally suited for integration into RL workflows due to its fast simulation capabilities and efficient data handling. By reducing the time required for each simulation, pyngspice enables RL algorithms to explore the

TABLE III. PERFORMANCE COMPARISON OF THE RL TRAINING

Library	***Elapsed Time (sec)***
Executable	1947.69
PySpice	1265.54
pyngspice	983.46

design space more quickly, leading to faster convergence on optimal designs.

B. *Case Study: AutoCkt*

To demonstrate the practical benefits of pyngspice in an RL context, we conducted a case study using AutoCkt [6], a tool that automates the design of analog circuits through RL. In this study, we compared the performance of pyngspice, PySpice, and the native ngspice executable in optimizing a common-source amplifier circuit. The experiment was configured using Ray, with 200 iterations and 6 workers for parallel processing.

The RL algorithm was tasked with optimizing the amplifier's gain and bandwidth by adjusting component values such as transconductances and capacitances. The optimization process involved running the circuit simulation thousands of times to evaluate different design configurations.

As expected from the benchmark results discussed in the previous section, pyngspice still outperformed the others during the reinforcement learning process, though the performance gap was smaller. As shown in Table 3, compared to PySpice, pyngspice reduced the overall training time by more than 20%, and by nearly 50% when compared to the executable method, allowing the RL algorithm to converge on an optimal solution more quickly. The reduced performance gain can be attributed to the additional computational overhead inherent in the RL environment. Factors such as the processing time of the RL algorithm itself, the communication between the RL agent and the simulator, and other non-simulation-related computations consume a significant portion of the total runtime, thereby lessening the relative impact of the simulation speed improvements provided by pyngspice.

This case study highlights the potential of pyngspice to accelerate RL-driven circuit design, making it a valuable tool for both researchers and practitioners in the field of electronic design automation.

V. CONCLUSION

This paper introduces pyngspice, a high-performance Python binding for ngspice, designed to overcome the limitations of existing Python-SPICE integration tools. pyngspice achieves significant performance improvements over PySpice while maintaining compatibility with existing workflows. The benchmarks and case studies demonstrate pyngspice's ability to accelerate circuit simulation and optimization tasks, particularly in reinforcement learning applications.

REFERENCES

[1] L. W. Nagel, "The 40th Anniversary of SPICE: An IEEE Milestone [Guest Editorial]," in *IEEE Solid-State Circuits Magazine*, vol. 3, no. 2, pp. 7-82, Spr. 2011,

[2] Salvaire, F., "PySpice: Simulate electronic circuits using Python and NgSpice," https://github.com/PySpice-org/PySpice (accessed Sep. 1, 2024).

[3] Rego, A. J., "ngspicepy: Python wrapper for ngspice," https://github.com/ashwith/ngspicepy (accessed Sep. 1, 2024).

[4] Brum, N. "spicelib: A Python library for electronic circuit simulation using ngspice," https://github.com/nunobrum/spicelib (accessed Sep. 1, 2024).

[5] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a targeted translator in FORTRAN," *Special Session on ATPG and Fault Simulation, Proc. 1985 IEEE Int. Symp. on Circuits and Systems (ISCAS'85)*, June 5-7, 1985.

[6] K. Settaluri, A. Haj-Ali, Q. Huang, K. Hakhamaneshi and B. Nikolic, "AutoCkt: Deep Reinforcement Learning of Analog Circuit Designs," *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 490-495, Mar. 2024.

[7] S. Hong et al., "Analog Circuit Design Automation via Sequential RL Agents and Gm/ID Methodology," in *IEEE Access*, vol. 12, pp. 104473-104489, 2024.