Journal of Advances in Developmental Research (IJAIDR)
E-ISSN: 0976-4844 ● Website: www.ijaidr.com ● Email: editor@ijaidr.com

A Spreadsheet-Driven, Open-Source Framework for Simulating Power Electronic Circuits in Python

M Rekha[1], P Srividya Devi[2].

[1]Assistant Professor, [2]Associate Professor,
Department of Electrical and Electronics Engineering, GRIET

Abstract

Circuit-level simulation plays a key role in the design and verification of power electronic systems, but many of the tools that engineers use for this purpose are either proprietary or require a considerable amount of specialist training to be used effectively. This paper examines Python Power Electronics (PPE), a lightweight, free, and open-source simulation environment written entirely in Python that targets exactly this gap. Rather than relying on a graphical schematic editor, PPE lets a user lay out a circuit inside an ordinary spreadsheet, from which the software automatically derives the node, branch and loop structure needed for analysis. The behaviour of the components, their connections, and the control logic are represented by simple and well-defined data structures. The user-written Python control functions can be linked to the circuit via a spreadsheet that specifies how the functions' inputs and outputs are mapped into the simulation variables. The internal representation of the components is explained as the two-stage (loop and nodal) analysis process used to advance the simulation in time, the method by which control functions are scheduled against arbitrary time events, and the various output formats that the tool provides. To show the type of results the simulator can produce, some typical waveforms, comprising a carrier and a PWM reference, diode and source currents, and the converter output voltage are given. This paper concludes with an assessment of the benefits of the tool's Python usage and the places where it currently impedes performance, along with a mention of the upcoming integration with NumPy and SciPy for signal post-processing.

Keywords: Open-source simulation, power electronics, nonlinear circuit analysis, Python, spreadsheet-based schematic capture, control function interfacing, PWM simulation

1. Introduction

Circuits built from power semiconductor devices sit at the heart of applications ranging from motor drives to renewable-energy interfacing and power-quality correction [8]. Because these circuits switch and therefore behave nonlinearly, verifying a design almost always means simulating it before it is built, typically through some form of modified nodal or loop analysis solved at each time step [9]. Commercial simulation packages can do this well, but licensing costs and steep learning curves put them out of reach

for many students, hobbyists and small research groups, and their closed internals make it difficult to see — let alone modify — how the underlying numerical methods work.

Python Power Electronics (PPE) was created to lower that barrier [1]–[3]. It is written entirely in the Python programming language, released under an open-source licence, and designed so that a circuit can be described and simulated without first learning a dedicated schematic-capture tool [1]. The premise is straightforward: if a language is expressive enough to build complex data structures quickly, and if it already has a mature ecosystem for scientific computing [4], [5], it is a reasonable foundation for a circuit solver — even one intended to handle large systems containing several converters.

The rest of this paper describes PPE from the viewpoint of someone setting it up for the first time, including how the simulator is set up internally, how a circuit is entered and represented, how components and control code are modelled as data, how the time-stepped solution actually works, and what the output looks like after a simulation has run. Finally, we briefly explore the trade-offs involved in developing a circuit solver in a high-level language like Python.

2. Overview of the Simulator

PPE grew out of a fairly narrow goal, simulating a handful of passive linear circuits and its expanded release as new component types, converter topologies and bug fixes were added [1]. Its scope now extends to systems with several interacting power converters, and its development has increasingly been aimed at circuits representative of wind- and solar-generation interfacing, including a shunt-connected three-phase VAR compensator used as one of its test cases [1].

Two properties have guided the project from the start. The first is portability: because the simulator is pure Python and stores circuit data in spreadsheet files, a simulation set up on one operating system runs unmodified on another. The second is accessibility: the source is free to inspect, extend and redistribute, which has allowed the tool to be corrected and extended incrementally as issues surface in use [2]. Fig 1 shows the landing page of the Python Power Electronics website . Documentation, downloads and worked case studies are maintained on a project website [3], and an early public release was distributed through SourceForge [2].

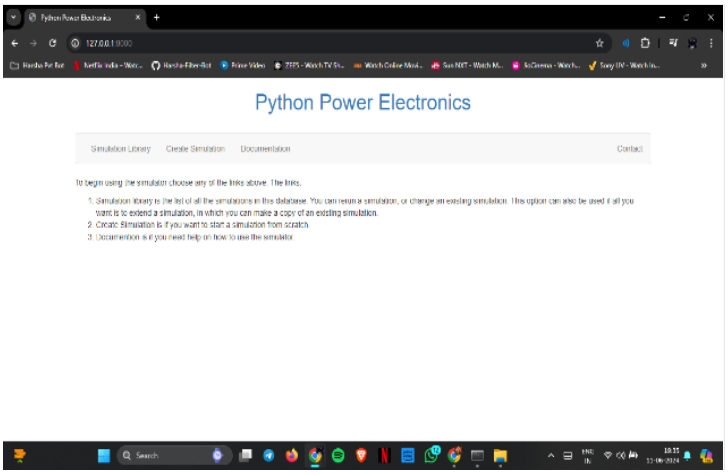

Fig. 1. Landing page of the Python Power Electronics project website.

3. Python as the Implementation Language

Python was chosen primarily because it makes it cheap to build and rearrange the data structures a circuit solver needs. Circuit analysis requires tracking branch maps, loop maps and related bookkeeping that is updated after every iteration as the simulation state changes; Python's native support for heterogeneous lists — a single list that freely mixes numbers, strings and even other lists — turns out to be a convenient way to hold this kind of information without committing to a rigid schema in advance.

There is an obvious tension in using a high-level, interpreted language for something that is meant to scale to large circuits: the interpreter overhead itself is a computational cost that a lower-level implementation would not carry. In practice, this has mattered less than expected. Because PPE has no graphical front end, it stays light on system resources, and its structure lends itself naturally to running several independent simulation cases side by side — for instance, sweeping different loads, input conditions or controller settings in parallel — which has more than offset the per-iteration cost of the interpreter for the workloads the tool has been used on so far.

Beyond convenience, Python's scientific-computing ecosystem was a second motivation for the choice. NumPy [4] and SciPy [5] are intended for future integration, primarily to support post-processing tasks such as spectral (Fourier) analysis and wavelet-based signal processing that are presently handled by separate, independently developed utilities. Bringing that functionality into the core simulator is viewed as a prerequisite for PPE to approach the breadth of analysis features found in established commercial tools.

4. Circuit Entry and Representation

Rather than a graphical schematic editor, PPE represents a circuit as a spreadsheet [1]. The user places components onto a grid of cells, and the simulator parses that grid into a matrix in which each entry denotes either a component, a length of wire, a jump label used to route connections that would otherwise cross, or an empty cell. From this matrix the software identifies the circuit's nodes, branches and loops, which are the quantities loop analysis and nodal analysis are subsequently built on [9].

This spreadsheet-based approach was adopted mainly for portability and ease of entry: any spreadsheet application capable of exporting to CSV can be used to build a schematic; simulation parameters and component values are entered in the same familiar format, and a file produced on one platform opens without modification on another. Fig 2 determines the example circuit representation in an Excel sheet. The same representation also scales to circuits containing multiple converters, which was one of the design targets for the tool.

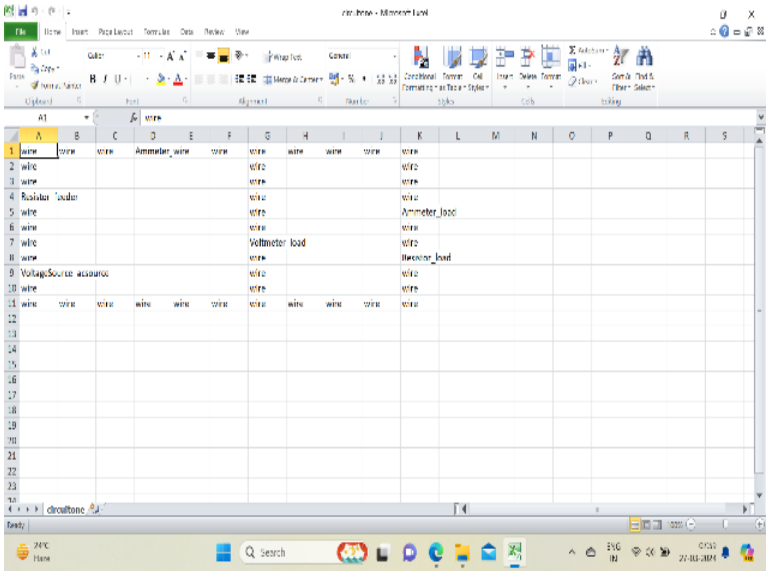

Fig. 2. Example circuit schematic laid out on a spreadsheet grid.

Because both circuit topology and simulation results move in and out of the tool as spreadsheets, the representation doubles as a lightweight interchange format: a circuit can be designed on one machine, handed to a collaborator, and simulated on another without any conversion step.

5. Internal Data Model for Components

Every component recognised by the simulator is represented internally by an object, and the data attached to that object falls into three categories: values generated automatically by the software, values the user enters directly, and values that change as the simulation proceeds. Four attributes are common to every component class: Type, Position, Tag and Index.

Type records what kind of device the object represents, for example, a resistor is tagged with type "Resistor". Position records where the device sits on the spreadsheet, but is stored as coordinates in the underlying circuit matrix rather than in the spreadsheet's own row-and-column notation; a resistor labelled R1 that a user places at cell 4A, for instance, is stored using its matrix coordinates rather than the string "4A". Tag preserves the component's name exactly as it appears on the sheet as "R1" in this example, so that references in control functions and output files remain human-readable. Index is assigned automatically as components are read in; it exists purely to make debugging easier and plays no role in the actual circuit computation.

Advancing a simulation in PPE proceeds in two conceptual stages at every time step. First, the parameters attached to each circuit component are translated into the corresponding branch parameters. Second, those branch parameters feed into the circuit analysis itself, which is split between loop analysis and nodal analysis, in the spirit of the modified-nodal formulation widely used in general-purpose circuit simulators [9].

Loop analysis builds a matrix equation directly from the connectivity extracted from the spreadsheet and solves it with numerical integration to obtain loop and, from those, branch currents. Nodal analysis instead targets branches that are numerically stiff, solving a matrix equation that is treated as fixed until some

event in the circuit — a switch changing state, for example — forces it to be rebuilt, an approach consistent with standard treatments of switched power-converter analysis [8]. The simulator alternates between these two forms of analysis, advancing the simulation clock and refreshing branch data after each pass, until the requested simulation duration has been reached [1].

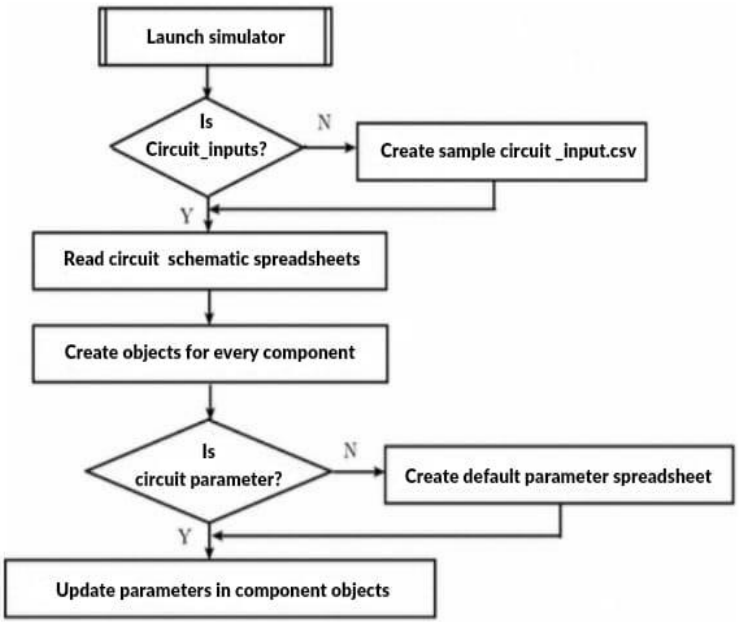

Fig. 3. High-level flow of the simulation launch and solution process.

Running a simulation in practice means invoking the main entry point, circuit_solver.py, with the Python interpreter, typically from a shell such as BASH with the command python circuit_solver.py. Fig 3 represents the high-level flow of the simulation launch and solution process. On startup, the simulator reads circuit_inputs.csv, a spreadsheet holding run-time parameters: the total simulation duration, the integration time step, and the filename to which results should be written. The duration sets the upper time limit for the run, the time step sets how far the clock advances at each iteration, and the output filename is left to the user, subject to whatever naming rules the host operating system imposes.

6. Attaching User-Written Control Functions

Control logic in PPE is written as ordinary Python files and declared in the simulation-parameter spreadsheet, rather than being hard-coded into the solver [1]. Each control function is paired with a descriptor spreadsheet that defines its interface — which simulation variables feed into it as inputs, which variables it writes back out, and a small set of special variables reserved for actions such as forcing a variable update or raising a timed event.

Because the interface is declared separately from the control logic itself, multiple controllers can be wired into a single simulation and connected to each other in a modular way, without editing the solver's core code. A descriptor entry is built from three fields: Input, which marks a row as an input port to the

controller; Meter, which names the circuit component supplying that input; and Access, which lets the user specify how the value should be read.

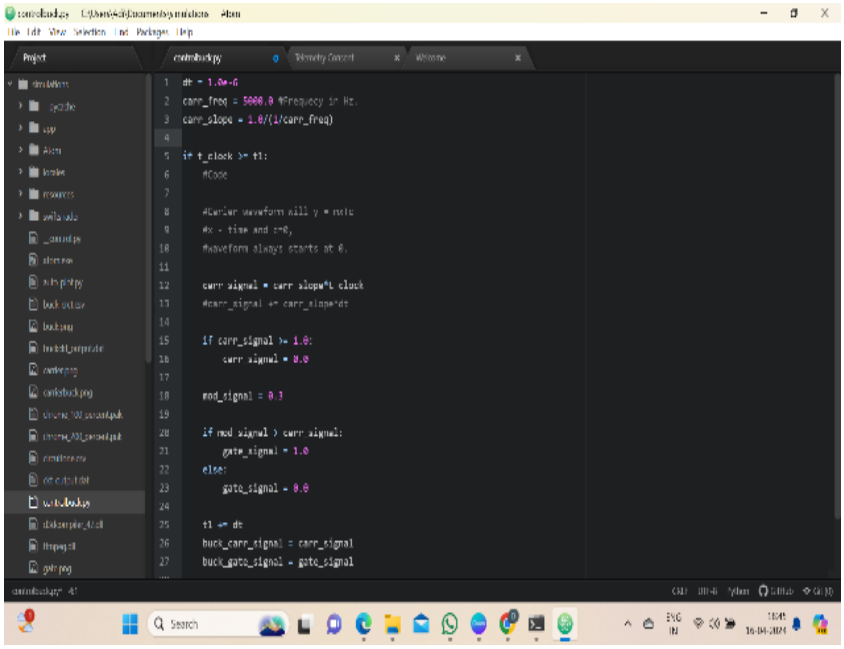

Fig. 4. Descriptor spreadsheet defining a control function's inputs and outputs.

Timing is handled through a general event mechanism rather than being tied to fixed multiples of the simulation time step. Each control function can request execution at a specific time instant or on a recurring period, and these periods need not divide evenly into the master time step or into one another. Fig. 4 shows the descriptor spreadsheet defining a control function's inputs and outputs. This gives the simulator a level of time resolution comparable to what a hardware implementation would offer, and it ensures a control function fires exactly when it was scheduled to, rather than being rounded to the nearest solver step.

7. Results and Discussion

Simulation results are written to a data file whose name is set in circuit_inputs.csv, and from there can be plotted — typically with a Python plotting library such as Matplotlib [7] — to track the progress or correctness of a run. Beyond the default ammeter and voltmeter traces, PPE offers two further conveniences. Large circuits can have their output automatically split into several files, each covering a limited time window, by setting "Split the Output File?" to "Yes" in the parameters sheet — useful when a single run would otherwise produce an unwieldy amount of data. Separately, any variable computed inside a control function — instantaneous power or peak voltage, for example — can be added to the plotted output simply by naming it in that function's descriptor spreadsheet, without further changes to the solver.Fig.5 illustrates the carrier, reference and PWM gating waveforms.Fig.6 illustrates the typical waveforms of diode current, input voltage source waveforms produced by a PPE simulation of a DC-DC converter: the carrier and reference signals used to generate a PWM gating pattern [10], the resulting diode and source currents, and the converter's output voltage [8]. Fig 7 represents the output voltage waveform of the converter.

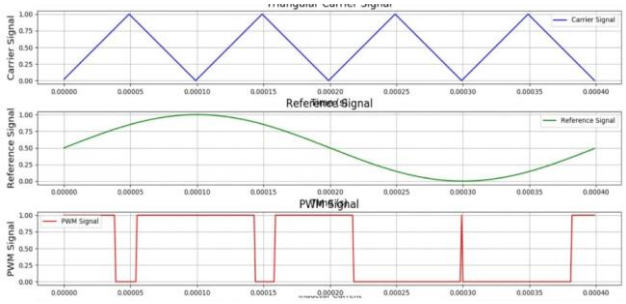

Fig. 5. Carrier, reference and PWM gating waveforms generated during a simulation run.

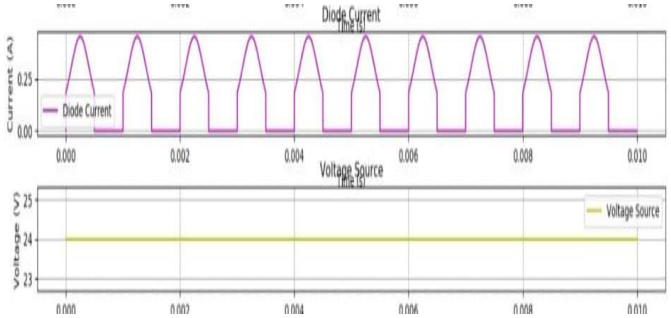

Fig. 6. Diode current and source voltage waveforms from the same run.

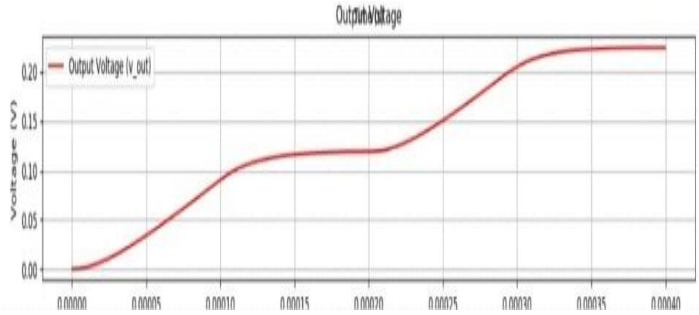

Fig. 7. Converter output voltage produced by Python Power Electronics.

8. Conclusion

Python Power Electronics demonstrates that a genuinely useful nonlinear circuit simulator can be built on an accessible, high-level language and a spreadsheet-based front end, without requiring a dedicated schematic editor. Circuits are entered as spreadsheets and parsed into components represented as simple name-and-position pairs; these are collected into a components-found dictionary keyed by component type, with each value holding the list of that type's instances. Simulation proceeds by alternating loop and nodal analysis, while user-supplied control logic implemented as ordinary Python scripts and connected through a descriptor spreadsheet is scheduled against a flexible, event-driven timing mechanism rather than a fixed step multiple.

The tool's reliance on plain files and a widely available scripting language gives it a portability that heavier, GUI-based packages generally lack: a simulation designed on one machine runs unchanged on another. What remains open is largely a matter of breadth: deeper integration with NumPy and SciPy for post-processing and continued extension to more converter topologies and larger interconnected systems, drawing on established power-electronics analysis methods rather than any limitation in the core approach.

References

1. S. V. Iyer, Simulating Nonlinear Circuits with Python Power Electronics: An Open-Source Simulator, Based on Python. Cham, Switzerland: Springer International Publishing, 2018.
2. Python Power Electronics, open-source project repository, SourceForge. [Online]. Available: https://sourceforge.net/projects/pythonpowerelec/
3. Python Power Electronics project documentation and case studies. [Online]. Available: http://www.pythonpowerelectronics.com
4. C. R. Harris, K. J. Millman, S. J. van der Walt et al., "Array programming with NumPy," Nature, vol. 585, no. 7825, pp. 357–362, 2020.
5. P. Virtanen, R. Gommers, T. E. Oliphant et al., "SciPy 1.0: Fundamental algorithms for scientific computing in Python," Nature Methods, vol. 17, pp. 261–272, 2020.
6. F. Salvaire, PySpice: Simulate electronic circuits using Python and the Ngspice/Xyce simulators. [Online]. Available: https://pyspice.fabrice-salvaire.fr/
7. J. D. Hunter, "Matplotlib: A 2D graphics environment," Computing in Science & Engineering, vol. 9, no. 3, pp. 90–95, 2007.
8. R. W. Erickson and D. Maksimović, Fundamentals of Power Electronics, 2nd ed. Norwell, MA: Kluwer Academic Publishers, 2001.
9. C.-W. Ho, A. E. Ruehli, and P. A. Brennan, "The modified nodal approach to network analysis," IEEE Transactions on Circuits and Systems, vol. 22, no. 6, pp. 504–509, 1975.
10. N. Mohan, T. M. Undeland, and W. P. Robbins, Power Electronics: Converters, Applications, and Design, 3rd ed. Hoboken, NJ: John Wiley & Sons, 2003.