badcrossbar: A Python tool for computing and plotting currents and voltages in passive crossbar arrays

Dovydas Joksas *, Adnan Mehonic

Department of Electronic and Electrical Engineering, University College London, London, United Kingdom

**Abstract**

Crossbar arrays are a popular solution when implementing systems that have array-like architecture. With the recent developments in the field of neuromorphic engineering, crossbars are now routinely used to implement artificial neural networks or, more generally, to perform vector–matrix multiplication in hardware. However, the interconnect resistance present in all crossbars can lead to significant deviations from the intended behaviour of these structures. In this work, we present badcrossbar—an open-source tool for computing currents and voltages in such non-ideal passive crossbar arrays. Additionally, the package allows to easily visualise currents and voltages (or other numerical variables) in the branches and on the nodes of these structures.

© 2020 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

**Introduction**

In the most general definition, crossbar arrays are structures with m inputs and n outputs producing a 2D grid of \( m \times n \) intersections. The inputs are applied to the conductive lines which are referred to as word lines. The outputs are obtained from the conductive lines which are referred to as the bit lines. Devices are placed at the intersections of the word and bit lines, as shown in Fig. 1A.

Even in their early implementations, crossbar array applications ranged from simple control of the flow of signals [1] to performing mathematical operations [2]. The matrix-like arrangement of crossbar arrays’ devices makes them natural candidates for some linear algebra operations. Specifically, crossbar arrays are capable of computing vector–matrix products. This can be achieved by employing analogue two-terminal devices with variable conductance, such as memristors [3].

Fig. 1B shows circuit diagram of a crossbar array (with crossbar devices depicted as memristors). Word and bit line segments bounded by neighbouring nodes (depicted as black dots), like crossbar devices, can have resistance. However, if the resistance of these segments (which we will refer to as interconnects) is negligible, then the output currents (depicted as \( I_1, I_2, \ldots \)) in Fig. 1B of a crossbar array are a product of a vector of applied voltages and a matrix of crossbar devices’ conductances. Crossbar arrays performing this function are usually referred to as dot-product engines (DPEs).

---

* Corresponding author.
E-mail addresses: dovydas.joksas.15@ucl.ac.uk (D. Joksas), adnan.mehonic.09@ucl.ac.uk (A. Mehonic).
With the development of memristive devices [3,4], practical implementations of DPEs became more realisable. Because memristive devices are capable of gradual conductance change [5], memristive crossbar arrays can be used to encode matrix entries. Although conductances can only be positive, it is possible to encode negative numbers by using pairs of memristors [6,7]. Because of the flexibility, DPEs are now often used for in-memory computing, most notably—in the implementation of memristive neural networks [8]. However, such systems are still often limited by the non-ideal nature of crossbar arrays—non-negligible resistance of interconnects can significantly affect the performance.

Computing currents and voltages in crossbar arrays is essential to understanding their behaviour under non-ideal conditions. So naturally, there have been careful analyses of line resistance effects in passive crossbar arrays in the past [9]. However, to the best of our knowledge, there are no open-source context-independent tools available to analyse line resistance effects in crossbar arrays. For example, being able to compute output currents of a crossbar array when multiple sets of inputs are applied would be useful in simulations of memristive neural networks [10]. Computing node voltages or currents in other branches than the output ones would be useful in the analysis of line resistance or sneak-path currents. Serb et al. have examined the issue of accurate readout of the resistive state of devices within selectorless crossbar arrays and concluded that the line resistance is significant even for smaller crossbars (12 × 12) [11]. Advanced mapping or compensation schemes have been proposed to mitigate the issue [12]. Additionally, our simulation tool could supplement the analysis of various effects, such as defects of crossbar devices (and their mitigation), that have been well studied in the past [13–15].

Furthermore, visualising the distribution of currents and voltages in a crossbar array can provide additional insight. Although the devices in crossbar arrays are arranged in a matrix-like fashion, they do not paint the complete picture. In addition to the devices, there are word- and bit-line branches, that all together form 3 × m × n unique branches (see Fig. 1B) through which the currents flow. Also, there are 2 × m × n unique nodes, each of which can be associated with a voltage value. Visualising currents and voltages (or any other numerical variables) in the branches and on the nodes of the crossbar array can help understand the nature of these structures.

2. Problems and background

To set up equations to solve for the currents and voltages in circuits containing only passive elements requires just Kirchhoff’s current law (KCL) and Ohm’s law. In the specific case of crossbar arrays, the relevant equations can be set up easily for arbitrary number of word lines, m, and arbitrary number of bit lines, n, due to simple architecture of these arrays. Most circuit simulators use nodal analysis (or its variants) to analyse circuits and it has been applied to crossbar arrays in the past [9]. However, we will briefly explain this method.

If interconnect resistances are non-zero, then there are 2 × m × n unknown node voltages one has to solve for, as shown in Fig. 2A. One can solve for these voltages by applying KCL to currents (shown in Fig. 2B) flowing into/out of a particular node. For the case of m, n > 1, we can set up KCL equations for a word line node and a bit line node at the intersection of jth word line and jth bit line using Eqs. (1a), b and (1c), d, respectively.

\[ I_{WL(i,j)} - I_{WL(i,j+1)} - I_{BL(i,j)} = 0 \quad \text{for } j < n \quad (1a) \]

\[ I_{WL(i,j)} - I_{BL(i,j)} = 0 \quad \text{for } j = n \quad (1b) \]

\[ I_{BL(i,j)} - I_{WL(i,j)} = 0 \quad \text{for } i = 1 \quad (1c) \]

\[ I_{BL(i,j)} + I_{BL(i-1,j)} - I_{BL(i,j)} = 0 \quad \text{for } i > 1 \quad (1d) \]

Using Ohm’s law, we can express currents using node voltages, applied voltages, conductances of the interconnects along the word lines, \( G_{WL} \), conductances of the interconnects along the bit lines, \( G_{BL} \), and conductances of the devices, \( G_{(i,j)} \). By doing that and rearranging, we get Eq. (2)a–f.

\[ (2G_{WL} + G_{(i,j)}) V_{WL(i,j)} - G_{WL} V_{WL(i,j+1)} - G_{(i,j)} V_{BL(i,j)} = G_{WL} V_i \quad \text{for } j = 1 \quad (2a) \]

\[ (2G_{WL} + G_{(i,j)}) V_{WL(i,j)} - G_{WL} V_{WL(i,j-1)} - G_{(i,j)} V_{BL(i,j)} = 0 \quad \text{for } 1 < j < n \quad (2b) \]

\[ (2G_{WL} + G_{(i,j)}) V_{WL(i,j)} - G_{WL} V_{WL(i,j-1)} - G_{(i,j)} V_{BL(i,j)} = 0 \quad \text{for } j = n \quad (2c) \]

\[ (2G_{BL} + G_{(i,j)}) V_{BL(i,j)} - G_{BL} V_{BL(i+1,j)} - G_{(i,j)} V_{WL(i,j)} = 0 \quad \text{for } i = 1 \quad (2d) \]

\[ (2G_{BL} + G_{(i,j)}) V_{BL(i,j)} - G_{BL} V_{BL(i-1,j)} - G_{(i,j)} V_{WL(i,j)} = 0 \quad \text{for } 1 < i < m \quad (2e) \]

\[ (2G_{BL} + G_{(i,j)}) V_{BL(i,j)} - G_{BL} V_{BL(i+1,j)} - G_{(i,j)} V_{WL(i,j)} = 0 \quad \text{for } i = m \quad (2f) \]
Because the left-hand sides of Eq. (2)a–f can be expressed as linear combinations of node voltages, \( V_{WL(i,j)} \) and \( V_{BL(i,j)} \), one can set up Eq. (3).

\[
GV = I \quad (3)
\]

where \( G \) is a coefficient matrix containing conductances, \( V \) is a vector (or a matrix, if more than one set of voltages is applied) of node voltages, and \( I \) is a vector (or a matrix) of constants which, in this context, is known currents.

One can solve for node voltages \( V \) using conventional linear algebra methods. Branch currents can then be computed by applying Ohm’s law. In a special case where either one type of the interconnects, or both, have zero resistance, the number of unique nodes is reduced. In that case, one has to apply KCL to compute unknown currents through the perfectly conductive interconnects because it is not possible to apply Ohm’s law across elements with zero resistance.

3. Software framework

The \texttt{badcrossbar} package is implemented in Python 3. This programming language was chosen due to its popularity, intuitive syntax, and availability of open-source packages. Specifically, it features packages such as \texttt{numpy} or \texttt{scipy} that enable efficient manipulation of vectors and matrices necessary for computing currents and voltages in crossbar arrays. It is true that Python is an interpreted language which might result in slower computation when compared to compiled languages. However, \texttt{numpy} and \texttt{scipy} are written mostly in C [16], so there may be only a small decrease in performance due to pure Python code—these two libraries handle the heaviest computations in \texttt{badcrossbar}. Finally, it is important to note that Python is widely used nowadays, especially among machine learning researchers. \texttt{numpy} arrays employed by \texttt{badcrossbar} are easy to convert to data structures used by the most popular machine learning frameworks, such as TensorFlow or PyTorch that are often utilised when simulating crossbar-based neural networks.

3.1. Software architecture

\texttt{badcrossbar} contains two sub-packages: \texttt{badcrossbar.computing} and \texttt{badcrossbar.plotting}. The former implements computation of currents and voltages in crossbar arrays described by parameters passed to function \texttt{badcrossbar.compute()} . The latter is used to colour branches and nodes of those arrays according to the values that are passed to functions \texttt{badcrossbar.plot.branches()} and \texttt{badcrossbar.plot.nodes()}.

3.2. Software functionalities

3.2.1. Computing

\texttt{badcrossbar.compute()} computes currents and voltages in a crossbar array. As an input, it requires applied voltages, resistances of the crossbar devices and resistances of the interconnects. Interconnect resistance can be specified separately for the interconnects along the word and bit lines, or a single value can be provided for both types of interconnects. \texttt{badcrossbar.computing} employs nodal analysis and so most of the computation time is spent on computing node voltages. However, if one does not intend to use all the branch currents and node voltages that the sub-package is capable of computing, optional keyword arguments can be used to return \texttt{None} instead of voltage values and some of the current values in order to save memory (savings in time are usually minimal). Finally, optional keyword arguments can be used to control the messages that are shown during various computing stages. A complete description of the arguments can be found in the docstrings of \texttt{badcrossbar.compute()}.

The output of \texttt{badcrossbar.compute()} is a named tuple with fields voltages and currents. Both of the fields are named tuples themselves. voltages has fields word_line and bit_line that contain node voltages on the word and bit lines, respectively (visualised in Fig. 2A). currents has fields device, word_line and bit_line that contain currents flowing through devices, word line interconnects and bit line interconnects, respectively (visualised in Fig. 2B). It additionally has a field output.
which contains the output currents as a 2D array. It represents essentially the same information as the currents in the last row of currents.bit_line, just in a more convenient format. Assuming a crossbar with m word lines, n bit lines and p sets of applied voltages, all the computed voltages and currents, except for the output currents, will be arrays of shape $m \times n$ if $p = 1$ and arrays of shape $m \times n \times p$ if $p > 1$. Output currents, on the other hand, will always be arrays of shape $p \times n$.

4.2. Plotting

Functions \texttt{badcrossbar.plot.branches()} and \texttt{badcrossbar.plot.nodes()} accept either individual arrays associated with different types of branches or nodes, or named tuples (usually produced by the function \texttt{badcrossbar.compute()}) containing currents and voltages. Although the most likely variables to be plotted using these functions are currents and voltages, the arrays that are being passed can be associated with any type of numeric variables. Importantly, a large number of optional keyword arguments allows to modify the produced diagrams. Widths of the wires, devices or nodes, the colour scheme of the colour bar, and other elements of the diagram can be changed. Because the functions produce vector diagrams (in PDF), they can be easily modified in any vector graphics manipulation program. Thus, the optional keyword arguments that are provided mostly allow to modify parameters that would be difficult to change once the diagram is produced. A complete description of the arguments can be found in the docstrings of \texttt{badcrossbar.plot.branches()} and \texttt{badcrossbar.plot.nodes()}.

5. Illustrative example

Suppose we wanted to compute branch currents in a crossbar array over multiple sets of inputs and then plot their average values over all those sets of applied inputs. The following piece of code computes branch currents and node voltages in a $3 \times 5$ crossbar array over four sets of applied voltages. It then plots average branch currents over those four sets of inputs. When executed, this code produces a PDF file named “Example.pdf”; its contents are shown in Fig. 4.

```python
import badcrossbar

# Applied voltages in volts.
applied_voltages = [[1.5, 4.1, 2.6, 2.1],
                    [2.3, 4.5, 1.1, 0.8],
                    [1.7, 4.0, 3.3, 1.1]]

# Device resistances in ohms.
resistances = [[345, 903, 755, 257, 646],
               [652, 401, 508, 166, 454],
               [442, 874, 190, 244, 635]]

# Interconnect resistance in ohms.
r_i = 0.5

# Computing the solution.
solution = badcrossbar.compute(
    applied_voltages, resistances, r_i)

# Plotting average branch currents over all
# sets of inputs.
badcrossbar.plot.branches(
    currents=solution.currents,
    axis_label='Average current (A)')

More examples can be found in the examples folder in the GitHub repository of \texttt{badcrossbar} package. Additionally, folder \texttt{tests} contains a number of tests that check the correctness of the results by comparing them with equivalent circuits simulated in Qucs circuit simulation software. Tests include both a conventional configuration, as well as a special case—a crossbar consisting of only a single device. Finally, there are tests that validate the outputs or behaviour of some of the individual functions.

6. Impact

\texttt{badcrossbar} is the first open-source Python package for analysing line resistance effects in resistive crossbar arrays. The package is able to compute voltages and currents in a matter of seconds or minutes even with relatively large crossbar arrays and large number of applied inputs. This will be especially useful in simulating crossbar-based neural networks, where such computations are necessary to accurately evaluate the outputs.
Furthermore, we plan to add more options for specifying plotting and to allow to specify the resistances of individual interconnects. More control over how voltages are applied across crossbar arrays is expected, as well as allowing to export to other formats, e.g., TikZ.

Fig. 3. Performance of computing currents and voltages in crossbar arrays. The median time (out of 10 for every data point) is shown for crossbars of shape \( n \times n \) for various numbers of sets of applied voltages.

Fig. 4. The output of a PDF file produced by `crossbar.plot.branches()` when given multiple sets of currents in different branches of the crossbar array. Because arrays contain data from multiple sets of applied voltages, average values over all sets are plotted.

We present a Python package for computing currents and voltages in crossbar arrays, as well as plotting them (or any other numerical variables) on these structures. The use of numpy enables easy integration with a wide range of packages, while the use of vector graphics in produced diagrams allows to modify them using external programs. In the future, we plan to give users more control over how voltages are applied across crossbar arrays and to allow to specify the resistances of individual interconnects. Furthermore, we plan to add more options for specifying plotting behaviour, as well as to allow to export to other formats, e.g., TikZ.

7. Conclusions

We present a Python package for computing currents and voltages in crossbar arrays, as well as plotting them (or any other numerical variables) on these structures. The use of numpy enables easy integration with a wide range of packages, while the use of vector graphics in produced diagrams allows to modify them using external programs. In the future, we plan to give users more control over how voltages are applied across crossbar arrays and to allow to specify the resistances of individual interconnects. Furthermore, we plan to add more options for specifying plotting behaviour, as well as to allow to export to other formats, e.g., TikZ.

Fig. 4. The output of a PDF file produced by `badcrossbar.plot.branches()` when given multiple sets of currents in different branches of the crossbar array. Because arrays contain data from multiple sets of applied voltages, average values over all sets are plotted.

of these structures and where large numbers of inputs have to be tested [10]. Additionally, the plotting functionality may help visualise physical variables and discover trends, such as where in crossbar arrays current decreases tend to be the most severe. We believe that this package will be useful in both the simulations and the design of crossbar arrays.

D. Joksas and A. Mehonic

SoftwareX 12 (2020) 100617

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

A.M. acknowledges funding from the Royal Academy of Engineering, United Kingdom under the Research Fellowship scheme. D.J. acknowledges studentship funding from the Engineering and Physical Sciences Research Council, United Kingdom (ref. 2094654).

References