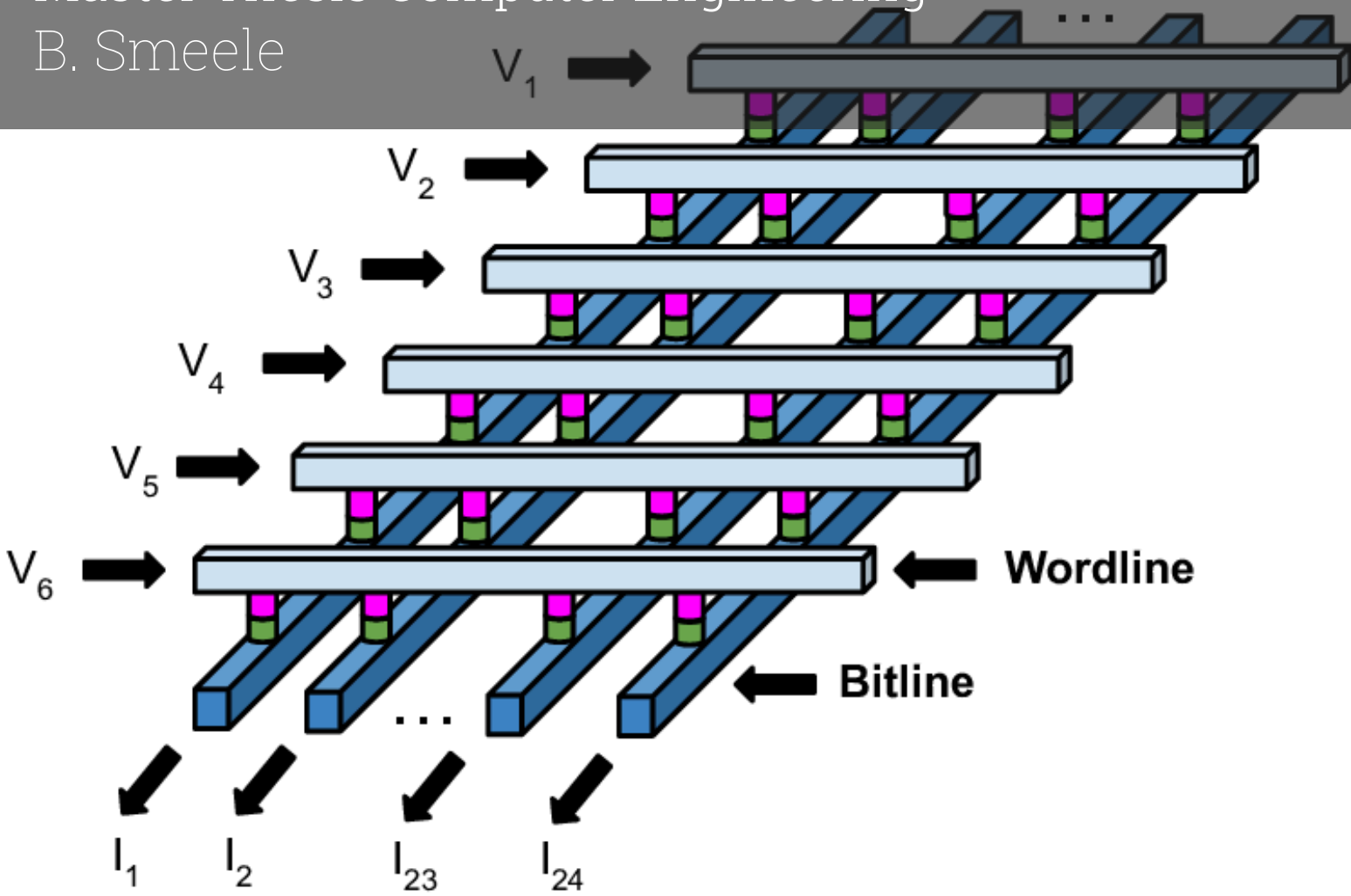
Precise and Fast Simulation of RRAM Crossbar Arrays

The design and implementation of a fast simulation framework for resistive memristor-based computation in memory crossbar arrays

Master Thesis Computer Engineering

B. Smeele



TU Delft

Precise and Fast Simulation of RRAM Crossbar Arrays

The design and implementation of a fast simulation framework for resistive memristor-based computation in memory crossbar arrays

by

B. Smeele

Student Number: 4895827

Supervisor:	G. Gaydadjev
Daily Supervisor:	T. Spyrou
Project Duration:	September, 2024 - August, 2025
Faculty:	Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft

Cover:	Arjun Tyagi and Shubham Sahay [1]
Style:	TU Delft Report Style, with modifications

TU Delft

Preface

This thesis represents the culmination of the last years work. The culmination of staring at diagrams until the shapes started making sense. And the culmination of long evenings arguing with code that refused to cooperate. Nevertheless, I'm happy with the results and the things I learned along the way.

First, I would like to thank my supervisors Georgi Gaydadjiev and Theofilos Spyrou for their knowledge and guidance during our regular progress meetings. Next, I would like to thank Matthias Möller for his support with several mathematical related problems I ran into. I would also like to thank Konstantinos Stavrakis and Emmanouil Arapidis for their interest in my work, help with gathering data, and useful discussion meetings. I would like to thank my fellow master students for their questions and suggestions. And finally, I would like to thank my friends and family for pretending to be interested whenever I needed to talk about something.

B. Smeele
Delft, August 2025

Abstract

This thesis presents a RRAM-crossbar simulation framework aimed at precise and fast simulation. Specifically, the model simulates a memristive crossbar with parasitic wire resistances, generating output currents and tracking the internal memristive states based on applied input voltages. Implemented in C++, the developed model has a clear interface making it suitable for integration into larger simulation frameworks for CIM-based accelerators. Experimental results demonstrate that the simulator is significantly faster than Cadence Spectre, while maintaining sufficient accuracy for use in behavioral-level CIM simulations.

The complete model is divided into three sub-modules: a memristor model, a linear crossbar model, and a non-linear crossbar model. The memristor model is based on the JART VCM v1b var model by Bengel et al. [2], and its implementation is a faithful translation of the associated Verilog-A code [3], with additional functionality for simulation in C++. Additionally, other memristor models can be implemented through an abstract memristor class.

Experimental results show that the developed simulation framework is significantly faster than Cadence Spectre while retaining a relative error below 1%. Additionally, the crossbar and the memristors are modeled to more closely reflect physical device behaviour than in other RRAM simulation frameworks. Finally, a study of linear and non-linear memristor simulation in a crossbar shows a measurable difference, suggesting approximating memristors as linear impacts simulation accuracy.

Contents

1 Introduction

This chapter briefly introduces the topic of this thesis, its relevance, and the main contributions of this work.

1.1. Motivation

There is an ever-increasing need for computing power. The recent rise of machine learning and deep neural networks has created an unprecedented demand for computational resources, while Big Data applications continue to drive the need for higher data throughput. In 2018, OpenAI released a report that found the compute demand for AI training doubles every 3.4 months [4] instead of every 2 years as predicted by Moore's Law. Similarly, High-performance computing (HPC) spending is increasing by 15% annually [5], compared to the $3-7\%$ increase of general IT hardware.

The traditional Von Neumann architecture is simply unable to keep up. It is estimated that $30-60\%$ of HPC workloads are limited by memory latency or bandwidth [6]. Google workloads are estimated to spend 62.7% of total system energy on data movement [7]. And generally, data transfer between CPUs and off-chip memory has been shown to be significantly slower and consume two orders of magnitude more energy than floating point operations [8].

This issue has been noticed as early as 1994, when Wulf and McKee [9] warned of a 'memory wall': a point where system performance is completely determined by memory speed; where making the processor faster will not affect the wall-clock time. despite advancements in cache hierarchies and memory technology, the issue still persists and may be more relevant than ever.

One promising approach to address the memory bottleneck is to move compute closer to the memory locations. Near-memory computing paradigms aim to place processing circuits next to memory arrays, enabling certain operations without the need for transferring data off-chip. Computing in-memory (CIM) paradigms take it one step further and aim to perform certain operations directly within the memory arrays. Both approaches reduce the effects of the memory wall by either reducing the distance over which data needs to be transferred, or by removing the need for data transfer altogether.

CIM in particular has become a popular technique for accelerating deep neural networks as its use of crossbar structures for memory arrays is well suited for matrix-vector multiplication. For example, Yuhao Ju et al. achieved a $37\% - 55\%$ end-to-end latency reduction on artificial intelligence related applications, and a $17.8\times$ CPU energy efficiency improvement [10]. Similarly, Sumon Kumar Bose et al. achieved over $70\times$ energy savings and over $3\times$ improved processing time for binary image filtering [11].

Within CIM, RRAM has emerged as an attractive technology due to its scalability, high density, low power consumption, fast operating speeds, and high compatibility with CMOS technologies [12]. Additionally, its resistive nature inherently supports fully parallel matrix-vector multiplications. However, RRAM is still an emerging memory technology, and reliable, large-scale manufacturing has yet to be

achieved. RRAM implementations need to account for non-idealities, such as nonlinearity, device-to-device and cycle-to-cycle variability, and limited resolution [13]. Furthermore, non-idealities of the crossbar, such as sneak-path current and line parasitic, also play an important role. As a result, simulation tools are required for accurate IC design.

1.2. Problem Statement

A wide range of simulation tools exist to support the design of RRAM-based systems. Notably, Cadence Spectre, and similar SPICE-class simulators, offer high accuracy for circuit-level simulation. However, such tools are designed for general purpose applications, thus can take a significant amount of time to produce simulation results. More specialized tools take advantage of the RRAM crossbar structure to greatly improve execution times. However, currently available RRAM specialized simulation tools either omit physically significant effects or oversimplify models, sacrificing accuracy for speed.

As a result, there is a lack of tooling that bridges the gap between high accuracy general purpose tools and low execution time specialized tools. This thesis addresses the question whether the fixed and repetitive structure of RRAM crossbar arrays be exploited to develop a tool with SPICE-comparable simulation accuracy with faster execution times.

1.3. Contributions

The aim of this thesis is to develop a RRAM crossbar simulation framework aimed at fast and precise simulation. The framework is written in C/C++ in a modular manner in order to be compatible with various simulation platforms and be integrable in a larger simulation framework for CIM-based accelerators. It incorporates a memristor model derived from an accurate Verilog-A implementation and offers abstractions for the implementation of other memristor models. Furthermore, the simulator accounts for parasitic line resistances within the crossbar and is compared to and validated with Cadence Spectre.

The main contributions of this work are as follows:

- **A physically accurate memristor model derived from Verilog-A**: the simulator used the JART VCM v1b var [2] memristor model by default. This C++ implementation is adapted from the original Verilog-A version [3] and includes additional functionality to replace features otherwise provided by SPICE simulators.
- **Abstract memristor implementation**: all memristor models in this simulator are derived from a common abstract memristor class, allowing for straightforward implementation of new memristor models.
- **Non-linear RRAM simulation with parasitic line resistances**: RRAM crossbar arrays are simulated with non-zero wire resistances and support non-linear memristive devices.
- **Validation using Cadence Spectre**: the simulator is validated using Cadence Spectre by simulating a large number of crossbars of various sizes.
- **Linear vs non-linear simulation analysis**: this work includes an analysis of the effectiveness of approximating non-linear memristive devices as linear.

1.4. Thesis Organization

The remainder of this thesis is as follows:

- **Chapter 2** provides necessary background information as well as an overview of related works
- **Chapter 3** discusses the relevant theory and implementation details of the simulation framework
- **Chapter 4** presents the results for the various subsections of the simulator, several experiments of the full framework, and an analysis of linear and non-linear simulation. This chapter also discusses the results of each experiment
- **Chapter 5** provides a summary of this thesis and suggest various areas of improvements and future research

2

Background

2.1. Memristors

In 1971, Chua [14] first proposed a fourth ”missing” circuit element called the memristor, a concatenation of *memory* and *resistor*. This ”missing” circuit element was envisioned to be a two-terminal device characterized by a relationship between charge and flux, and theorized to behave somewhat like a non-linear resistor with memory. Later in 1976, Chua et al. [15] generalized the concept to a broader class systems called *memristive systems*. These more general systems are defined by two equations: a state equation (2.1), and an output equation (2.3). This broader definition allows the flux in the memristive system to no longer be uniquely defined by the charge.

In 2008, HP labs announced the first physical realization of a memristive device along with a physical model [16] with matching behavior. Their device was based on a Tin-Oxide filament between two electrodes, where the state change was realized by migrating oxygen vacancies.

In general, a memristor can be considered a two- or three-terminal device described by a state equation, an output equation, and having certain characteristics, often called fingerprints [17]. The most recognizable of these characteristics is a pinched I-V hysteresis loop: a looped curve with two distinct paths depending on the direction traveled, and which always passes through the origin.

2.1.1. Resistive Memristor Mechanism

A resistive memristor is a two-terminal device consisting of two electrodes with some filament in between. Based on an applied voltage or current, the internal state of the filament can change. This change of internal state alters the resistance of the device, and can generally be reversed by with a second applied voltage or current. The polarity of this second applied voltage or current determines whether the device is considered unipolar or bipolar. For a unipolar device, the switching voltages or currents have the same polarity. Whereas for a bipolar device, the switching voltages or currents have opposite polarity. In both cases, the result is that the voltage-current relationship forms a hysteresis loop pinched at the origin where both current and voltage are zero.

Generally, resistive memristor are said to have two states: a SET state where the device is in a low resistance state, and a RESET state where the device is in a high resistive state. However, by carefully controlling the applied voltage and current, intermediate resistance states can also be achieved. Several distinct physical processes can drive this change in resistance, most common of which are the following:

- Resistive switching memristors, or RRAM, which consists of an insulating layer in between two metal electrodes. State change is achieved through the formation and rupture of conductive filaments in the insulating layer. [18]
- Phase change memory, which relies on a chalcogenide material as its switching medium. Phase change is achieved by selectively heating and quenching the material in such a way to switch between a crystalline and a amorphous phase. [19]

- Spin-transfer torque devices, which consist of a tunneling dielectric in between two ferromagnetic layers. By altering the relative orientation of the two ferromagnetic layers, the resistance of the device can be altered. [20]

2.1.2. Memristor modeling

The defining feature of a memristor is a changing resistances based on changes in the filament of the device. Thus, central to memristor modeling is a state variable that quantifies the state of the filament. Furthermore, the state of the memristive filament changes over time based on applied voltages and currents. This is generally modeled with a state equation (2.1): an equation that expresses the time derivative of the state variable as a function of the input (u), the state variable (w), and time (t).

$$\frac{dw}{dt} = g(w, u, t) \tag{2.1}$$

Generally, the state variable of a model is linked to some physical mechanism of a memristive device. This mechanism operates within some limits, thus the state variable should be bound to some interval. This can be modeled by adding a window function to the state equation (2.2).

$$\frac{dw}{dt} = g(w, u, t) * f(w) \tag{2.2}$$

The state equation is multiplied by the window function $f(w)$ to ensure the state variable stays within its bounds. Moreover, certain window functions can model additional boundary effects.

The second necessity of a memristor model is an output equation (2.3): an equation that defines the relation between the output (y), and the input (u) and state variable (w). Generally a memristive device takes a voltage as input which produces an output current, although current controlled memristive devices also exist.

$$y = h(w, u, t) * u \tag{2.3}$$

Depending on the specific implementation of the memristor model, several other parameters can play an important role [21], the most important of which are:

- R_{ON}: the resistance in the SET state, also called the Low Resistance State (LRS)
- R_{OFF}: the resistance in the RESET state, also called the High Resistance State (HRS)
- I_{ON}: the maximum amount of current flowing through the memristor during the SET process
- I_{OFF}: the maximum amount of current flowing through the memristor during the RESET process
- V_{SET}: the minimum voltage needed to SET the memristor
- V_{RESET}: the minimum voltage needed to RESET the memristor
- T_{SET}: the minimum time required for the SET procedure
- T_{RESET}: the minimum time required for the RESET procedure

2.1.3. Notable Memristor Models

Current research offers a wide variety of memristor models for both unipolar and bipolar memristors [12] [21]. These models can either be based on physical devices and mechanisms, or they can be analytical and tunable to fit experimental data. The following are some notable memristor models:

- Linear Ion Drift model, developed by HP lab [16], is the first physical memristor model. It is a model for bipolar devices which models the memristor as a device with two regions: a doped region, and an undoped region. By changing the width of the doped region, a change in resistance is achieved.
- Non-linear Ion Drift model, also developed by HP lab [22], is the first non-linear physical model. The model equally splits the device into a doped and undoped region. However, this model defines a nonlinear dependence between the internal state derivative and the voltage. Additionally, the I-V relationship is also non-linear.
- Simmons Tunneling Barrier [23] [24] model is the first SPICE compatible model. This physical model has a higher complexity, allowing for more accuracy. Memristors are modeled as having a resistance in series with a tunneling barrier, of which the width acts as the state variable.

- TEAM [25] model is an analytical model designed to be similar to the Simmons Tunneling Barrier model, but using simpler mathematical equations. One of the simplification is a threshold current, below which the state variable does not change. Additionally, the I-V relationship is not fixed, thus can be chosen to fit any device. This results in a memristor model that can be tuned to fit a variety of other models.
- VTEAM [26] model is a modified version of the TEAM model. This model uses a threshold voltage instead of a current, making it more appropriate for certain logic and memory applications. It retains the same flexibility as the TEAM model.
- Stanford/ASU [27] model is a SPICE compatible model specifically developed for RRAM applications. It models bipolar memristors as having a growing conductive filament, at the end of which is a tunneling gap. Additionally, this is one of the first model to include joule heating effects.

For this thesis, the JART VCM v1b var [2] model will be used. However, different models can be implemented through an abstract memristor class as described in section 3.1.2. The model and its implementation is further discussed in section 3.1.

2.2. Compute-in-Memory

During conventional operation, at most one memory cell in a crossbar array is activated per bitline. This ensures accurate sensing and avoids accidental write operations, conflicting reads, and other possible issues. However, by carefully manipulating the activation, several cells can be activated to allow for certain computational operations from within a memory array. For example, activating two cells per bitline could act as an OR gate: if either memory cell has a stored 1, the bitline will be driven to 1. Otherwise it will remain 0.

SRAM is a memory technology widely used in crossbar arrays for its high speeds and reliability. Consisting of 6 transistors, its structure provides a simple interface and is fully CMOS compatible. Naturally, the SRAM memory cell has also been adapted to allow for compute in memory. For example, Monga et al. [28] adapted the 6T structure into a 9T cell capable of performing logic operations within a memory array. However, due to the nature of SRAM cells, care should be taken to avoid accidental bit-flips and reduce leakage power when using this technology for in-memory operations.

RRAM is an emerging non-volatile memory technology capable of high speed, low power operation with a small footprint. It is especially well suited for compute-in-memory applications due to its resistive nature. Each RRAM cell contains a resistive element, which combined with Ohm's law, inherently performs a multiplication: the applied voltage multiplied by the RRAM conductance equals the current. Combined with Kirchhoff's current law, RRAM crossbar arrays are well suited for performing multiply-and-accumulate operations and vector-matrix products within memory.

2.2.1. Resistive Crossbar Array

Resistive crossbar arrays are crossbar arrays where the memory cells consist of a resistive device. Voltages applied to the crossbar edges propagate to the memory cells which, through Ohm's law, results in current proportionals to their resistances. The current flowing through each cell is accumulated along each column through Kirchhoff's current law and represents the output of the crossbar.

This structure of resistive memory cells mimics a vector-matrix multiplication when an input voltage is applied. For example, a voltages applied to the rows of a crossbar mimic an input vector when columns are set to ground. The voltage across each memory device is subsequently identical to the voltage applied to the row that device is in. Through Ohm's law, the conductance of each memory cell is multiplied by the appropriate input, meaning the conductance of the memory cells mimic a weight matrix. Finally, Kirchhoff's current law accumulates the multiplications along each column, which in turn is the result vector of the vector-matrix multiplication. Fig. 2.1 shows an example of how a vector-matrix multiplication could map to a resistive crossbar array.

Resistive crossbars are commonly modeled with wire resistances to mimic real-world material properties. These resistances are places in between memory devices on rows and columns. The result is a more realistic representation of a resistive crossbar. However, these additional resistances can significantly impact operation results. Resistance on a wire before the memory cell inhibits small voltage

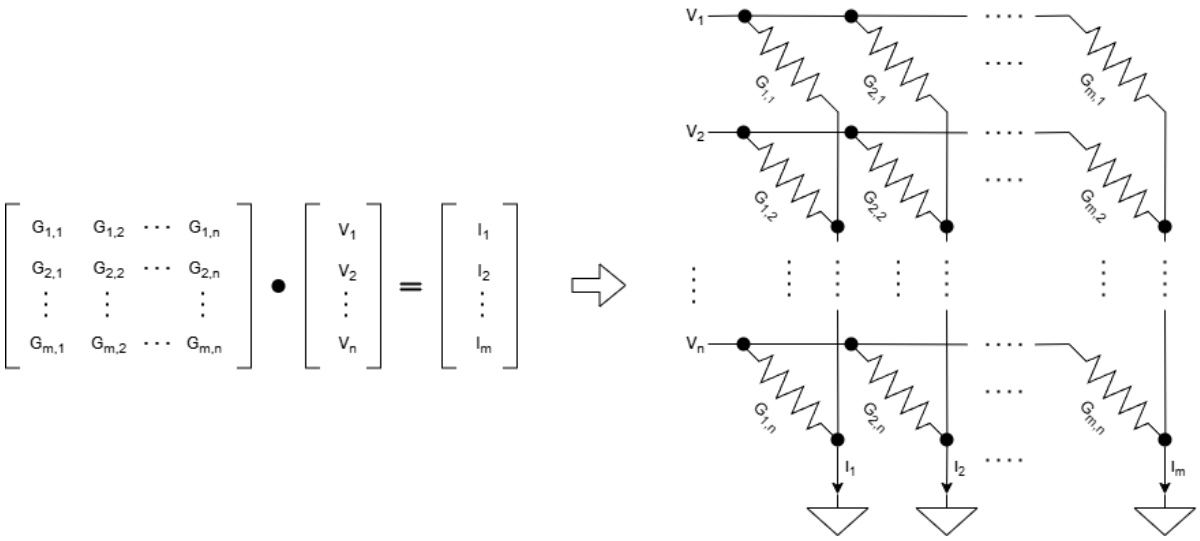


Figure 2.1: Vector-Matrix multiplication mapped to a resistive crossbar

drops, which in turn reduces the total voltage applied to a memory cell. This reduced voltage in turn results in a lower current, which reduces the accumulated current on the columns of the crossbar.

2.2.2. Resistive Crossbar Modeling

The emergence of new memory technologies requires the development of new simulation tools. RRAM in particular requires a specific model for simulating a resistive crossbar array. The result of an operation on such an array is the output currents at the bitlines. Therefore, those values are of particular importance when developing a model. Alternatively, a list of voltages at each wordline and bitline node could provide the same result. Such a list would include the voltage across each memory device, from which individual currents can be calculated and accumulated.

Other important simulation results include energy consumption and propagation delay. In a purely resistive network, energy consumption can be calculated from the nodal voltage list. Propagation delay, on the other hand, requires a more complicated model.

Finally, resistive crossbar arrays are usually constructed with memristors as part of each memory cell. These resistive devices need some additional consideration when paired with a crossbar model. Namely, memristors are not linear devices. Thus a crossbar model assuming purely linear resistors can not be applied directly. More importantly, memristors have an internal state that must be modeled, particularly during write operations. This possibly introduces additional requirements for a crossbar model.

The following are three notable memristor models:

- The so called ”Fast Crossbar Model” (FCM), part of the RxNN framework [29], provides a lossless transformation of a purely resistive crossbar to a matrix equation of the form $I = G * V$. Once the transformation is performed, a crossbar can be evaluated with a single matrix-vector product. This model is specifically designed for read operations, and therefore assumes linear devices and provides no method of simulating the individual memristor state variables. As part of the RxNN framework, the crossbar model provides additional methods of modeling memristor variation and modeling Digital-to-Analogue converters (DACs) and Analog-to-Digital converters (ADCs).
- Xie et al. [30] propose a crossbar model that reduces a resistive crossbar to two matrix equations - one representing the wordlines and representing the bitlines. To solve this system, they introduce a method that combines the two equations and solves the resulting system using Newton’s method. The Jacobian matrix required for this process is subsequently determined using the Generalized Minimal Residual (GMRES) method, with a preconditioner to accelerate its execution. The resulting method solves for the nodal voltages of a resistive crossbar array with

non-linear memristive devices.

- Fouda et al. [31] propose a method which models a crossbar array as a resistive network with parasitic capacitances and memristive devices. The result is a first-order differential equation expressed in terms of the nodal voltages. This equation has a closed-form solution in the time domain or s-domain, assuming linear memristive devices. When assuming non-linear memristive devices, it results in a system of non-linear equations depending on the I-V characteristics. Additionally, Fouda et al. propose an extension to this model for interconnect, and present a closed form equation for modeling Elmore delay.

The crossbar models discussed in this section offer variety of approaches offering a several levels of granularity. RxNN [29] is the most simple model discussed here, meant for static crossbars with linear devices. The other two models are more general: both support non-linear memory devices and allow for integration in systems with individual device models. Furthermore, the model developed by Fouda et al. [30] includes parasitic capacitances and delay models at the cost of added complexity.

For this thesis, a crossbar model developed by A. Chen [32] is used. This model performs a similar role to the model by Xie et al. [30] with a more simple matrix equation. It only supports linear devices but can easily be extended to support non-linear devices. This model and its implementation is further discussed in section 3.2.

2.2.3. Crossbar simulators

Simulation frameworks for RRAM crossbars have been developed to model their performance and behavior in various computational applications, particularly for deep neural networks. RRAM based crossbars are well-suited for vector-matrix multiplication, making training and inference of such network a natural fit.

The following are four notable RRAM simulation frameworks and how each handles crossbar simulation.

- MemTorch [33] is an open-source simulation framework for memristive deep learning systems. It enables the training and inference of neural networks by integrating directly with Pytorch, a widely used Machine Learning library. MemTorch is written in Python and uses bindings to C++ and CUDA for performance critical tasks. In terms of crossbar simulation, MemTorch assumes a linear resistive crossbar, where each memristive device is assumed to have an R_{on} or R_{off} depending on its internal state. Furthermore, MemTorch supports various non-ideality models such as quantization, device faults, and retention.
- RxNN [29] is a simulation framework designed for the functional evaluation of large-scale deep neural networks. It is based on the Caffe deep learning framework and makes use of the Basic Linear Algebra Subprograms (BLAS) interface. In terms of simulation, RxNN operates in three stages: first the weight matrices of a given model are mapped to a target hardware architecture. Next, the conductance matrices for each resistive crossbar are transformed into a non-ideal conductance matrix based on device variation profiles and crossbar non-idealities. Finally, the network can be evaluated for a given input by applying a digital-to-analog model, the non-ideal crossbar model, and an analog-to-digital model. The primary objective of this simulation framework is to asses the inference accuracy of a trained neural network. However, RxNN can also generate execution traces to enable performance and energy estimation, and can be used as a model-in-the-loop to assist in retraining.
- NeuroSim [34] is a circuit-level macro model designed to estimate the area, latency, dynamic energy, and leakage power of neuro-inspired architectures. By itself, NeuroSim does not perform circuit level simulation and therefore can able asses inference accuracy. However, NeuroSim has been extended in several ways to address these limitations. Specifically, DNN+NeuroSim [35] [36] extends NeuroSim to interface with PyTorch and TensorFlow, enabling training and inference of neural network. In terms of simulation, DNN+NeuroSim relies on PyTorch and TensorFlow, and applies hardware based constraints to the network weights, including retention models, variation profiles, and weight precision.
- The IBM Analog Hardware Acceleration Kit [37] is an open source framework designed for simulating analog crossbar arrays. The toolkit is written in python, makes use of an optimized C++/CUDA

core library, and is well integrated with PyTorch. Central to the toolkit is the ”analog tile”, which corresponds to a 2D weight matrix stored on a resistive crossbar array, and includes pre- and post-processing hardware. Each ”analog tile” performs vector-matrix multiplication on the weight matrix while applying non-ideality models such as quantization, variation profiles, and device response curves.

Based on the available literature, there seems to be a general trend in RRAM simulators towards machine learning in python-based environments. This is not surprising given the current prevalence of Deep Neural Network and the widespread use of python frameworks such as PyTorch and TensorFlow within the machine learning field.

Furthermore, available RRAM simulators can be broadly categorized into two classes based their treatment of crossbar modeling. In simulators, such as NeuroSim [35] [36] and the IBM Analog Hardware Acceleration Kit [37], each memory device is represented by a single value corresponding to a weight in the network. Simulation is subsequently preformed using an idealized crossbar and additional non-ideality functions can be applied to the weights and results to mimic the underlying hardware. Other simulators, such as MemTorch [33] and RxNN [29], include line resistances in their model of a crossbar array. And similarly to the other simulator class, additional non-ideality functions can be applied to weights and results.

However, neither simulator approach is able to accurately capture the behaviour of individual memristor models. MemTorch comes closest to addressing this limitation. Whereas the other discussed simulators reduce a memory device to a single value, MemTorch allows for memristor to be implemented as classes with methods to simulate certain behaviors. Nonetheless, their implementation is still inherently limited to linear memristor devices and approximates non-linearity with external non-ideality profiles. As a result, none of the discussed simulators is able to provide modeling with SPICE level accuracy.

Table 2.1 compares the discussed frameworks with the simulator developed in this Thesis. The points of comparison are as follows:

- Abstraction level:
  - Algorithm: Simulation does not include underlying circuitry. Profile and approximation functions may be applied
  - Circuit: Simulation models the underlying crossbar as a circuit
  - Device: Simulation includes individual device models
- ML framework integration: Whether the framework has integrated external machine learning platforms
- GPU acceleration: Whether the framework is GPU accelerated
- Language: The language in which the framework is written
- Open-source: Whether the source code of the frame-work is freely available
- IR-drop / sneak paths: Whether the framework simulates IR-drop and sneak path currents
- Retention / drift: Whether the framework models retention and drift
- Simulated writing: Whether the framework is able to simulate write operations

	RxNN	IBM	NeuroSim	MemTorch	This work
Abstraction Level	Circuit	Algorithm	Algorithm	Device and Circuit	Device and Circuit
Device Abstraction	Linear weight	Linear weight	Linear weight	Linear model	Non-linear model
ML framework integration	Caffe	PyTorch	PyTorch	PyTorch	None
GPU accelerated	No	Yes	No	Yes	No
Language	C++	Python, C++, CUDA	C++	Python, C++, CUDA	C++
Open-source	No	Yes	Yes	Yes	Yes
IR-drop / sneak paths	Yes	No	No	Yes	Yes
Retention / drift	No	Yes	Yes	Yes	Yes
Simulated writing	No	Yes	Yes	Yes	Yes

Table 2.1: Comparison table between this work and various other simulation frameworks

3

Theory and Implementation

This chapter provides the relevant theory, derivations, and implementation details related to the memristor crossbar simulator. The simulator is designed to accurately simulate a crossbar containing memristive devices to which time varying voltages are applied. To achieve this, the crossbar simulator is split into three subcomponents: a memristor model, a linear crossbar solver, and a non-linear crossbar solver.

The memristor model, described in section 3.1, represents the memory devices that are used in the crossbar. The model contains a complete description of a memristor, including a voltage-current relationship, the switching mechanics, and time varying behaviour. As part of the complete system, a collection of model instances are used, one representing each memory device in the crossbar. The specific model used in this thesis serves as an example, different models can be implemented through an abstract memristor class as described in section 3.1.2.

The linear crossbar solver, described in section 3.2, is responsible for determining the effect the line resistances have on the system. Based on supplied input voltages, this solver calculates what the voltage is at every point in the crossbar. Notably, this solver replaces every memory device with a resistance, hence the name 'linear crossbar solver.'

The non-linear crossbar solver, described in section 3.3, combines the linear crossbar solver and the memristor model. Memristors are generally non-linear devices, and are thus not compatible with the linear crossbar solver. This component bridges that gap.

All components are combined into the complete crossbar simulator described in section 3.4. This compete system manages the creation and operation of its subcomponents, and provides an interface to interact with the simulator. The developed code is available on GitHub linked in appendix D.

Finally, this simulator uses the notion of a 'time step' to simulate time varying behaviour. Time-continuous signals are discretized and each simulation step captures the system at a specific moment. Furthermore, the system only calculates its evolution from one instance to the next, based on the length of each step.

3.1. Memristor model

The memristor model used in this thesis is the JART VCM v1b var model developed by Bengel et al. [2]. This physics based model simulates the behaviour of a bipolar memristive device that operates via the valence change mechanism (VCM). These devices consist of an insulating metal-oxide layer positioned between two metal electrodes. Due to the different work functions of the electrodes, the device forms a Schottky contact at one end, and an ohmic contact at the other end of the metal-oxide layer.

A change in resistance in this device is achieved through redistribution of oxygen vacancies. Applying positive or negative voltages at the electrodes can cause this redistribution, leading to the formation

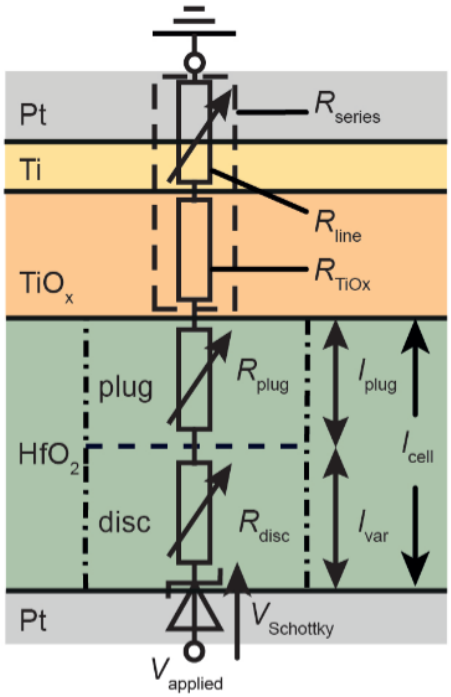


Figure 3.1: Equivalent circuit diagram of the JART VCM v1b var memristor model [2]

or dissolution of a conductive filament in the metal-oxide layer. In practice, this redistribution primarily occurs near one of the electrodes, while in other parts of the filament remain constant. Therefore, the model simplifies the filament by splitting it into two regions: a disc region, where the concentration of oxygen vacancies changes during switching, and a plug region, where the vacancy concentration remains constant. As a result, the entire device can be modeled by an equivalent circuit diagram shown in Fig. 3.1.

3.1.1. Model description

This section presents the equations describing the JART VCM v1b var model [38]. The purpose of each equation will briefly be explained, however the physical interpretation of these equations is outside the scope of this thesis. The implementation details for each equation will be further discussed in section 3.1.2.

The original description of the JART VCM v1b var model [38] includes mechanisms for device-to-device and cycle-to-cycle variability. These variation mechanisms are outside the scope of this thesis, thus have been removed from the model description and subsequent implementation.

A description of each parameter used in the model and the model equations can be found in appendix C.

As can be seen in Fig. 3.1, the model can be decomposed into four components, all of which are in series:

- a Schottky diode
- a disc resistance
- a plug resistance
- a series resistance

The series resistance can be further decomposed into a line resistance and an oxide resistance.

Disc Region

The resistance of the disc region (R_{disc}) is dependent on the oxygen concentration in that region (N_{disc}) as described in equation 3.1.

$$R_{disc} = \frac{l_{disc}}{zv_oAN_{disc}\mu_n} \tag{3.1}$$

The oxygen concentration in this region is consequently also the state variable of this model. The change in the state variable is mainly dependent on an ionic current (I_{ion}) as described in equation 3.2.

$$\frac{dN_{disc}}{dt} = -\frac{I_{ion}}{zv_oeAl_{disc}} \tag{3.2}$$

The ionic current is in turn mainly dependent on the hopping migration barriers for forward jumps ($\Delta W_{A_{min}}$) and backwards jumps ($\Delta W_{A_{max}}$) as described in equation 3.3.

$$I_{ion} = zv_oeAcv_oav_oF_{limit} \cdot (\exp(-\frac{\Delta W_{A,min}}{k_BT}) - \exp(-\frac{\Delta W_{A,max}}{k_BT})) \tag{3.3}$$

Additionally, I_{ion} includes F_{limit} which serves as a limiting factor for the ionic current to keep the vacancy concentration in the disc between $N_{disc,min}$ and $N_{disc,max}$. F_{limit} is described in equation 3.4.

$$F_{limit} = \begin{cases} 1 - (\frac{N_{disc,min}}{N_{disc}})^{10}, & \text{for } V_{applied} > 0\ V \\ 1 - (\frac{N_{disc}}{N_{disc,max}})^{10}, & \text{for } V_{applied} < 0\ V \end{cases} \tag{3.4}$$

The hopping migration barriers are mainly dependent on γ, as described in equation 3.5 and equation 3.6.

$$\Delta W_{A,min} = \Delta W_A \cdot (\sqrt{1-\gamma^2} - \gamma\frac{\pi}{2} + \gamma \cdot \arcsin(\gamma)) \tag{3.5}$$

$$\Delta W_{A,max} = \Delta W_A \cdot (\sqrt{1-\gamma^2} + \gamma\frac{\pi}{2} + \gamma \cdot \arcsin(\gamma)) \tag{3.6}$$

γ is mainly dependent on the electric field during SET and RESET operations, as described in equation 3.7.

$$\gamma = \frac{ezv_oaE_{SET/RESET}}{\Delta W_A\pi} \tag{3.7}$$

Finally, the electric field during SET and RESET operations are mainly depended on the voltage in the device, as is described in equation 3.8 and equation 3.9.

$$E_{SET} = \frac{V_{disc}}{l_{disc}} \tag{3.8}$$

$$E_{RESET} = \frac{V_{Schottky} + V_{disc} + V_{plug}}{l_{cell}} = \frac{V_{cell}}{l_{cell}} \tag{3.9}$$

For simplicity it can be said that the ionic current is mainly dependent on the voltages in the cell, as other parameters remain constant within a given time step.

Plug Region

The resistance of the plug region is dependent on the on the oxygen concentration in that region, as described in equation 3.10.

$$R_{plug} = \frac{l_{plug}}{zv_oAN_{plug}\mu_n} \tag{3.10}$$

However, the oxygen concentration in the plug region is assumed to remains constant during operation, thus the plug resistance also remains constant.

Series Resistance

The series resistance consists of two parts: $R_{series} = R_{TiO_x} + R_{line}$. The oxide resistance remains constant, while the line resistance is current dependent. The line resistance is described in equation 3.11.

$$R_{line} = R_0 \cdot (1 + \alpha_{line}R_0I^2R_{th,line}) \tag{3.11}$$

Schottky Diode

The Schottky diode is described by a voltage current relationship split into two equations for positive and negative voltages. These are described in equation 3.12 and equation 3.13 respectively.

$$I_{Schottky,V>0} = AA^* \cdot T^2 \exp(\frac{-e\phi_{B_n}}{k_B T}) \cdot (\exp(\frac{eV_{Schottky}}{k_B T}) - 1) \tag{3.12}$$

$$I_{Schottky,V<0} = -A\frac{A^* \cdot T}{k_B}\sqrt{\pi W_{00} e \cdot (-V_{Schottky} + \frac{\phi_{Bn}}{\cosh^2(\frac{W_{00}}{k_B T})}} \cdot \exp(\frac{-e\phi_{Bn}}{W_0})(\exp(\frac{-eV_{Schottky}}{\epsilon'}) - 1) \tag{3.13}$$

Where ϕ_{Bn} is described as in equation 3.14.

$$\phi_{Bn} = \phi_{Bn0} - \sqrt[4]{\frac{e^3 z v_O N_{disc}(\phi_{Bn0} - \phi_n - V_{Schottky})}{8\pi^2 \epsilon_{\phi_B}^3}} \tag{3.14}$$

Where W_{00} is described as in equation 3.15.

$$W_{00} = \frac{eh}{4\pi}\sqrt{\frac{z v_0 N_{disc}}{m^* \epsilon_s e}} \tag{3.15}$$

Where W_0 is described as in equation 3.16.

$$W_0 = W_{00} \tanh(\frac{W_{00}}{k_B T}) \tag{3.16}$$

And where ϵ' is described as in equation 3.17.

$$\epsilon' = \frac{W_{00}}{\frac{W_{00}}{k_B T} - \tanh(\frac{W_{00}}{k_B T})} \tag{3.17}$$

Since all components are in series, the current described by the Schottky diode is also the current through the other components. This in turn determines the resistances and voltages across the other components. The voltages then determine the ionic current, which changes the state parameter. However, it should be noted there is no direct equation given for the voltage across the Schottky diode. Instead it can be determined by subtracting the voltages across the other components from the total supplied voltage:

$$V_{Schottky} = V_{applied} - V_{disc} - V_{plug} - V_{series} \tag{3.18}$$

Temperature

For the whole filament, a homogeneous temperature is assumed. This temperature is dependent on the power dissipated in the cell, as described by equation 3.19.

$$T = I \cdot (V_{disc} + V_{plug} + V_{Schottky}) \cdot R_{th,SET/RESET} + T_0 \tag{3.19}$$

3.1.2. Model implementation

Bengel et al. [3] have implemented this model using Verilog-A and simulated it in Cadence Spectre. Verilog-A is a hardware description language specialized for analog systems. This allows for direct translation of the aforementioned equations while the simulation software takes care of the simulation itself. For this thesis however, the model is to be written in C++. C++ is a general purpose programming language and thus has no native support for analog hardware simulation, meaning it requires additional methods to properly implement this memristor model.

The C++ model is structured as a class with methods for the various equations from section 3.1.1, and with class fields for each parameter found in appendix C. Upon creation, the class is initialized with default values which can be changed by writing to the appropriate field. When possible, methods have been directly translated from the Verilog-A [3] implementation. The remainder of this section will outline each class method with a comparison to the associated equations and Verilog-A code when applicable.

UpdateTemperature()

This method calculates and updates the temperature variable as described in equation 3.19. listing 3.1 and listing 3.2 show the equivalent implementations in Verilog-A and C++ respectively.

Compared to equation 3.19, the Verilog-A and C++ implementations both calculate V_{disc} and V_{plug} from $V_{disc} + V_{plug} = (V_{discplugserial} * (R_{disc} + R_{plug})/(R_{disc} + R_{plug} + R_{series})$ instead of using V_{disc} and V_{plug} directly.

Listing 3.1: Verilog-A implementation of the temperature calculation based on equation 3.19

```
Pwr(ith_branch) <+ -(V(schottky)+V(discplugserial)*(Rdisc+Rplug)/(Rdisc+Rplug+Rseries))*I(
    schottky);
Pwr(rth_branch) <+ Temp(rth_branch)/Rtheff;
Treal = T0 + Temp(rth_branch);
```

Listing 3.2: C++ implementation of the temperature calculation based on equation 3.19

```
void JART_VCM_v1b_var::UpdateTemperature(double V_schottky, double V_discplugserial, double
    I_schottky) {
    Treal = I_schottky * (V_schottky + V_discplugserial * (Rdisc + Rplug) / (Rdisc + Rplug +
    Rseries)) * Rtheff + T0;
}
```

ComputeSchottkyCurrent()

This method calculates the current flowing through the Schottky diode based on the equation in section 3.1.1. Listing 3.3 and listing 3.4 show the equivalent implementations in Verilog-A and C++ respectively.

Compared to the equations in section 3.1.1 the computation of ϕ_{Bn} is slightly altered. Under certain conditions this parameter can become negative or complex, which is not physically meaningful. To prevent this, guards are added to limit ϕ_{Bn} to zero if it would be negative, and to ϕ_{Bn0} if it would be complex.

Listing 3.3: Verilog-A implementation of the Schottky current calculation based on the equations in section 3.1.1

```
if (V(schottky)<phibn0-phin)
        begin
                psi=phibn0-phin-V(schottky);
                phibn=phibn0-sqrt(sqrt(pow(`P_Q,3)*zvo*Ndisc*1e26*psi/(8*pow(`M_PI,2)*(pow(
                    epsphib_eff,3)))));
                if (phibn<0)
                        begin
                                phibn=0;
                        end
        end
else
        begin
                psi=0;
                phibn=phibn0;
        end

if (V(schottky)<0) //TFE Schottky SET direction
        begin
                W00=(`P_Q*`P_H/(4*`M_PI))*sqrt(zvo*Ndisc*1e26/(mdiel*eps_eff));
                W0=W00/tanh(W00/(`P_K*Treal));
                epsprime=W00/(W00/(`P_K*Treal)-tanh(W00/(`P_K*Treal)));
                Ischottkytunnel=-A*Arichardson*Treal/`P_K*sqrt(`M_PI*W00*`P_Q*(abs(V(schottky
                    ))+phibn/pow(cosh(W00/(`P_K*Treal)),2)))*exp(-`P_Q*phibn/W0)*(exp(`P_Q*
                    abs(V(schottky))/epsprime)-1);
        end
else            //Schottkydiode TE      RESET direction
        begin
                Ischottkytunnel= A*Arichardson*pow(Treal,2)*exp(-phibn*`P_Q/(`P_K*Treal))*(
                    exp(`P_Q/(`P_K*Treal)*V(schottky))-1);
        end
I(schottky) <+ Ischottkytunnel;
```

Listing 3.4: C++ implementation of the Schottky current calculation based on the equations in section 3.1.1

```
double JART_VCM_v1b_var::ComputeSchottkyCurrent(double V_schottky) {
```

```
    double phibn;
    if (V_schottky < phibn0 - phin) {
        double psi = phibn0 - phin - V_schottky;
        phibn = phibn0 - sqrt(sqrt(pow(P_Q, 3) * zvo * Ndisc * 1e26 * psi / (8 * pow(M_PI, 2)
     * (pow(epsphib_eff, 3)))));
        if (phibn < 0) { phibn = 0; }
    } else { phibn = phibn0; }

    if (V_schottky < 0) { // TFE Schottky SET direction
        double W00 = (P_Q * P_H / (4 * M_PI)) * sqrt(zvo * Ndisc * 1e26 / (mdiel * eps_eff));
        double W0 = W00 / tanh(W00 / (P_K * Treal));
        double epsprime = W00 / (W00 / (P_K * Treal) - tanh(W00 / (P_K * Treal)));
        return -A * ((Arichardson * Treal) / P_K) * sqrt(M_PI * W00 * P_Q * (fabs(V_schottky)
     + phibn / pow(cosh(W00/(P_K * Treal)), 2)))
        * exp(-P_Q * phibn / W0) * (exp(P_Q * fabs(V_schottky) / epsprime) - 1);
    } else { // Schottky TE RESET direction
        return A * Arichardson * pow(Treal, 2) * exp(-phibn * P_Q / (P_K * Treal)) * (exp(P_Q
     / (P_K * Treal) * V_schottky) - 1);
    }
}
```

UpdateResistance()

This methods calculates and updates the resistance variables for the disc region, plug region, and series region as described in equations 3.1, 3.10, and 3.11 respectively. Listing 3.5 and listing 3.6 show the equivalent implementations in Verilog-A and C++ respectively, where the aforementioned equations have been implemented without alterations.

Listing 3.5: Verilog-A implementation of the device internal resistances based on equations 3.1, 3.10, and 3.11

```
Rdisc=ldisc*1e-9/(Ndisc*1e26*zvo*`P_Q*un*A);
Rplug=((lcell-ldisc)*1e-9/(Nplug*1e26*zvo*`P_Q*un*A));
Rseries=RTiOx+R0*(1+R0*alphaline*pow(I(discplugserial),2)*Rthline);
```

Listing 3.6: C++ implementation of the device internal resistances based on equations 3.1, 3.10, and 3.11

```
void JART_VCM_v1b_var::UpdateResistance(double I_discplugserial) {
    Rdisc = ldisc * 1e-9 / (Ndisc * 1e26 * zvo * P_Q * un * A);
    Rplug = ((lcell - ldisc) * 1e-9 / (Nplug * 1e26 * zvo * P_Q * un * A));
    Rseries = RTiOx + R0 * (1 + R0 * alphaline * pow(I_discplugserial, 2) * Rthline);
}
```

UpdateConcentration()

This method calculates the change and updates the internal state variable as described in equation 3.2. Notably, this method does not include the calculation of the ionic current, which instead is done in the ‘ComputeIonCurrent()‘ method described in section 3.1.2. Listing 3.7 and listing 3.8 show the equivalent implementations in Verilog-A and C++ respectively.

According to equation 3.2, the difference in oxygen concentration between two points in time is calculated by integrating over that interval. In Verilog-A this is correctly implemented using the build in ‘idt()‘ function. In C++, no such function is available, thus a different method has to be used. One approach to find this integral is to determine the anti-derivative. However, from equations 3.2–3.9 it follows that the time-derivative of the oxygen concentration depends on the voltages in the device, which in turn change with time. This dependency is nested in various root and exponent operations, thus finding the anti-derivative for this function is not feasible.

Instead, the C++ implementation approximates the integral using the Euler method, as described in equation 3.20. This method assumes the voltages in the device remain constant in a given time step and assumes the time step is sufficiently small. To address this second assumption, the ’ApplyVoltage()’ method, discussed in section 3.1.2, includes a mechanism to dynamically reduce the size of the time step. Another downside of this method is that errors compound over time. However, this downside is remedied by a mechanism of the model where the oxygen concentration is bounded by a minimum and maximum value.

$$\Delta N_{disc} = -\frac{I_{ion}}{z v_o e A l_{disc}} \cdot \Delta t \tag{3.20}$$

Listing 3.7: Verilog-A implementation for updating the internal state variable based on equation 3.2

```
Nchange=idt(-trig/(A*ldisc*1e-9*`P_Q*zvo)*V(ion,gnd)/1e26,0);
Ndisc=Ninit + Nchange;
```

Listing 3.8: C++ implementation for updating the internal state variable based on equation 3.2

```
void JART_VCM_v1b_var::UpdateConcentration(double I_ion, double dt) {
    double Nchange = (-trig / (A * ldisc * 1e-9 * P_Q * zvo) * I_ion / 1e26) * dt;
    Ndisc += Nchange;
    if (Ndisc > Ndiscmax) { Ndisc = Ndiscmax; }
    else if (Ndisc < Ndiscmin) { Ndisc = Ndiscmin; }
}
```

ComputeIonCurrent()
This method calculates the the ionic current based on the equations 3.3–3.9. The result of this method will subsequently be used to update the oxygen concentration in the device, as described in section 3.1.2. Listing 3.9 and listing 3.10 show the equivalent implementations in Verilog-A and C++ respectively. These implementations are equivalent to the equations they are based on, except that the ionic current is set to zero if the oxygen concentration would grow outside its bounds.

Listing 3.9: Verilog-A implementation for calculating the ionic current based on equation 3.3–3.9

```
//I_ion is realised through a voltage source here
if (((Ndisc<Ndiscmin)&(V(AE,OE)>0))|((Ndisc>Ndiscmax)&(V(AE,OE)<0)))    // keep concentration
     Nreal in the borders of Ndiscmin and Ndiscmax
        begin
                V(ion,gnd)<+0;
                trig=0;
        end
else
        begin
                cvo = (Nplug+Ndisc)/2*1e26;
                if (V(AE,OE)>0)
                        begin
                                E_ion=(V(schottky)+V(discplugserial)*(Rdisc+Rplug)/(Rdisc+
                                    Rplug+Rseries))/(lcell*1e-9);
                                Rtheff = Rth0*pow(rdet/rdisc,2)*Rtheff_scaling;
                                Flim=1-pow(Ndiscmin/Ndisc,10);
                        end
                else
                        begin
                                E_ion=V(discplugserial)*Rdisc/(Rdisc+Rplug+Rseries)/(ldisc*1e
                                    -9);
                                Rtheff = Rth0*pow(rdet/rdisc,2);
                                Flim=1-pow(Ndisc/Ndiscmax,10);
                        end
                gamma=zvo*`P_Q*E_ion*a/(`M_PI*dWa*`P_Q);
                dWamin=dWa*`P_Q*(sqrt(1-pow(gamma,2))-gamma*`M_PI/2+gamma*asin(gamma));
                dWamax=dWa*`P_Q*(sqrt(1-pow(gamma,2))+gamma*`M_PI/2+gamma*asin(gamma));
                V(ion,gnd)<+zvo*`P_Q*cvo*a*nyo*A*(exp(-dWamin/(`P_K*Treal))-exp(-dWamax/(`P_K
                    *Treal)))*Flim;
        end
end
```

Listing 3.10: C++ implementation for calculating the ionic current based on equation 3.3–3.9

```
double JART_VCM_v1b_var::ComputeIonCurrent(double V_applied, double V_schottky, double
    V_discplugserial) {
    if ((Ndisc <= Ndiscmin && V_applied > 0) | (Ndisc >= Ndiscmax && V_applied < 0)) { //
    Keep concentration Nreal in the borders of Ndiscmin and Ndiscmax
        trig = 0;
        return 0;
    } else {
        double cvo = (Nplug + Ndisc) / 2. * 1e26;
```

```
        double E_ion;
        double Flim;
        if (V_applied > 0) {
            E_ion = (V_schottky + V_discplugserial * (Rdisc + Rplug) / (Rdisc + Rplug +
    Rseries)) / (lcell * 1e-9);
            Flim = 1 - pow(Ndiscmin/Ndisc, 10);
        } else {
            E_ion = V_discplugserial * Rdisc / (Rdisc + Rplug + Rseries) / (lvar * 1e-9);
            Flim = 1 - pow(Ndisc/Ndiscmax, 10);
        }
        double gamma = zvo * P_Q * E_ion * a / (M_PI * dWa * P_Q);
        double dWamin = dWa * P_Q * (sqrt(1 - pow(gamma, 2)) - gamma * M_PI/2 + gamma * asin(
    gamma));
        double dWamax = dWa * P_Q * (sqrt(1 - pow(gamma, 2)) + gamma * M_PI/2 + gamma * asin(
    gamma));
        return zvo * P_Q * cvo * a * nyo * A * (exp(-dWamin / (P_K * Treal)) - exp(-dWamax /
    (P_K * Treal))) * Flim;
    }
}
```

ApplyVoltage()

This method simulates a voltage being applied to the device and has no direct Verilog-A equivalent. It takes the voltage applied to the device and a delta time as input, updates the state variable, and returns the current flowing through the device. The mechanism by which this is performed is as follows:

First the current flowing through the device is calculated based on the applied voltage. This calculation does not have a closed form equation and instead uses an iterative solver which will be further discussed in section 3.1.3. If the delta time parameter is 0, the calculated current is returned and any later steps are skipped. Otherwise, the state variable and temperature are updated based on the applied voltage, calculated current, and the provided delta time.

Finally, the change in state variable is compared to a threshold value. If the change is smaller than this threshold value, the ‘ApplyVoltage()‘ function returns the current and exits. If the change is above the threshold, the change is considered too large to be applied in one time step. In this case, the state variable is reverted to it’s original value and the delta time is divided into multiple sub-steps. Then the ‘ApplyVoltage()‘ method is applied to each sub-step until the change in state variable is lower than the threshold, or until the delta time of the sub-step is below a minimum delta time threshold.

The purpose of this dynamic time-stepping behaviour is to increase the accuracy of the switching. Experimental results show that the state variable of this memristor model changes little until some threshold voltage is reached. After this threshold voltage, the state variable changes rapidly. By increasing the amount of time steps taken during this region, the fast switching behaviour is more accurately captured.

Fig. 3.2 shows a flow diagram of the described ‘ApplyVoltage()‘ method.

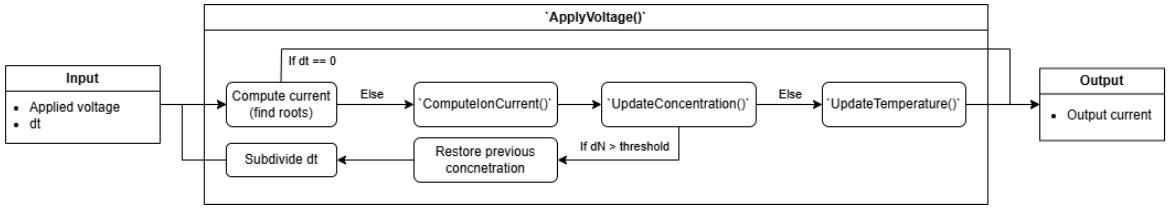


Figure 3.2: Flow diagram of the ‘ApplyVoltage()‘ method described in section 3.1.2

GetResistance()

This method calculates the resistance of the device for a specified applied voltage and has no direct Verilog-A equivalent. Since the resistance of the device is current dependent, this method makes use of the ‘ApplyVoltage()‘ method with a delta time of 0. The resistance is subsequently calculated by V/I. For small voltages this method can be unstable, in which case the sum of internal resistances is taken where the current is assumed to be 0. Fig. 3.1.2 shows a flow diagram of the described method.

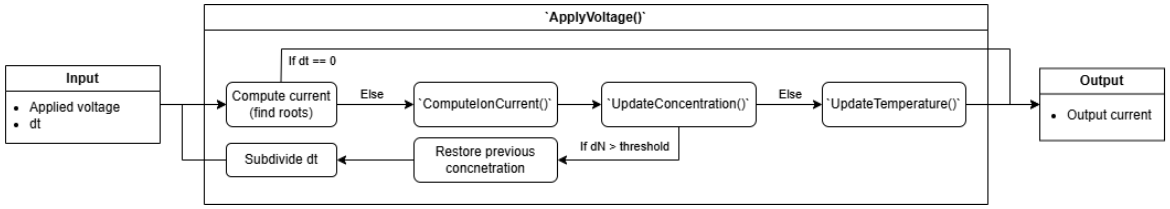


Figure 3.3: Flow diagram of the 'GetResistance()' method described in section 3.1.2

SetState()

This method sets the state to the device based on a provided boolean value and has no direct Verilog-A equivalent. A value of True means the device will be set to LRS, where $N_{disc} = N_{disc,max}$. A value of False means the device will be set to HRS, where $N_{disc} = N_{disc,min}$.

Inheritance

The 'ApplyVoltage()', 'GetResistance()', and 'SetState()' described in sections 3.1.2, 3.1.2, 3.1.2 act as the interface between the memristor model and the other simulation components. Subsequently, these three functions define the parent memristor class and different models can be implemented by inheriting from this parent. For any memristor model, the three functions described earlier are expected to do the following:

- 'ApplyVoltage()' expects a voltage and a delta time, and should return the current flowing through the device based on that voltage. Additionally, this function is generally used to update the state variable of the model
- 'GetResistance()' expects a voltage and should return the resistance of the device at that voltage. For linear devices, this input voltage can remain unused.
- 'SetState()' expects a boolean value and should set the state of the memristor based on that value. Gerenally, a value of True sets the memristor to the ON state or LRS, and a value of False is sets it to the OFF state or HRS

Listing [3.11] shows an example of a simple memristor model implementation. In this example, the memristor has either an ON or OFF resistance and does not change state. Instead the resistance can only be changed through the 'SetState()' function. Effectively, this example device is a resistor.

Listing 3.11: Example of a memristor model implementation inheriting from the memristor parent class

```
class ExampleMemristor: public Memristor {
public:
    const float Ron = 100.;
    const float Roff = 10000.;
    float R;

    ExampleMemristor() {  R = Roff; }

    double ApplyVoltage(double V_applied, double dt) override {
        return V_applied / R;
    };
    double GetResistance(double V_applied) {
        return R;
    };
    void SetWeight(bool weight) {
        if (weight) { R = Ron; }
        else { R = Roff; }
    };
};
```

3.1.3. Find Roots

As mentioned in section 3.1.1, there is no direct way to calculate the Schottky voltage from the applied voltage. Instead, this value can be determined through the voltage across the other components.

Solving this is akin to finding the roots of the following equation:

$$F(V_{Schottky}) = V_{applied} - V_{Schottky} - V_{disc+plug+series} \tag{3.21}$$

Where $V_{disc+plug+series}$ is described as in equation 3.22.

$$V_{disc+plug+series} = [R_{disc} + R_{plug} + R_{series}(I(V_{Schottky}))] * I(V_{Schottky}) \tag{3.22}$$

Fig. 3.4a, Fig. 3.4b, and Fig. 3.4c show three graph of this equation as generated by Desmos with $V_{applied} = 1V$, $V_{applied} = 0.5V$, and $V_{applied} = 0.1V$ respectively. While the exact graph changes based on input, state parameters, and tuning parameters, the shape remains largely the same.

From those graphs, it can be seen that equation 3.21 has at most 3 roots. Each root corresponds to a Schottky voltage such that the voltages across all subcomponents in the memristor model add up to $V_{applied}$. Without any additional considerations, each of there roots are equally possible.

In reality however, the memristor will have had some history. The Schottky voltage will not have jumped instantly to a different value, thus one of the roots can be chosen based on that history. The correct Schottky value for the purposes of simulation will be the root that is closest to the value calculated in the previous time step.

In order to find this root, an iterative solver is used based on Newton's method. Such a solver will only find the closest root based on some initial guess and can be accelerated if that initial guess is close. The exact iterative solver is derived as follows:

First, a general equation for an iterative solver is formulated as in equation 3.23.

$$x_{n+1} = x_n - g(x_n) \cdot f(x_n) \tag{3.23}$$

Where $f(x_n)$ is the function for which we want to find the roots. In Newton's method, $g(x_n)$ would be defined as in equation 3.24.

$$g(x_n) = \frac{1}{f'(x_n)} \tag{3.24}$$

The derivative for equation 3.21 is described in equation 3.25

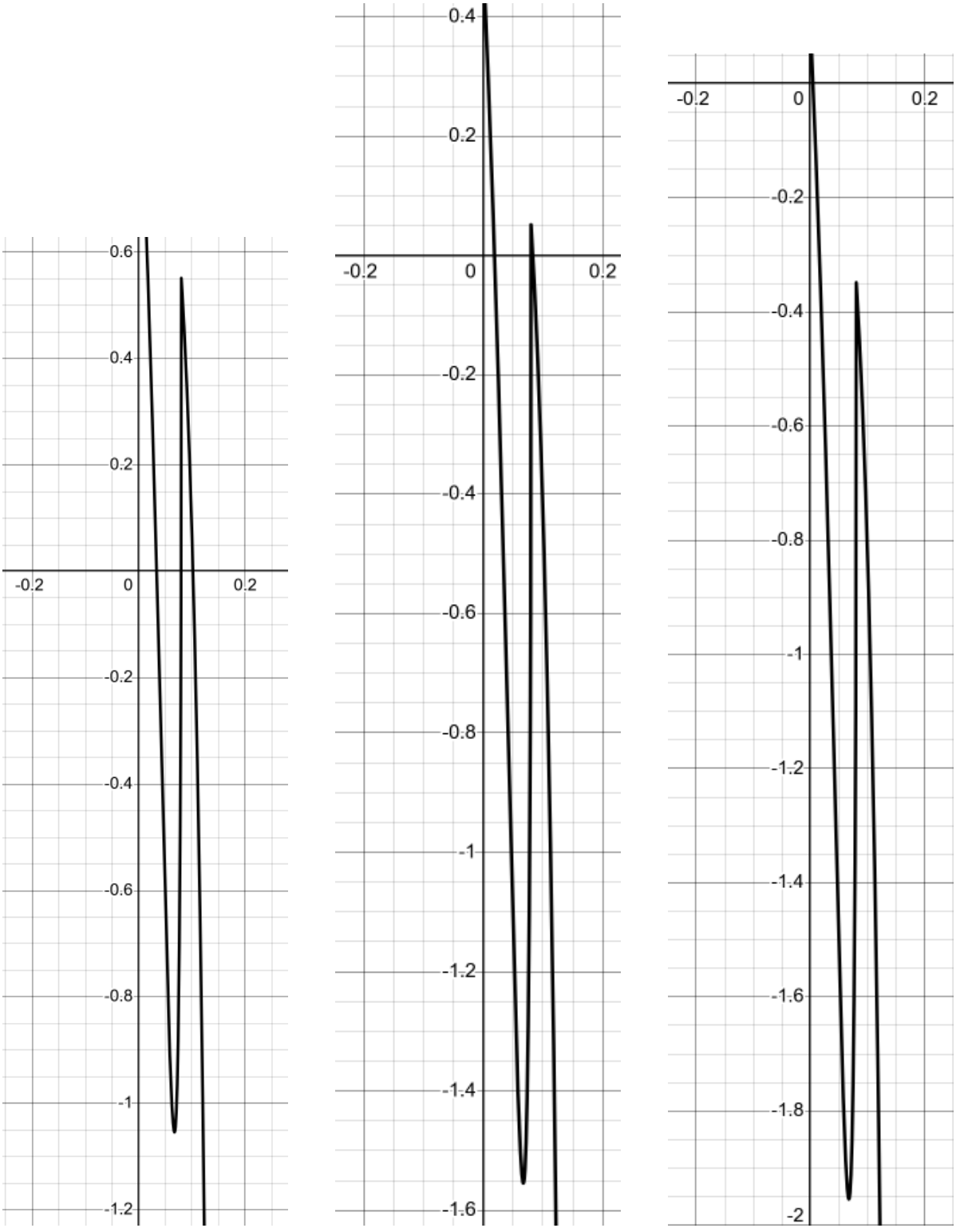
$$\frac{d}{dV_{Schottky}} F(V_{Schottky}) = -1 - \frac{d}{dV_{Schottky}} [R_{disc} + R_{plug} + R_{series}(I(V_{Schottky}))] * I(V_{Schottky}) \tag{3.25}$$

Unfortunately, $I(V_{Schottky})$ is a piece-wise complex function involving nested exponential and root functions, making its derivative difficult to compute directly. Furthermore, the guards added to ϕ_{Bn} described section 3.1.2 introduce additional piecewise behaviour and ensures the derivative is not continuous everywhere. At the time of writing, functional implementation of this derivative has not been achieved, thus is instead approximated as 0 to simplify the solver's implementation. The derived derivation can be found in appendix A. The consequence of the approximation reduces the Newton update effectively to a fixed-slope form, resulting in $g(x_n) = -1$. From this follows the iterative equation described in equation 3.26.

$$x_{n+1} = x_n + f(x_n) \tag{3.26}$$

The approximation of the derivative reduces computational complexity while maintaining the accuracy of the final solution. However, since the steps in each iteration are less optimized compared to the full Newton's method, this may increase convergence time. A good initial guess can help mitigate this effect by reducing the number of iterations needed for convergence.

One final property of equation 3.21 should be discussed before the derived iterative equation can be used. Namely, equation 3.21 is unstable: a small change in the input leads to a large change in the output. Were the iterative solver to be use as is, this could lead to oscillations which could grow to infinity. One possible solution to mitigate this is to multiply $f(x_n)$ with a damping function. Ideally, the damping function reduces when $f(x_n)$ is large and increases as $f(x_n)$ approaches zero. Alternatively,

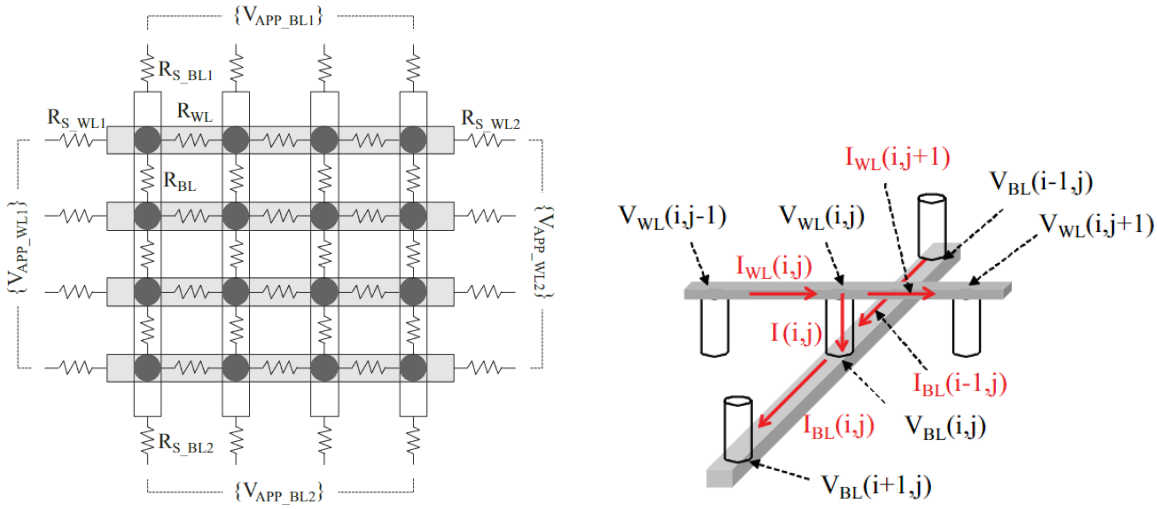


(a) Graph of equation 3.21 with $V_{applied} = 1V$ made with Desmos

(b) Graph of equation 3.21 with $V_{applied} = 0.5V$ made with Desmos

(c) Graph of equation 3.21 with $V_{applied} = 0.1V$ made with Desmos

Figure 3.4: Desmos graphs of equation 3.21 using various $V_{applied}$ values



(a) Crossbar circuit diagram [32]

(b) Crossbar node circuit diagram [32]

Figure 3.5: Crossbar Circuit diagrams

the damping function could increase with each solver iteration. Or the damping function could be taken as constant. For this simulation, a simple damping function as described by equation 3.27 was selected through experimental evaluation.

$$a = \begin{cases} 5e-3 & if \text{ it} \leq 100 \\ 5e-2 & otherwise \end{cases} \tag{3.27}$$

Where 'it' is the iteration number of the iterative solver.

3.2. Linear Crossbar Solver

In the context of computer memory, a crossbar refers to a grid of intersecting horizontal and vertical lines, called wordlines and bitlines respectively. These lines are not directly connected to a the intersections, but are connected through some memory device instead. For the purposes of this thesis, these devices are memristive devices, such as described in section 3.1.

Ideally, the crossbar lines connects these devices without altering incoming and outgoing signals in any way. In practice, however, some conductive material has to be used to form these lines, which exhibit electrical properties that have to be considered. This thesis will limit the simulation of these properties to resistance effects only.

The resistances of crossbar lines can have significantly impact the behaviour of a memristive crossbar. Specifically, the existence resistance in between two nodes causes voltages drops, also called IR-drop. Consequently, nodes connected to the same line will not be at the same voltage, and nodes further away from voltage sources will experience a larger drop. This uneven voltage distribution can in certain crossbar configurations induce sneak path currents: unintended currents that flow in alternate or reverse routes through memory devices and would not be present with no line resistance.

The remainder of this section describes and implements a model for a resistive crossbar. In this model, line resistances are modeled as resistive components in between nodes, and the memory devices are similarly replaced by resistors. Fig. 3.5a shows a circuit diagram of the described crossbar model.

3.2.1. Crossbar Model Description

The goal of the linear crossbar model is to calculate the voltage at every node based on the input voltages and the crossbar configuration. Simulation programs, such as SPICE, solve this problem by applying applying Kirchhoff's current law to every node in the network, assembling the subsequent equations into a matrix form, and solving the resulting system. While very useful for general networks, the analysis required for every node can be quite slow and is not necessary for this application. In-

stead, the fixed and repeating structure of a resistive crossbar can be exploited to determine the matrix equation in advance. During simulation, the relative matrix elements merely have to be filled in before the system can be solved.

This thesis uses a model developed by A. Chen [32] for simulation. This method is able to losslessly transform a resistive crossbar, as described in Fig. 3.5a, into an equation of the form $G * V = E$. This system can subsequently be solved using numerical or iterative approaches to obtain the nodal voltages V. The derivation for this transformation is as follows:

First Kirchhoff's current law is applied at every wordline and bitline node, where a node is defined as in Fig. 3.5b. This leads to equations 3.28 and 3.29.

$$I_{WL}(i,j) = I(i,j) + I_{WL}(i,j+1) \tag{3.28}$$

$$I_{BL}(i,j) = I(i,j) + I_{BL}(i-1,j) \tag{3.29}$$

This can be rewritten in terms of voltages to obtain equations 3.30, 3.31, 3.32, 3.33, 3.34, and 3.35. Note that the previous two equations expand into six equations due to cases where a node is at the edge of the crossbar.

$$\frac{V_{WL}(i,j-1) - V_{WL}(i,j)}{R_{WL}} = \frac{V_{WL}(i,j) - V_{BL}(i,j)}{R(i,j)} + \frac{V_{WL}(i,j) - V_{WL}(i,j+1)}{R_{WL}}$$
$$\text{for } 1 \leq i \leq m \text{ and } 2 \leq j \leq n-1 \tag{3.30}$$

$$\frac{V_{APP\_WL1}(i) - V_{WL}(i,j)}{R_{S\_WL1}} = \frac{V_{WL}(i,j) - V_{BL}(i,j)}{R(i,j)} + \frac{V_{WL}(i,j) - V_{WL}(i,j+1)}{R_{WL}}$$
$$\text{for } 1 \leq i \leq m \text{ and } j = 1 \tag{3.31}$$

$$\frac{V_{WL}(i,j-1) - V_{WL}(i,j)}{R_{WL}} = \frac{V_{WL}(i,j) - V_{BL}(i,j)}{R(i,j)} + \frac{V_{WL}(i,j) - V_{APP\_WL2}(i)}{R_{S\_WL2}}$$
$$\text{for } 1 \leq i \leq m \text{ and } j = n \tag{3.32}$$

$$\frac{V_{BL}(i,j) - V_{BL}(i+1,j)}{R_{BL}} = \frac{V_{WL}(i,j) - V_{BL}(i,j)}{R(i,j)} + \frac{V_{BL}(i-1,j) - V_{BL}(i,j)}{R_{BL}}$$
$$\text{for } 2 \leq i \leq m-1 \text{ and } 1 \leq j \leq n \tag{3.33}$$

$$\frac{V_{BL}(i,j) - V_{BL}(i+1,j)}{R_{BL}} = \frac{V_{WL}(i,j) - V_{BL}(i,j)}{R(i,j)} + \frac{V_{APP\_BL1}(j) - V_{BL}(i,j)}{R_{S\_BL1}}$$
$$\text{for} i = 1 \text{ and } 1 \leq j \leq n \tag{3.34}$$

$$\frac{V_{BL}(i,j) - V_{APP\_BL2}(j)}{R_{S\_BL2}} = \frac{V_{WL}(i,j) - V_{BL}(i,j)}{R(i,j)} + \frac{V_{BL}(i-1,j) - V_{BL}(i,j)}{R_{BL}}$$
$$\text{for } i = m \text{ and } 1 \leq j \leq n \tag{3.35}$$

Next, these equations are rewritten into a more convenient form by grouping voltages together. Additionally, equations 3.39, 3.40, and 3.41 are multiplied by -1 to make the final G matrix symmetric. This decision aids in simulating the transformed crossbar model and will be further discussed in section 3.2.2. The result of this step leads to equations 3.36, 3.37, 3.38, 3.39, 3.40, and 3.41.

$$V_{WL}(i,j-1) \cdot \frac{-1}{R_{WL}} + V_{WL}(i,j) \cdot (\frac{1}{R(i,j)} + \frac{2}{R_{WL}}) + V_{WL}(i,j+1) \cdot \frac{-1}{R_{WL}} + V_{BL}(i,j) \cdot \frac{-1}{R(i,j)} = 0$$
$$\text{for } 1 \leq i \leq m \text{ and } 2 \leq j \leq n-1 \tag{3.36}$$

$$V_{WL}(i,j)\cdot(\frac{1}{R_{S-WL1}}+\frac{1}{R(i,j)}+\frac{1}{R_{WL}})+V_{WL}(i,j+1)\cdot\frac{-1}{R_{WL}}+V_{BL}(i,j)\cdot\frac{-1}{R(i,j)}=V_{APP-WL1}(i)\cdot\frac{1}{R_{S-WL1}}$$
for $1 \leq i \leq m$ and $j = 1$ (3.37)

$$V_{WL}(i,j-1)\cdot\frac{-1}{R_{WL}}+V_{WL}(i,j)\cdot(\frac{1}{R_{S-WL2}}+\frac{1}{R(i,j)}+\frac{1}{R_{WL}})+V_{BL}(i,j)\cdot\frac{-1}{R(i,j)}=V_{APP-WL2}(i)\cdot\frac{1}{R_{S-WL2}}$$
for $1 \leq i \leq m$ and $j = n$ (3.38)

$$V_{WL}(i,j)\cdot\frac{-1}{R(i,j)}+V_{BL}(i-1,j)\cdot\frac{-1}{R_{BL}}+V_{BL}(i,j)\cdot(\frac{1}{R(i,j)}+\frac{2}{R_{BL}})+V_{BL}(i+1,j)\cdot\frac{-1}{R_{BL}}=0$$
for $2 \leq i \leq m-1$ and $1 \leq j \leq n$ (3.39)

$$V_{WL}(i,j)\cdot\frac{-1}{R(i,j)}+V_{BL}(i,j)\cdot(\frac{1}{R_{S-BL1}}+\frac{1}{R(i,j)}+\frac{1}{R_{BL}})+V_{BL}(i+1,j)\cdot\frac{-1}{R_{BL}}=V_{APP-BL1}\cdot\frac{1}{R_{S-BL1}}$$
for$i = 1$ and $1 \leq j \leq n$ (3.40)

$$V_{WL}(i,j)\cdot\frac{-1}{R(i,j)}+V_{BL}(i-1,j)\cdot\frac{-1}{R_{BL}}+V_{BL}(i,j)\cdot(\frac{1}{R_{S-BL2}}+\frac{1}{R(i,j)}+\frac{1}{R_{BL}})=V_{APP-BL2}\cdot\frac{1}{R_{S-BL2}}$$
for $i = m$ and $1 \leq j \leq n$ (3.41)

Each of these equations is centered around a node (i, j) and references the voltages of adjacent nodes as well as the resistances in between. For equations 3.36, 3.37, and 3.38, the central node is a wordline node. Consequently, the adjacent nodes are two more wordline nodes, where one can be a source node, and one bitline node. For equations 3.39, 3.40, and 3.41, the central node is a bitline node, and the adjacent nodes are consequently two other bitline nodes, where similarly one can be a source node, and one wordline node.

Since each equation is centered around one node, the complete system of equations describing a crossbar consists of $2 \cdot M \cdot N$ equations for an $M \times N$ crossbar. Furthermore, by ordering these equations by their central voltage, the system can be equated to the matrix equation described earlier:

$$G \cdot V = E \tag{3.42}$$

The ordering used for this is a row-wise flattening of all wordline voltages followed by a row-wise flattening of all bitline voltages, or as described in equation 3.43.

$$\begin{aligned} V = [&V_{WL}(1,1)\cdots V_{WL}(1,n),\cdots,V_{WL}(m,1)\cdots V_{WL}(m,n) \\ &V_{BL}(1,1)\cdots V_{BL}(1,n),\cdots,V_{BL}(m,1)\cdots V_{BL}(m,n)]^T \end{aligned} \tag{3.43}$$

The remaining two elements of equation 3.42 are matrix G and vector E. For an $M \times N$ crossbar, G is an $2MN \times 2MN$ matrix containing all coefficients of the wordline and bitline voltages. Similarly, for an $M \times N$ crossbar E is an $2MN$ element vector containing all coefficients related to the sources.

To make its construction more clear, the G matrix can be subdivided into four matrices of identical size, as in equation 3.44.

$$G = G_{ABCD} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \tag{3.44}$$

Submatrix A contains the coefficients for the wordline voltages for the wordline Kirchhoff equations, where the wordline Kirchhoff equations are 3.36, 3.37, and 3.38. From that observation, submatrix A can be described as in equation 3.45.

$$A = \begin{bmatrix} A_1 & 0 & 0 & \cdots & 0 \\ 0 & A_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \cdots & A_M \end{bmatrix} \tag{3.45}$$

Where A_i is described by equation 3.46.

$$A_i = \begin{bmatrix} \frac{1}{R_{S-WL1}} + \frac{1}{R(i,1)} + \frac{1}{R_{WL}} & \frac{-1}{R_{WL}} & 0 & \cdots & 0 \\ \frac{-1}{R_{WL}} & \frac{1}{R(i,2)} + \frac{2}{R_{WL}} & \frac{-1}{R_{WL}} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \frac{-1}{R_{WL}} \\ 0 & \cdots & 0 & \frac{-1}{R_{WL}} & \frac{1}{R_{S-WL2}} + \frac{1}{R(i,N)} + \frac{1}{R_{WL}} \end{bmatrix} \quad for\ 1 \leq i \leq M \tag{3.46}$$

Submatrix B contains the coefficients for the bitline voltages of the wordline Kirchhoff equations, where the wordline Kirchhoff equations are 3.36, 3.37, and 3.38. From that observation, submatrix B can be described as in equation 3.47.

$$B = \begin{bmatrix} B_1 & 0 & 0 & \cdots & 0 \\ 0 & B_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \cdots & B_M \end{bmatrix} \tag{3.47}$$

Where B_i is described by equation 3.48.

$$B_i = \begin{bmatrix} \frac{-1}{R(i,1)} & 0 & 0 & \cdots & 0 \\ 0 & \frac{-1}{R(i,2)} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \cdots & \frac{-1}{R(i,N)} \end{bmatrix} \quad for\ 1 \leq i \leq M \tag{3.48}$$

Submatrix C contains the coefficients for the wordline voltages of the bitline Kirchhoff equations, where the bitline Kirchhoff equations are 3.39, 3.40, 3.41. From that observation, submatrix C can be described as in equation 3.49.

$$C = \begin{bmatrix} C_1 & 0 & 0 & \cdots & 0 \\ 0 & C_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \cdots & C_M \end{bmatrix} \tag{3.49}$$

Where C_i is described by 3.50

$$C_i = \begin{bmatrix} \frac{-1}{R(i,1)} & 0 & 0 & \cdots & 0 \\ 0 & \frac{-1}{R(i,2)} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \cdots & \frac{-1}{R(i,N)} \end{bmatrix} \quad for\ 1 \leq i \leq M \tag{3.50}$$

More simply, submatrix C can be described as $C = B$.

Submatrix D contains the coefficients for the bitline voltages of the bitline Kirchhoff equations, where the bitline Kirchhoff equations are 3.39, 3.40, 3.41. From that observation, submatrix D can be described as in equation 3.51.

$$\begin{cases} \begin{cases} D(i*N+j, i*N+j) = \frac{1}{R_{S-BL1}} + \frac{1}{R(i,j)} + \frac{1}{R_{BL}} \\ D(i*N+j, (i+1)*N+j) = \frac{-1}{R_{BL}} \end{cases} & for\ i = 1\ and\ 1 \leq j \leq N \\ \begin{cases} D(i*N+j, (i-1)*N+j) = \frac{-1}{R_{BL}} \\ D(i*N+j, i*N+j) = \frac{1}{R(i,j)} + \frac{2}{R_{BL}} \\ D(i*N+j, (i+1)*N+j) = \frac{-1}{R_{BL}} \end{cases} & for\ 2 \leq i \leq M-1\ and\ 1 \leq j \leq N \\ \begin{cases} D(i*N+j, i*N+j) = \frac{1}{R_{S-BL2}} + \frac{1}{R(i,j)} + \frac{1}{R_{BL}} \\ D(i*N+j, (i-1)*N+j) = \frac{-1}{R_{BL}} \end{cases} & for\ i = M-1\ and\ 1 \leq j \leq N \\ 0\ otherwise \end{cases} \tag{3.51}$$

Since all these submatrices are of identical size, for an $M \times N$ crossbar, G will be an $2MN \times 2MN$ matrix, and submatrices A, B, C, and D will each be an $MN \times MN$ matrix.

The final part of equation 3.44 is vector E, which relates to the sources at the boundaries of the crossbar. Vector E can be split into two subvectors of identical size, as in equation 3.52.

$$E = \begin{bmatrix} E_W \\ E_B \end{bmatrix} \tag{3.52}$$

Subvector E_w contains coefficients related to the wordline sources, and can be described as in equation 3.53.

$$E_W = \begin{bmatrix} E_{W_1} \\ E_{W_2} \\ \vdots \\ E_{W_m} \end{bmatrix} \tag{3.53}$$

Where E_{W_i} is described by equation 3.54.

$$E_{W_i} = \begin{bmatrix} \frac{V_{app-WL1}(i)}{R_{S-WL1}} \\ 0 \\ 0 \\ \vdots \\ \frac{V_{app-WL2}(i)}{R_{S-WL2}} \end{bmatrix} \; for\ 1 \leq i \leq M \tag{3.54}$$

Subvector E_b contains coefficients related to the bitline sources, and can be described as equation 3.55.

$$E_B = \begin{bmatrix} \frac{V_{app-BL1}(1)}{R_{S-BL1}} \\ \frac{V_{app-BL1}(2)}{R_{S-BL1}} \\ \vdots \\ \frac{V_{app-BL1}(N)}{R_{S-BL1}} \\ 0 \\ \vdots \\ 0 \\ \frac{V_{app-BL2}(1)}{R_{S-BL2}} \\ \frac{V_{app-BL2}(2)}{R_{S-BL2}} \\ \vdots \\ \frac{V_{app-BL2}(N)}{R_{S-BL2}} \end{bmatrix} \; such\ that\ E_B\ has\ M \cdot N\ elements \tag{3.55}$$

Since E_W and E_B are of identical size, for an $M \times N$ crossbar, E will be a vector of size $2MN$, and subvectors E_W and E_B will each be a vector of size MN.

A 3×3 example of this crossbar model is provided in appendix B

3.2.2. Crossbar Model Implementation

The transformation of the crossbar model described in section 3.2.1 results in an equation of the form $G_{ABCD} \cdot V = E$, where vector V is the unknown. However, the size of matrix G_{ABCD} scales quadratically with crossbar size, making it impractical to solve for V using direct analytical methods such as matrix inversion. Instead, G_{ABCD} has some useful properties that make it well-suited for a certain iterative solver, namely the conjugate gradient method. This method is an iterative algorithm designed to solve systems of linear equations and is commonly used to solve large sparse systems. However, it requires that the system matrix is positive semidefinite.

The equations for submatrices A, B, C, and D show that G_{ABCD} is both a sparse and symmetric matrix. Additionally, it can be seen that entries on the diagonal elements of G_{ABCD} are always non-negative, while off-diagonal elements are always non-positive. Moreover, the absolute value of every individual

element in a row is never larger than the absolute value of the corresponding diagonal element, making it a diagonally dominant matrix. A symmetric and diagonally dominant matrix is by necessity also positive semidefinite, which makes the conjugate gradient method applicable to this system.

The conjugate gradient solver, as well as the transformation of the crossbar into the matrix equation, is implemented using the C++ Eigen library [39] and its constructs for matrices, vectors, and solvers. First, the submatrices A, B, C, and D are constructed in their corresponding locations in the G_{ABCD} matrix. Next, vector E is constructed. Finally, the system is solved for V using the sparse conjugate gradient solver build into Eigen. The result is a 'solve' function that takes in the relevant input-, device-, and crossbar parameters, and returns the calculated nodal voltages V. Additionally, this function can be supplied with an initial guess for V, which can be used by the conjugate gradient solver to possibly speed up convergence.

As will be seen in section 3.3, the linear crossbar solver is used repeatedly during simulation. Therefore, the function described in this section benefits greatly from any improvements to execution time. The first of such improvements is using more specialized constructs for the matrices, vectors, and solvers. Eigen differentiates between Dense and Sparse structures and this function greatly benefits from the sparse-specific optimizations. Secondly, it can be noted that large parts of the G_{ABCD} matrix do not depend on device parameters. Thus, it can be partially precomputed, since crossbar parameters do not change. Similarly, the E vector exclusively consists of input related parameters. This it can also be precomputed for a given input configuration.

Fig. 3.6 shows a flow diagram of the described implementation.

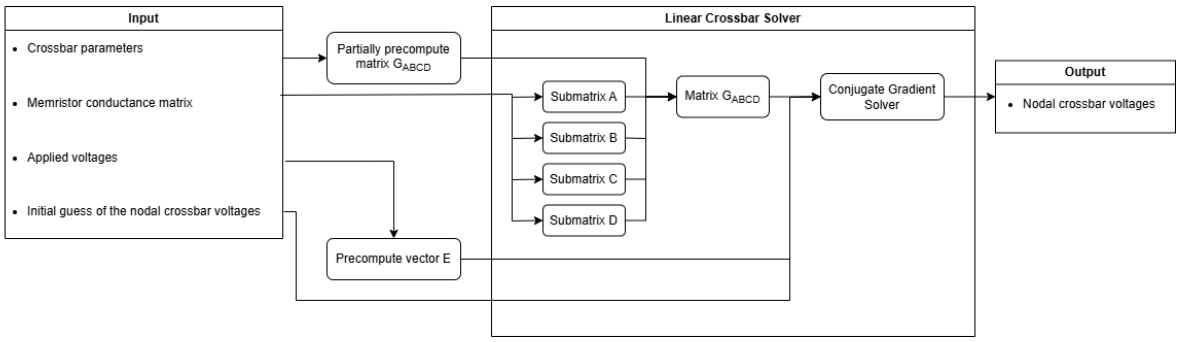


Figure 3.6: Flow diagram of the linear crossbar solver described in section 3.2.2

3.3. Non-linear crossbar solver

The two models discussed so far form the foundation of the crossbar simulator. However, these two models can not directly be used together. The linear crossbar model requires the devices to be linear, while the memristor model is not a linear device. In order to solve a crossbar with non-linear devices, an outer solver is necessary to bridge these differences.

3.3.1. Implementation

The non-linear crossbar solver combines the memristor model, from section 3.1, and the linear crossbar model, from section 3.2, by applying an outer iterative solver. This iterative solver makes some initial guess of the nodal voltages V in the crossbar and progressively improves this guess to be closer to their true values. For a given guess, the solver operates as follows:

First, the 'GetResistance()' method is used to determine the resistance of each individual memristor based on the current voltage guess. These values are then collected into a matrix where each element corresponds to the resistance of one memristor.

Next, a crossbar is solved where the memristors are replaced by their calculated resistances. For this, the linear crossbar model is used to calculated the nodal voltages as if the memristors had those resistance values. The current guess for the nodal voltages can be used to possibly speed up the execution of the linear solver.

Finally, the calculated voltages are compared to the current guess. If they are equal, or within a certain error margin, the guess can be assumed to be the true value and the non-linear crossbar is considered solved. If they are not equal, the guess is updated and the previous steps are repeated for the new guess.

Fig. 3.7 shows a flow diagram of the described implementation.

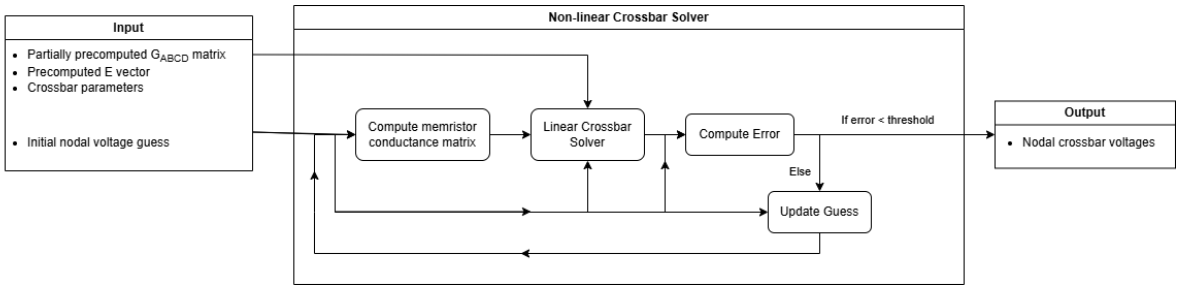


Figure 3.7: Flow diagram of the linear crossbar solver described in section 3.3.1

The method for updating the guess each iteration is derived similarly as in section 3.1.3, except here a vector form is used. First, a general equation for an iterative solver is formulated as in equation 3.56.

$$\overrightarrow{X}_{n+1} = \overrightarrow{X}_n - H(\overrightarrow{X}_n) \cdot F(\overrightarrow{X}_n) \tag{3.56}$$

Here, $F(\overrightarrow{X})$ is defined as the current guess minus the results of the linear crossbar model.

In a conventional Newton's method, $H(\overrightarrow{X})$ is the inverse of the Jacobian of $F(\overrightarrow{X})$. However, as is the case for this memristor model, it is unfeasible to derive this Jacobian matrix. Thus $H(\overrightarrow{X})$ will be approximated by the identity matrix. This reduces the equation to the form described in equation 3.57.

$$\overrightarrow{X}_{n+1} = \overrightarrow{X}_n - F(\overrightarrow{X}_n) \tag{3.57}$$

An optional damping factor could be added similar to the iterative solver for the memristor model. However, experimental evaluation showed a damping factor of 1 to be sufficient.

3.4. Combined Crossbar Simulator

The three modules discussed in this chapter are all that is necessary to simulate a memristive crossbar. The linear crossbar solver simulates the crossbar as if the memristors are linear devices. The non-linear crossbar solver extends this to allow for non-linear devices. And the memristor model provides the non-linear behaviour of the memristors as well as simulating their internal state. The final result being a 'CrossbarSimulator' class which combines these functionalities into a single interface.

3.4.1. Implementation

The goal of the 'CrossbarSimulator' class is to provide an abstraction layer between the underlying crossbar simulation logic and an end-user. Upon creation, the class initializes all crossbar parameters, creates a matrix of memristors initialized with default values, and precomputes the G_{ABCD} matrix as described in section 3.2.2. Each parameter can be accessed and modified with the appropriate fields. However, changing crossbar parameters change the G_{ABCD} matrix, thus instead a 'SetCrossbarParameters()' function is provided.

The 'CrossbarSimulator' class provides an interface consisting of the following methods:

CrossbarSimulator()

Listing 3.12: Function declaration of the 'CrossbarSimulator' constructor

```
1     CrossbarSimulator(int M, int N);
```

The constructor of the 'CrossbarSimulator' initializes the crossbar and all relevant parameters and structures. Memristors are not initialized by this function. Instead, an empty matrix is created which

later is filled in by the ‘Initialize()‘ function. This additional function call is required before the crossbar simulator can be used.

The constructor requires the crossbar size to be provided as two integers. Listing 3.12 shows the function declaration of the constructor.

Initialize()

Listing 3.13: Funcion declaration of the ‘Initialize()‘ method

```
1    template <typename MemristorType>
2    void Initialize();
```

This method initializes the memristors of the provided type into the crossbar array. It is required to call this function before the crossbar simulator can be used.

Listing 3.13 shows the function declaration of this method.

SetCrossbarParameters()

Listing 3.14: Function declaration of the ‘SetCrossbarParameters()‘ method

```
1    void SetCrossbarParameters(float Rswl1, float Rswl2, float Rsbl1, float Rsbl2, float Rwl,
         float Rbl);
```

This method acts as the SET method for all line and source resistances. Due to the crossbar simulator using partially precomputed structures for some if its methods, altering line resistances directly does not result in correct simulation. Instead, each line resistance can be provided as a float value, and this method ensures the proper correction of internal structures.

Listing 3.14 shows the function declaration of this method.

SetRRAM()

Listing 3.15: Function declaration of the ‘SetRRAM()‘ method

```
1    void SetRRAM(std::vector<std::vector<bool>> weights);
```

This method takes in a matrix of boolean values and sets the memristors to their ’ON’ or ’OFF’ state. Notably, this function does not simulate a write operation. Instead it calls the memristor’s ‘SetState()‘ function, which in turn sets the state variable appropriately.

Listing 3.15 shows the function declaration of this method.

SetAccessTransistors()

Listing 3.16: Function declaration of the ‘SetAccessTransistors()‘ method

```
1    void SetAccessTransistors(std::vector<bool> gate_lines);
```

The ‘CrossbarSimulator‘ class provides a simple extension for access transistors. Instead of just a singe memristor, each cell in the crossbar has an additional transistor to allow the memristor to be disconnected from the rest of the crossbar. These access transistors are simulated using boolean values, meaning that ’OFF’ is simulated as infinite resistance, and ’ON’ is simulated as no resistance.

This method provides functionality to set these access transistors. It takes a vector of booleans as inputs, where each element corresponds to a row of the crossbar. Subsequently, every access transistor in that row is set to that boolean value. Alternatively, access transistors can be set without the use of this function by writing to the appropriate fields.

Listing 3.16 shows the function declaration of this method.

LinearSolve()

Listing 3.17: Function declaration of the ‘LinearSolve()‘ method

```
1    Eigen::VectorXf LinearSolve(
2        Eigen::VectorXf Vguess,
3        const Eigen::VectorXf& Vappwl1, const Eigen::VectorXf& Vappwl2,
4        const Eigen::VectorXf& Vappbl1, const Eigen::VectorXf& Vappbl2
5    );
```

This method calls the linear crossbar simulator described in section 3.2 and calculates the nodal voltages for the given input voltages. As discussed in section 3.2.2, it first precomputes the E vector before starting the linear solver. Notably, this does not simulate applying a voltage to the individual memristors; the timestep for any 'memristor.ApplyVoltage()' call is taken as 0.

As described in section 3.1.2, memristors are implemented as non-linear devices: the resistance is depended on the applied voltage. This method approximates linearity by taking an operating point around which the memristor is assumed to be linear. Possible operating points include 0V, supply voltage, or a user provided voltage through the 'Vguess' input parameter. This can be configured in the 'simulation_settings.h' file, as described in section 3.4.3.

This method takes in five input parameters. 'Vguess' is a vector containing one element for each node in the crossbar. It can be used as a user defined operating point for memristors and is used as the starting point for the linear crossbar solver. Alternatively, the vector can be left as all 0. 'Vappwl1', 'Vappwl2', 'Vappbl1', and 'Vappbl2' are vectors corresponding to the source voltages for the left, right, top, and bottom edge of the crossbar respectively. During conventional operation, 'Vappwl1' contains all wordline sources, 'Vappbl2' is set to ground by setting all elements to 0, and 'Vappwl2' and 'Vappbl1' are unused. By default, the source resistances for 'Vappwl2' and 'Vappbl1' are set to infinity, leaving these sources disconnected.

Listing 3.17 shows the function declaration for this method.

NonlinearSolve()

Listing 3.18: Function declaration of the 'NonlinearSolve()' method

```
1    Eigen::VectorXf NonlinearSolve(
2        Eigen::VectorXf Vguess,
3        const Eigen::VectorXf& Vappwl1, const Eigen::VectorXf& Vappwl2,
4        const Eigen::VectorXf& Vappbl1, const Eigen::VectorXf& Vappbl2
5    );
```

This method calls the non-linear crossbar simulator described in section 3.3 and calculates the nodal voltages for the given input voltages. As discussed in section 3.2.2, it first precomputes the E vector before starting the non-linear solver. Notably, this does not simulate applying a voltage to the individual memristors; the timestep for any 'memristor.ApplyVoltage()' call is taken as 0.

Similar to 'LinearSolve()', this method takes in five input parameters. 'Vguess' is a vector containing one element for each node in the crossbar and is used as the starting point for the non-linear crossbar solver. Alternatively, the vector can be left as all 0. 'Vappwl1', 'Vappwl2', 'Vappbl1', and 'Vappbl2' are vectors corresponding to the source voltages for the left, right, top, and bottom edge of the crossbar respectively. During conventional operation, 'Vappwl1' contains all wordline sources, 'Vappbl2' is set to ground by setting all elements to 0, and 'Vappwl2' and 'Vappbl1' are unused. By default, the source resistances for 'Vappwl2' and 'Vappbl1' are set to infinity, leaving these sources disconnected.

Listing 3.18 shows the function declaration for this method.

CalculateIout()

Listing 3.19: Funcion declaratino of the 'CalculateIout()' method

```
1    std::vector<float> CalculateIout(const Eigen::VectorXf& Vout);
```

This method calculates the output current of the crossbar based on given nodal voltages. It first calculates the current flowing through each individual memristors using the 'memristor.ApplyVoltage()' method with a timestep of 0. Next, the output currents are accumulated into a vector where each element corresponds to the output current of one column of the crossbar. The function is intended to be used with the output of the 'LinearSolve()' and 'NonlinearSolve()' methods.

Listing 3.19 shows the function declaration for this method.

ApplyVoltage()

Listing 3.20: Function declaration of the ‘ApplyVoltage()‘ method

```
std::vector<std::vector<float>> ApplyVoltage(
    Eigen::VectorXf Vguess,
    const Eigen::VectorXf& Vappwl1, const Eigen::VectorXf& Vappwl2,
    const Eigen::VectorXf& Vappbl1, const Eigen::VectorXf& Vappbl2,
    const float dt,
    bool linear = false
);
```

This method simulates a voltage being applied to the crossbar and calculates the current flowing through each memristor. It first determines the crossbar node voltages through ‘LinearSolve()‘ or ‘NonlinearSolve()‘ depending on the provided mode. Next, the ‘memristor.ApplyVoltage()‘ method of each individual memristor is called, where the voltage being applier is determined from the calculated node voltages. Finally, the result of each ‘memristor.ApplyVoltage()‘ call is the current flowing through that memristor, which is then collected into a matrix and returned at the end of the function.

This function takes in seven input parameters. The first five parameters are identical to the input parameters of both the ‘LinearSolve()‘ and the ‘NonlinearSolve()‘ methods. The ‘dt‘ parameter is the amount of time that the voltage is applied for. It is used to determine the state changes by the subsequent ‘memristor.ApplyVoltage()‘ method calls. Finally, the ‘linear‘ determines whether the crossbar solver is used in linear or non-linear mode.

Listing 3.20 shows the function declaration for this method.

Simulate()

Listing 3.21: Function declaration of the ‘Simulate()‘ method

```
void Simulate(
    const std::vector<bool> Vwl1, const std::vector<bool> Vwl2,
    const std::vector<bool> Vbl1, const std::vector<bool> Vbl2,
    const std::vector<std::vector<bool>> weights,
    const std::vector<std::array<float, 2>> waveform,
    const float dt,
    std::vector<std::vector<float>>& Iout,
    std::vector<float>& Iout_MAC,
    bool linear = false, bool drift = true
);
```

This method is intended as the primary interface of the crossbar simulator and encompass a complete simulation routine for conventional use case. First, this method sets the memristor states and access transistors using the ‘SetState()‘ and ‘SetAccessTransistors()‘ methods. The ‘SetState()‘ function ensures every memristor is in the state corresponding to the provided weights, while the ‘SetAccessTransistors()‘ ensures memristors are only connected if a voltage is applied to their wordline. This method of transistor activation avoids sneak path currents.

Next, the input waveform is discretized with the provided time step, and is used to repeatedly call ‘ApplyVoltage()‘. The input waveform is defined as a list of break points, where a break point consists of a voltage and a time stamp. The waveform is discretized by taking two breakpoints and linearly interpolating between to two votlages based on the time stamps and the time step size. Fig. 3.8 shows an example wave definition and the waveform it describes.

After every call to ‘ApplyVoltage()‘, the result is appended to a vector. The result is a vector where each entry is a matrix with elements corresponding to the current running through an individual memristor. When the entire wave is simulated, this vector is averaged across the time period of interest and is written to the ‘Iout‘ parameter. Thus this parameter contains one element for each memristor, where each element corresponds to the average current running through that memristor during the time period of interest.

Similarly, the average multiply-and-accumulate current on the bitlines is calculated with the averaged values, and is added to the ‘Iout_MAC‘ parameter. Consequently, the input parameters ‘Iout‘ and ‘Iout_MAC‘ are used as the output of the ‘Simulate()‘ function. As such, both parameters will be cleared before use.

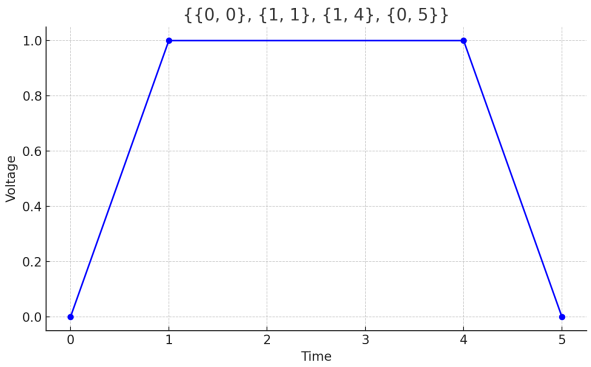


Figure 3.8: Example wave definition and graph of the described waveform

Finally, the ‘Simulate()‘ method includes a ‘linear‘ and ‘drift‘ input parameter. The ‘linear‘ parameter determines whether to use the ‘LinearSolve()‘ method or the ‘NonlinearSolve()‘ method during the ‘ApplyVoltage()‘ calls. The ‘drift‘ parameters indicates whether to simulate state drift of the memristors. This is achieved by setting the timestep to 0 for ‘ApplyVoltage()‘ calls when drift is not desired. By default linear mode is turned off and drift is turned on.

Listing 3.21 shows the function definition for this method.

3.4.2. Multithreading

In the ‘ApplyVoltage()‘ function and the non-linear solver, described in section 3.3, there are instances where a certain method is called for every memristor in the crossbars. Each of these method calls only pertains to a single memristor and are completely independent of the other method calls. This means that such a routine could greatly benefit by parallelism such as multithreading.

The crossbar simulator achieves this through a thread pool: a collection of several worker threads and a task list. During operation, tasks can be added to this list, which the worker threads subsequently remove to execute. By utilizing this method, unnecessarily creating and freeing threads during simulation is avoided and the size of the thread pool is limited to a predetermined amount. The amount of worker threads can be set in the ‘simulation_settings.h‘ file described in section 3.4.3.

3.4.3. Simulation Settings

Various aspects of the simulator require values to determine the behaviour or accuracy of the simulation. For example, the iterative method used in the non-linear solver, described in section 3.3, uses a certain error value for its exit criterion, and uses a certain maximum number of allowed iterations. All these settings involved in simulation are collected into a single header file from which they can be altered if so desired. The available settings are as follows:

- Settings specific to the JART VCM v1b var memristor model described in section 3.1:
  - ‘memristor_fixedpoint_criterion‘: exit criterion of the memristor fixed-point solver described in section 3.1.3. By default set to 10^{-6}
  - ‘memristor_fixedpoint_it_max‘: iteration limit of the memristor fixed described in section 3.1.3. By default is set to 10^{3}
  - ‘memristor_fixedpoint_warn_iteration_limit‘: flag that determines whether the memristor fixed-point solver, described in section 3.1.3, should warn that the iteration limit has been reached. By default set to false
  - ‘memristor_dynamic_time_step_N_limit‘: changes in memristor state above which the dynamic time stepping mechanism, described in section 3.1.2, should activate. By default set to 10^{-2}.
  - ‘memristor_dynamic_time_step_t_limit‘: time step sizes below which the dynamic time stepping mechanism, described in section 3.1.2, should not activate. By default set to 10^{-12}

  - 'memristor_dynamic_time_step_time_division': the amount of time step divisions the dynamic time stepping mechanism, described in section 3.1.2, should apply when activated. By default set to 2
  - 'memristor_get_resistance_voltage_threshold': the amount of voltage below which the current is assumed to be 0 for the purposes of calculating resistance using the 'GetResistance()' method, as described in section 3.1.2. By default set to 10^{-6}
- Settings related to the non-linear crossbar solver described in section 3.3
  - 'non_linear_fixed_point_a': dampening factor of the fixed-point solver. By default set to 1
  - 'non_linear_fixed_point_it_max': iteration limit of the fixed-point solver. By default set to 100
  - 'non_linear_fixed_point_criterion': exit criterion of the fixed-point solver. By default set to 10^{-6}
  - 'non_linear_warn_iteration_limit': flag that determines whether the fixed-point solver should warn that the iteration limit has been reached. By default set to true
- Settings related to wave definition. These settings are not used in the simulator and are exclusively intended for altering test setups. Example test setups are further described in section 3.4.4
  - 'voltage_pulse_width': total width of a voltage pulse including rise and fall times. By default set to $50 \cdot 10^{-6}$
  - 'voltage_pulse_height': maximum amplitude of a voltage pulse. By default set to 0.1
  - 'voltage_pulse_rise_time': amount of time required to reach the maximum voltage of a voltage pulse from $0V$. By default set to $5 \cdot 10^{-6}$
  - 'voltage_pulse_fall_time': amount of time required to return to $0V$ from the voltage peak of a voltage pulse. By default set to $5 \cdot 10^{-6}$
  - 'simulation_time_step': width of the time step used to simulate the voltage pulse. Note that memristor dynamically reduce the time step size temporarily, as described in section 3.1.2. By default set to 10^{-6}
- Miscellaneous:
  - 'linear_operating_point' determines the linear operating point used for the 'LinearSolve()' method described in section 3.4.1. A value of 0 indicates an operating point of 0V. A value of 1 indicates an operating point equal to the supply voltage. And a value of 2 indicates a custom operating point provided through the 'Vguess' input parameter of the method. By default, this setting is set to 1.
  - 'simulation_num_threads': the amount of threads used for the threadpool described in section 3.4.2. A value of 0 indicates an empty thread pool and ensures any thread usage during simulation is bypassed. By default set to 4

3.4.4. Usage Example

Included in the GitHub repository, appendix D, are two files intended for experimentation and validation and provide a usage example of this crossbar simulator.

'RRAM_validation.cpp' is intended to compare the results of this simulator with other simulators. It takes in the file path to a folder containing input and weight data, runs the simulation, and writes the individual memristor current as well as the multiply-and-accumulate current to the same folder. If provided, individual and MAC currents can also be loaded and compared to the results of this simulator.

'RRAM_test_setup.cpp' is intended as a general purpose method to generate simulation data based on various input parameters. It generates random input and weight matrices based on provided input and weight ratios, simulates the crossbar with the provided wave definition, and writes the weights, input, and results to files. Additional settings allow for operating in linear mode, removing state drift, and utilizing different memristor models.

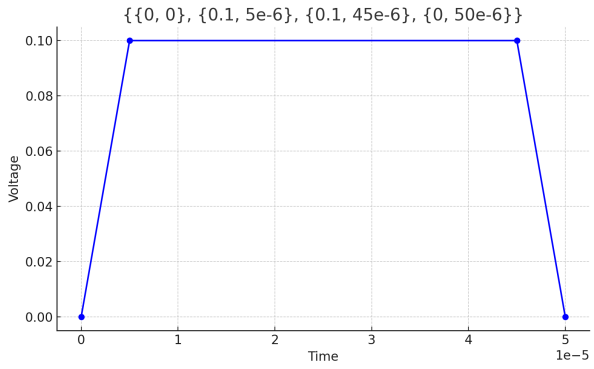


Figure 3.9: Wave form as defined using listing 3.24

The usage of the crossbar simulator in ‘RRAM_validation.cpp‘ and ‘RRAM_test_setup.cpp‘ is largely the same. First, the crossbar simulator is constructed and initialized with the chosen memristor model, as shown in listing 3.22.

Listing 3.22: Crossbar simulator creation and initialization exampe

```
1    CrossbarSimulator crossbar = CrossbarSimulator(M, N);
2    crossbar.Initialize<JART_VCM_v1b_var>();
```

Next, the appropriate parameters for line resistance are set. ‘RRAM_validation.cpp‘ and ‘RRAM_test_setup.cpp‘ use default parameters thus do not require this step. However, line resistances can be set using the ‘SetCrossbarParameters()‘ method as shown in listing 3.23. In this example, every line resistance is set to 1Ω except for the top and right source resistances. Those are instead set to infinity indicating those sources are disconnected.

Listing 3.23: Crossbar simulator ‘SetCrossbarParameters()‘ example

```
1    crossbar.SetCrossbarParameters(1, INFINITY, INFINITY, 1, 1, 1);
```

Next, a wave is defined. Waves are described by a list breakpoints, each of which consist of a voltage and a time stamp. During simulation, the crossbar simulator will linearly interpolate between those breakpoints to discretize the wave. Both ‘RRAM_validation.cpp‘ and ‘RRAM_test_setup.cpp‘ use the default settings provided in ‘simulation_settinghs.h‘ to define their wave. Using these settings, the wave shown in Fig. 3.9 is defined as shown in listing 3.24.

Listing 3.24: Wave definition example

```
1    float dt = simulation_time_step;
2    std::vector<std::array<float, 2>> Vwave = {
3        {{0, 0}},
4        {{voltage_pulse_height, voltage_pulse_rise_time}},
5        {{voltage_pulse_height, voltage_pulse_width - voltage_pulse_fall_time}},
6        {{0, voltage_pulse_width}}
7    };
```

Finally, the crossbar can be simulated by providing the input, weight, and wave data to the ‘Simulate()‘ method. Additionally, Empty vectors for ‘Iout_avg‘ and ‘Iout_MAC‘ have to be provided for the result of the simulation. Listing 3.25 shows an example of this process. In this example, three empty vectors are provided for the top, right, and bottom sources to indicate $0V$. Additionally, the top and right sources will be disconnected due to their line resistances being set to infinite in listing 3.23. A diagram of the full simulation process is shown in figure 3.10

Listing 3.25: Crossbar ‘Simulate()‘ example

```
1    std::vector<std::vector<float>> Iout_avg;
2    std::vector<float> Iout_MAC;
3    crossbar.Simulate(input, {}, {}, {}, weights, Vwave, dt, Iout_avg, Iout_MAC, linear,
     drift);
```

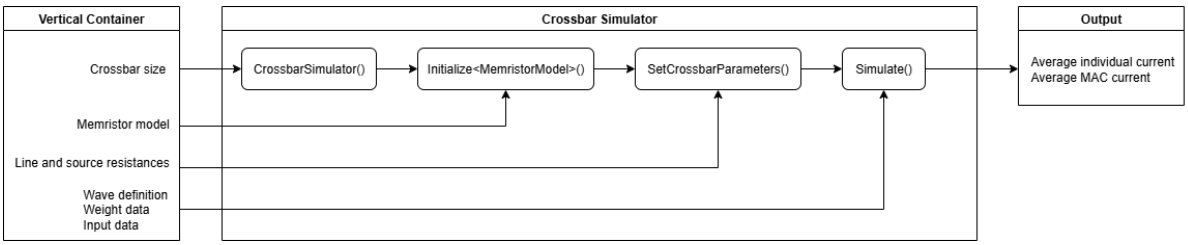


Figure 3.10: Flow diagram of an example usage of the crossbar simulator as described in section 3.4.4

4

Results & Discussion

This chapter presents the results of validation and performance metric experiments. Each sub-module, as described in chapter 3, is first tested separately, followed by an experiment of the complete system.

Due to limited access to the TU Delft Q&CE Cluster, not all experiments have been performed in a comparable computing environment. When mentioned, experiments have been performed on a personal computer instead. Due to this computer maintaining other tasks and not having access to all necessary tools, performance results from these experiments are not reproducible. However, experiments on this device are generally run in quick succession, thus can still be compared to each other.

4.1. Memristor Model

This section describes the experimental setup and results of the model described in section 3.1.

4.1.1. Experimental Setup

In order to validate the implemented memristor model, the output of the C++ simulation is compared to the output of a Cadence Spectre simulation. This simulation entails applying a voltage signal to the terminals of a singular memristor model, and measuring the output current and other relevant metrics. The applied voltage signal is a triangle wave which travels from $0V$ at $t = 0s$, to $-1.5V$ at $t = 1.5s$, to $1.5V$ at $t = 4.5s$, and returns to $0V$ at $t = 6s$ for a total of 6 seconds. Both the C++ and Cadence Spectre simulation use default model parameters with a time step of $1ms$. However, it should be noted that both the C++ and the Cadence Spectre simulation include mechanisms to force a smaller time step when deemed necessary. The Verilog-A code used for the Cadence Spectre simulation is available on the EMRL website [3]. Additionally, their exact simulation setup is explained in section 3.1.2 of their provided user guide [38].

4.1.2. Validation

Fig. 4.1 shows the simulation results and comparison of the C++ and Cadence Spectre simulation. The top left image plots the I-V relationship of the memristor model on a logarithmic scale. The top right image plots the oxygen vacancy concentration in the disc region throughout the simulation. And, he bottom left image plots the temperature in the device throughout the simulation. These three images include both the results of the Cadence Spectre simulation and the C++ simulation. However, the C++ graph might not be visible at certain points due to it being overlapped by the Cadence Spectre graph.

The bottom right image plots the difference in output current between the Cadence Spectre simulation and the C++ simulation. This error is calculated by taking the absolute value of the difference between the two output currents at any given point. However, the measurement time stamps of the two simulations generally do not align, thus results have been linearly interpolated to more accurately calculate the difference. Differences below $1e - 12$ have not been shown on this graph.

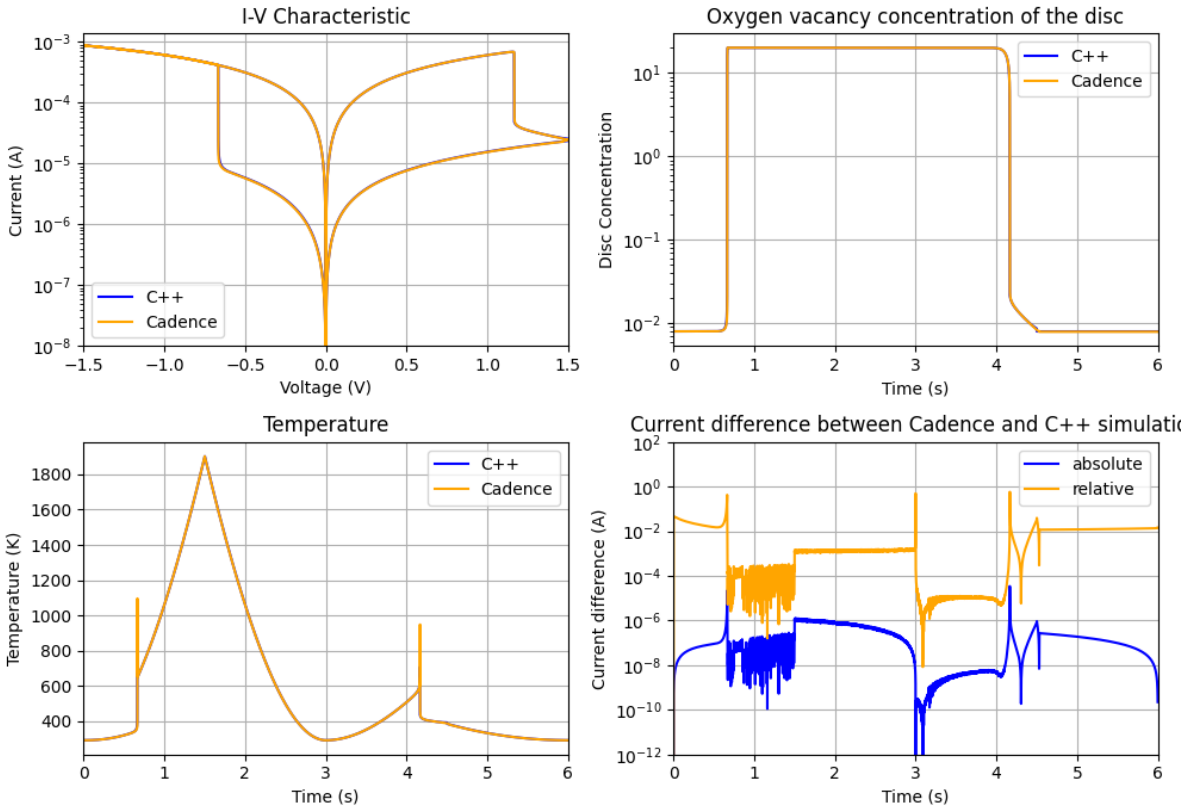


Figure 4.1: Simulation results and comparison between C++ and Cadence Spectre

4.1.3. Performance Metrics

The execution time for both simulators is measured using the same experimental setup as for validation: a 6-second triangle waveform as input with a maximum time step of $1ms$. The resulting metrics are the average measurement of 10 single threaded simulations.

The C++ simulation averaged $97.18ms$, where the execution time is measured as the elapsed time of the simulation excluding I/O operations. The Cadence Spectre simulation averaged $867.4ms$, where the execution time is measured as the total CPU time required for transient analysis as reported by the Cadence Spectre output log.

4.1.4. Discussion

Compared to Cadence Spectre, the C++ simulation of a single memristor model shows a relative error of around 1% or bellow. Some interpolation has been performed to calculate the error, possibly introducing further inaccuracies. However, any additional error introduced by this interpolation is not significant enough to influence these results. In the context of CIM simulation, the absolute error of the model is sufficiently low for reliably distinguishing between LRS and HRS. Furthermore, this model proves sufficiently accurate to simulate the switching mechanism.

Three short instances can be observed in Fig. 4.1 where the relative error significantly rises above 1%. These error spikes occur around $0.66s$, $3s$, and $4.16s$, and coincide with either a change in internal state or an applied voltage near $0V$. During switching, the internal state, and consequently the output current, changes rapidly. Small deviations in the exact switching moment can therefore lead to a larger relative error. Directly before and after these switching events, the error remains low, signifying accurate simulation. Closely around an applied voltage of $0V$, the output current is minimal. Any small error in calculation is therefore comparatively larger around this point, resulting in a larger relative error. Fig. 4.1 shows that the absolute error around $3s$ remains low.

The relative error plot in Fig. 4.1 can be split into distinct sections defined by switching events and

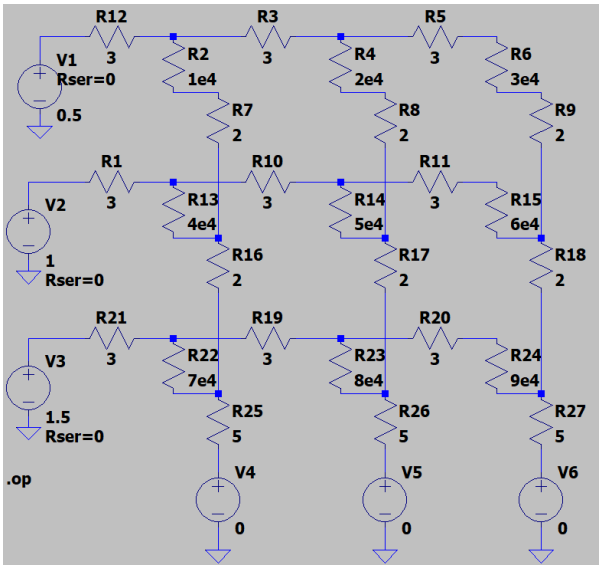


Figure 4.2: LTspice simulation circuit

voltage trends. Notably, the relative error seems to be higher in the sections at the beginning and end of the simulation, during which the memristor is in the HRS state. Comparatively, the output current during HRS is significantly lower than during RLS, possibly contributing to the difference in relative error. In CIM simulations, overall output current of crossbar columns is largely dependent on LRS current. As a results, the higher relative HRS error has a lesser impact on crossbar evaluation. Nevertheless, the exact source of this error deviation is still unknown and should be investigated further.

In terms of execution time, the C++ simulation is significantly faster than Cadence Spectre. Experimental results indicate a speedup of approximately $9\times$. However, both simulators use different mechanisms for determining time steps, precision, and other simulation aspects. Therefore, a direct comparison might not fully be justified as the C++ simulator will inevitably have made approximations compared to Cadence Spectre.

4.2. Linear Crossbar Solver

This section describes the experimental setup and results of the model described in section 3.2.

4.2.1. Experimental Setup

The linear crossbar solver is tested with two separate experiments. The first experiment is aimed at validating the model, while the second experiment is aimed at measuring performance metrics.

In the validation experiment, the results of the C++ simulation is compared with the results from an LTspice operation point simulation. Both simulations are performed on a 3×3 crossbar array with the following parameters:

- A wordline resistance of 3Ω
- A bitline resistance of 2Ω
- A wordline access resistance of 3Ω
- A bitline access resistance of 5Ω
- A wordline voltage of $0.5V$, $1V$, and $1.5V$
- A bitline voltage of $0V$
- A unique resistance for each device ranging from $1e4\Omega$ to $9e4\Omega$

The LTspice equivalent of this circuit is shown in Fig. 4.2.

This experiment is repeated with an 8×8 crossbar with similar parameters.

Table 4.1: Simulation results for a 3×3 crossbar

	LTspice	C++
I(V4)	9.62983e-5	9.62983e-5
I(V5)	6.36856e-8	6.36856e-5
I(V6)	4.99559e-5	4.9956e-5

Table 4.2: Simulation results for a 8×8 crossbar

	LTspice	C++
I(V9)	8.24893e-5	8.24893e-5
I(V10)	5.56515e-5	5.56515e-5
I(V11)	4.56712e-5	4.56711e-5
I(V12)	4.00561e-5	4.0056e-5
I(V13)	3.62733e-5	3.62733e-5
I(V14)	3.34609e-5	3.34609e-5
I(V15)	3.1239e-5	3.1239e5
I(V16)	2.94112e-5	2.94111e-5

Crossbar Size	Cadence Spectre	C++
8×8	$5.921ms$	$1.282ms$
16×16	$18.28ms$	$9.939ms$
32×32	$126.1ms$	$84.43ms$
64×64	$2,006ms$	$784.3ms$

Table 4.3: Linear crossbar simulation execution time for various crossbar sizes

The second experiment aims to measure the execution time for the C++ simulation and a Cadence Spectre DC operating point analysis. Both simulations are executed as single threaded applications and are repeated 100 times for each platform with randomized resistances and input voltages. Furthermore, this process is repeated for various crossbar sizes.

Each simulation has the following parameters:

- A wordline resistance of 3Ω
- A bitline resistance of 2Ω
- A wordline access resistance of 3Ω
- A bitline access resistance of 5Ω
- A wordline voltage randomized to either $1.5V$ or $0V$ applied to the left edge. The right edge remains disconnected
- A bitline voltage of $0V$ applied to the bottom edge. The top edge remains disconnected
- Device resistances linearly randomized between $1e3\Omega$ and $1e5\Omega$

4.2.2. Validation

Table 4.1 shows the results of both the LTspice operating point and the C++ simulation. The results are measured in terms of current flowing out of the bitlines of the crossbar. $I(V4)$, $I(V5)$, and $I(V6)$ refer to the currents flowing through the voltage sources $V4$, $V5$, and $V6$, which refer to the voltage sources of the same name as in Fig. 4.2.

Table 4.2 shows the results of a similar experiment for an 8×8 crossbar. The results are similarly measured in terms of current flowing out of the bitlines of the crossbar.

4.2.3. Performance metrics

Table 4.3 shows the execution time of C++ and Cadence Spectre simulations for various crossbar sizes. These results are an averaged measurement of 100 single threaded simulations with identical setups for both the Cadence Spectre and C++ simulations, as described in section 4.2.1. For the C++ simulation, the execution time is taken as the measured elapsed time of the simulation without the partial precalculation of the G_{ABCD} matrix and E vector, as described in section 3.2.2. For Cadence Spectre simulation, the execution time is taken as the DC simulation CPU time as reported by the Cadence Spectre output log.

4.2.4. Discussion

Compared to LTSpice, the C++ simulation of the linear crossbar shows nearly identical results. From table 4.1 and table 4.2 it can be observed that the two simulators exclusively deviate in the least signif-

Crossbar Size	C++	Cadence Spectre
8×8	$4ms$	$140.4ms$
16×16	$270ms$	$1,006ms$
32×32	$1,746ms$	$79,380ms$

Table 4.4: Non-linear crossbar simulation execution time for various crossbar sizes

icant digit, which could be attributed to rounding errors. Therefore, the C++ simulation can be said to be comparable to LTSpice in terms of accuracy.

In terms of execution time, the C++ simulation is measurably faster than Cadence Spectre, as can be seen in table 4.3. Experimental results indicate speedup of at least $1.5\times$ across the tested configurations. There does not seem to be a clear relation between speedup and crossbar size. In terms of scaling, the C++ simulation seems to execute around 8 or 9 times slower for a crossbar with twice the amount of rows and columns.

4.3. Non-linear Crossbar Solver

This section describes the experimental setup and results of the model described in section 3.3.

4.3.1. Experimental Setup

The aim of this experiment is to measure the execution time of solving the non-linear crossbar for a single time step. Thus, this experiment does not include updating the internal state of the memristor models. The experimental setup for testing the non-linear crossbar solver is similar to the experimental for the linear crossbar solver, discussed in section 4.2.1. Both the execution time of the C++ and Cadence Spectre simulation are measured for a single threaded application, and averaged over 100 iterations. The internal state of each memristor and the input voltages are randomized, and this process is repeated for various crossbar sizes.

Each simulation has the following parameters:

- A wordline resistance of 3Ω
- A bitline resistance of 2Ω
- A wordline access resistance of 3Ω
- A bitline access resistance of 5Ω
- A wordline voltage randomized to either $1.5V$ or $0V$ applied to the left edge. The right edge remains disconnected
- A bitline voltage of $0V$ applied to the bottom edge. The top edge remains disconnected
- Memristor internal state randomized to either $N_{disc,min}$ or $N_{disc,max}$

4.3.2. Performance metrics

Table 4.4 shows the execution time of C++ and Cadence Spectre simulations for various crossbar sizes. These results are an averaged measurement of 100 single threaded simulations with identical setup for both the Cadence Spectre and C++ simulations, as described in section 4.3.1. For the C++ simulation, the execution time is taken as the measured elapsed time of the simulation without the partial precalculation of the G_{ABCD} matrix, as described in section 3.2.2. For Cadence Spectre, the execution time is taken as the DC simulation CPU time as reported by the Cadence Spectre output log.

4.3.3. Discussion

The non-linear crossbar solver is the combination of the linear crossbar solver and numerous instances of the memristor model. Therefore, it inherits the performance results of those previous modules, which is reflected in table 4.4. However, there does not seem to be a clear trend between speedup across different crossbar sizes. Similarly, there does not seem to be a clear trend between the increase in execution time and the increase in crossbar size. Analysis of a broader set of crossbar configurations could still reveal such a trend.

It should be noted that validation results of the non-linear crossbar solver are not included. Due to time

constraints and limited experience with Cadence Spectre, adequate validation could not be performed. However, the accuracy of this module can be inferred from the validation results of the other experiments. Specifically, the combined crossbar simulator relies extensively on the non-linear crossbar solver, and its results strongly suggest that the solver is sufficiently accurate.

4.4. Combined Crossbar Simulator

This section describes the experimental setup and results of the combined crossbar simulator described in section 3.4.

4.4.1. Experimental Setup

In order to validate and measure the performance of the complete memristive crossbar simulator, the first layer of an MNIST classification neural network is simulated. For this experiment, the classification of this network is not considered. Instead, the simulators will be compared based on the currents generated by the crossbar.

The complete experiment consists of 10 inputs being given to 784 crossbars with 32×32 memristors each, making for a total of 7840 simulations. For each simulation, the memristors are preloaded with binary weights. Next, an input voltage pulse is applied to the crossbar based on the neural network's input. For an input value of true, the applied voltage pulse has a rise time of $5\mu s$, a fall time of $5\mu s$, a width of $50\mu s$, and a peak voltage of $100mV$. Additionally, memristors will only have their access transistor turned on if the associated input row receives a voltage pulse. The remaining relevant simulation parameters are as follows:

- Input voltages are applied to the left edge of the crossbar. The right edge remains disconnected
- Input voltages are simulated with a time-step of $1\mu s$
- The bottom edge of the crossbar is set to $0V$. The top edge remains disconnected
- All wordline access resistances, bitline access resistances, wordline resistances, and bitline resistances are 1Ω
- Memristors are initialized with default parameters before being loaded with the appropriate weights

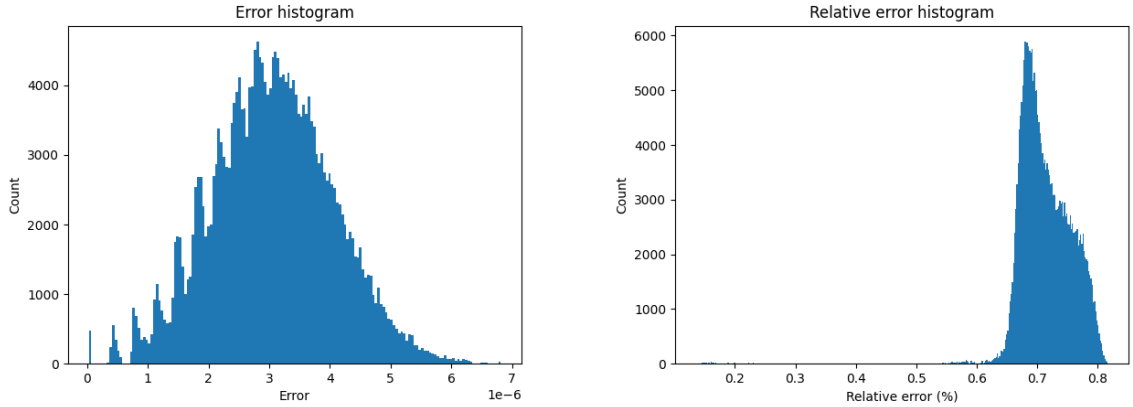
The result of each simulation is a vector containing the accumulated currents for each column of the crossbar, and a matrix containing the current through each individual memristor.
The complete experiment is repeated using Cadence Spectre with similar parameters.

4.4.2. Validation

Fig. 4.3a shows a histogram of the error of the accumulated crossbar currents. For each column, the error is calculated by taking the absolute difference between the C++ and the SPICE simulation currents. Similarly, Fig. 4.3b shows a histogram of the relative error, where each error is calculated as the absolute error divided by the current of the SPICE simulations. These figures show an error in accumulated current between $0.3\mu A$ and $6.5\mu A$ with an average around $3\mu A$. Compared to the total accumulated current, these values deviate 0.6% to 0.8%, with the majority deviating around 0.68%.

Fig. 4.4a shows a heat map of the average error for each individual memristor corresponding to their placement in the crossbar. For each entry, the error is calculated by taking absolute difference between the C++ and the SPICE simulation current. Similarly, Fig. 4.4b shows a heat map of the average relative error, where each entry is calculated as the absolute error divided by the current of the SPICE simulation. These figures show an error in individual current between $0.16\mu A$ and $0.26\mu A$. Compared to the total individual current, these values deviate around 0.4% to 0.6%.

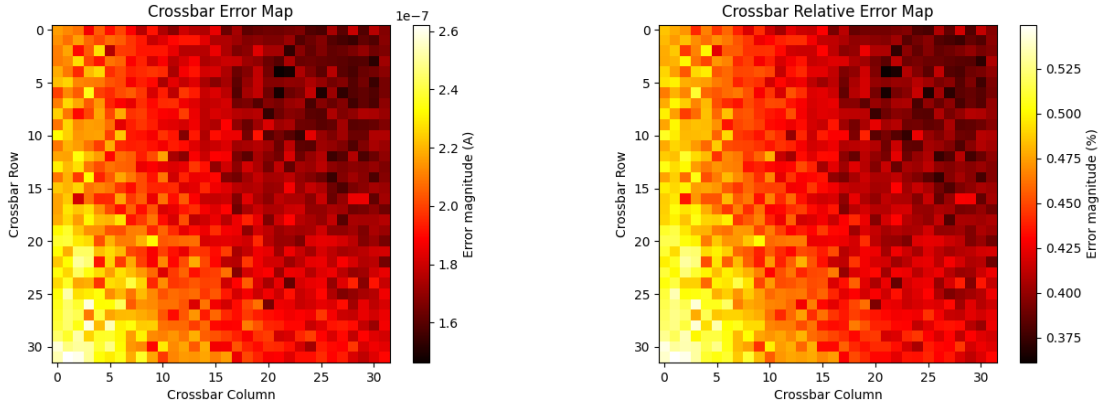
The individual error calculation includes contributions from both memristor cells in LRS as well as HRS. However, both states operate at different resistance values and thus differ in terms of current amplitude. Fig. 4.5a and 4.5b show the absolute individual error and the relative individual error when considering only LRS contributions. Fig. 4.6a and 4.6b show similar graphs for HRS contributions. For LRS, the absolute error is approximately from $0.3\mu A$ to $0.5\mu A$ with a relative error between 0.65% and 0.85%. And for HRS the absolute error is approximately from $1.5nA$ to $3.5nA$ with a relative error between 0.1% and 0.24%.



(a) Absolute error compared to SPICE

(b) Percentage error compared to SPICE

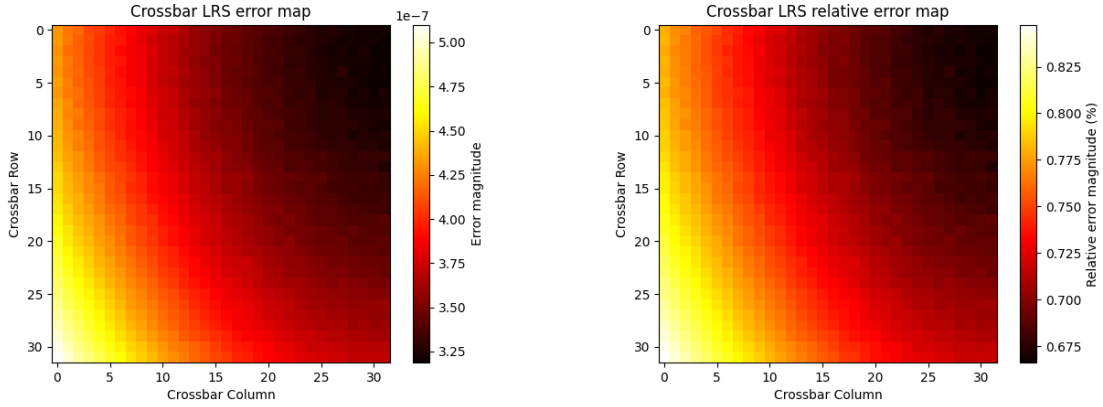
Figure 4.3: Accumulated current error histograms for a 32×32 crossbar



(a) Absolute error compared to SPICE

(b) Percentage error compared to SPICE

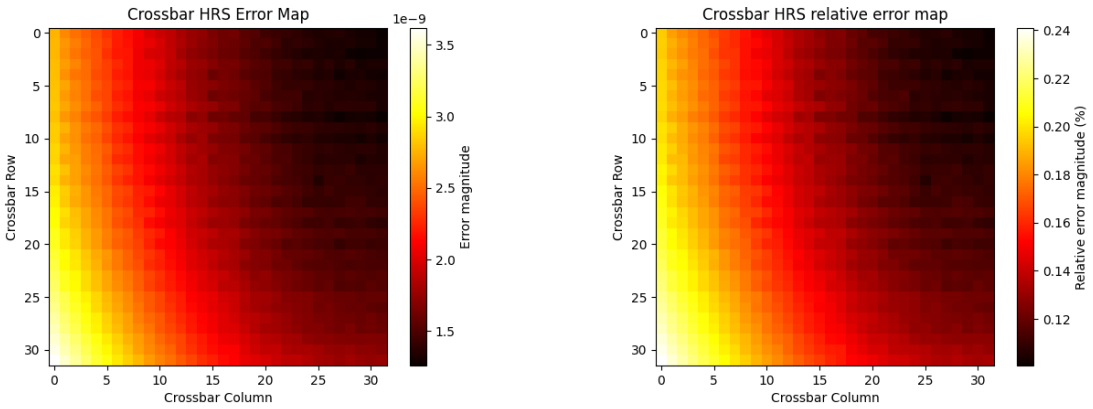
Figure 4.4: Error heat map of individual memristors currents for a 32×32 crossbar



(a) Absolute error compared to SPICE

(b) Percentage error compared to SPICE

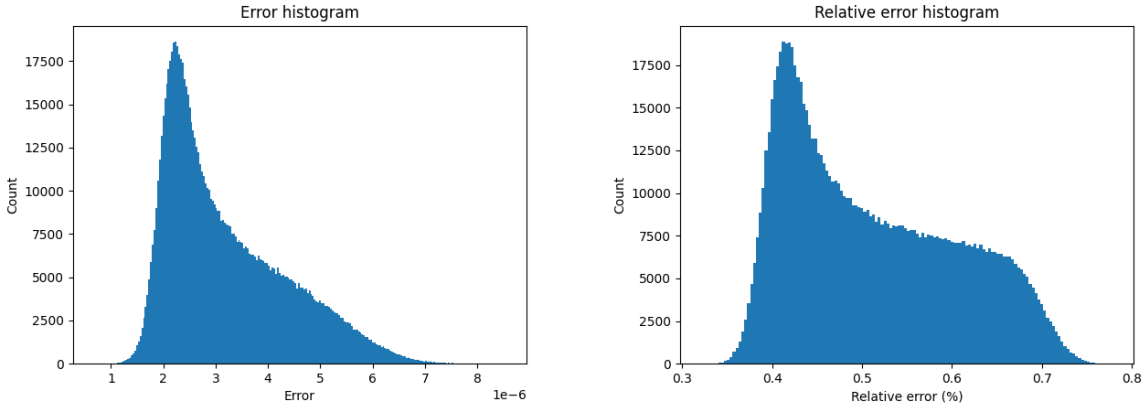
Figure 4.5: Error heat map of individual memristors currents in the LRS state for a 32×32 crossbar



(a) Absolute error compared to SPICE

(b) Percentage error compared to SPICE

Figure 4.6: Error heat map of individual memristors currents in the HRS state for a 32×32 crossbar



(a) Absolute error compared to SPICE

(b) Percentage error compared to SPICE

Figure 4.7: Accumulated current error histograms for a 64×64 crossbar

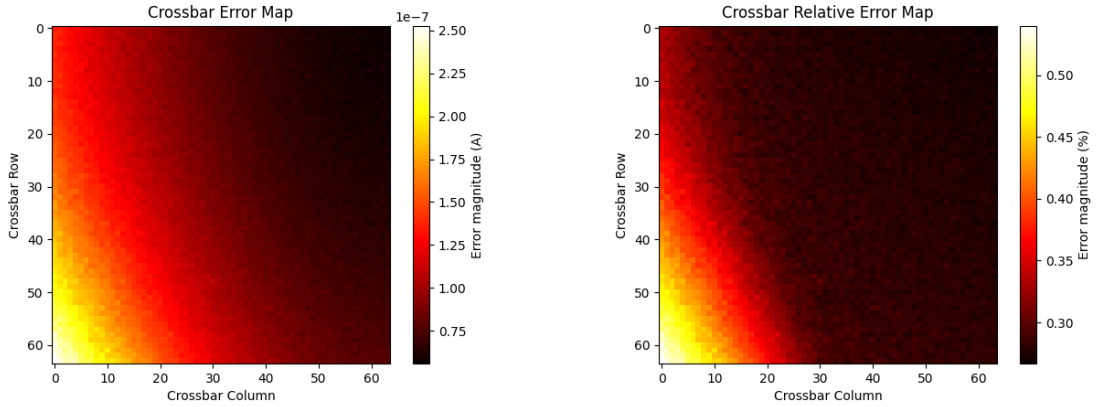
64x64

Fig. 4.7a–4.10b show validation results for an experiment similar to the experiment described in section 4.4.1 where the crossbar size is increased to 64×64. Fig. 4.7a and 4.7b show the accumulated bitline current error to approximately be between $1\mu A$ and $7.5\mu A$ with a relative error between 0.35% and 0.75%. Fig. 4.8a and 4.8b show the individual memristor error to approximately be between $0.05\mu A$ and $2.5\mu A$ with a relative error between 0.25% and 0.55%. Fig. 4.9a and 4.9b show the individual error from memristor in the LRS state to approximately be between $1\mu A$ and $5\mu A$ witha relative error between 0.4% and 0.8%. And Fig. 4.10a and 4.10b show the individual error from memristor in the HRS state to approximately be between $0.5nA$ and $3.5nA$ with a relative error between 0.025% and 0.25%.

4.4.3. Performance metrics

The execution time for both simulators is measured using the same experimental setup as validation process. The C++ simulation averaged $527.9ms$ per simulation, or 1 hour, 9 minutes, and 59 seconds for the complete experiment of the 32×32 crossbar, where execution time is measured as the elapsed time of the complete simulation process, including initialization and I/O operations. The Cadence Spectre measurements of the 32×32 experiments have been obtained from a previous experiments, the performance metrics of which have been lost. However, the entire experiment was estimated to take about a week.

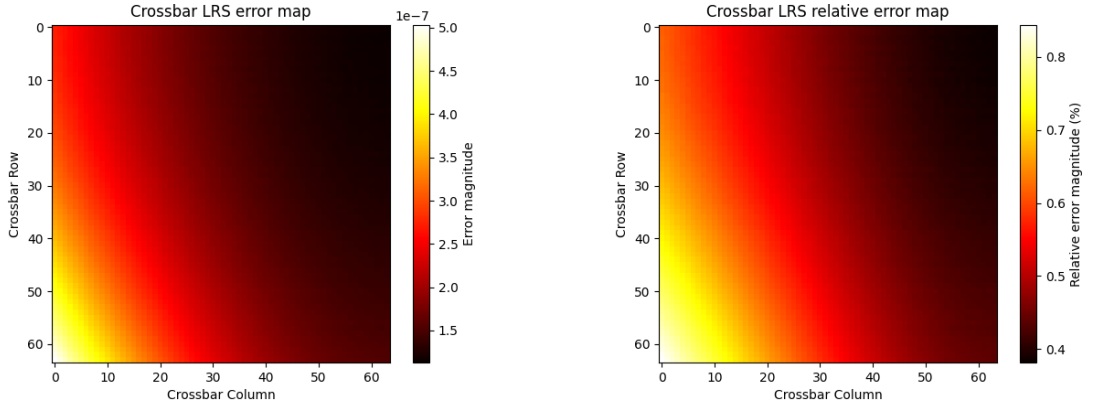
Similarly, the C++ and Cadence experiment for the 64×64 experiment both could not be performed on the QCE cluster. Instead the 64×64 experiment has been performed on a personal computer, while



(a) Absolute error compared to SPICE

(b) Percentage error compared to SPICE

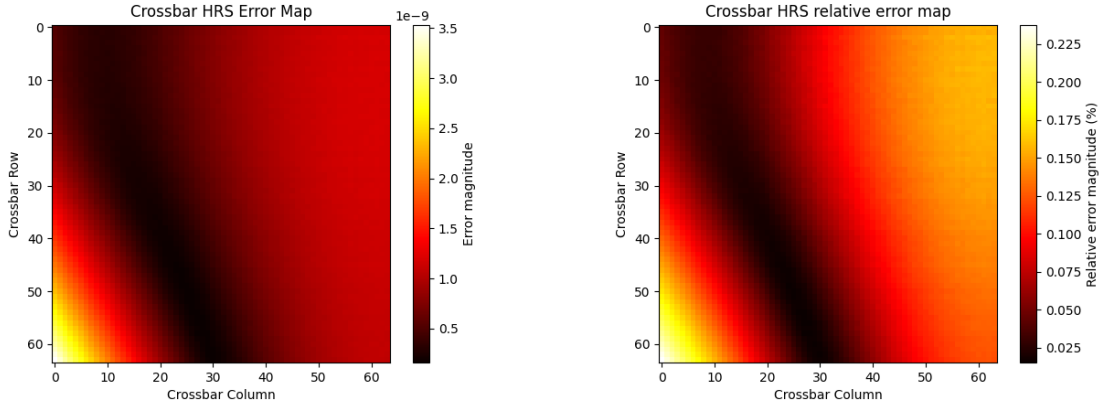
Figure 4.8: Error heat map of individual memristors currents for a 64×64 crossbar



(a) Absolute error compared to SPICE

(b) Percentage error compared to SPICE

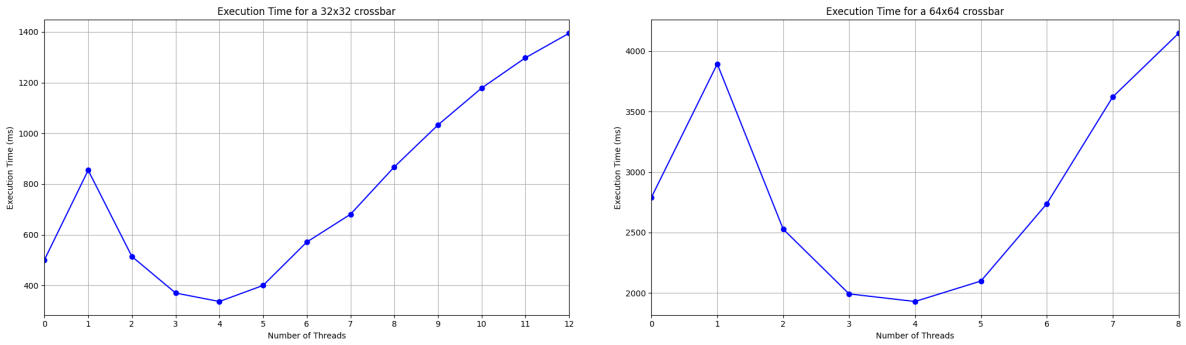
Figure 4.9: Error heat map of individual memristors currents in the LRS state for a 64×64 crossbar



(a) Absolute error compared to SPICE

(b) Percentage error compared to SPICE

Figure 4.10: Error heat map of individual memristors currents in the HRS state for a 64×64 crossbar



(a) 32×32 crossbar

(b) 64×64 crossbar

Figure 4.11: Execution times of simulation using various thread pool sizes

the Cadence measurements have been obtained from a previous experiment. The C++ simulation averaged $2753ms$ per simulation across 15000 simulations, where the execution time is measures as the elapsed time of the complete process, including initialization and I/O operations. The performance metrics for 64×64 Cadence experiment have similarly been lost. However, the transient analysis of each simulation was estimated to take about 9 minutes.

4.4.4. Multithreading

Thus far, all experiments in this chapter have been performed using a single threaded simulator. However, section 3.4.2 describes the implementation of multi-threaded functionality. Therefore, this section presents a comparison in terms of execution time of simulations using different thread pool sizes. Unfortunately, this experiment could not be performed on the TU Delft Q&CE cluster due to it not being available at the time of writing. Instead, this experiment is performed on a personal computer. Results of this experiment are not comparable performance metrics of other experiments in this chapter, and trends in the results might not be accurate for different systems.

The experiment performed is similar to the validation experiment of section 4.4.1: a crossbar with 1Ω line resistances, a voltage pulse with an $100mV$ amplitude and width of $50\mu s$, input supplied to the left edge, ground set to the bottom edge, and the other two edges disconnected. Inputs and weights are set randomly with a ratio of 0.5, and each measurement represents the average of 100 simulations.

Fig. 4.11a shows a graph of the results for a 32×32 crossbar. Fig. 4.11b shows the results for a similar experiment using a 64×64 crossbar.

4.4.5. Discussion

The combined crossbar simulator is the final product of this thesis, and its performance ultimately decides the success of this work. Therefore, the experiment described in section 4.4.1 is a practical example of what this simulator could be used for. The goal of this experiment is to ascertain whether results are accurate enough to simulate a vector-matrix product without read errors.

Fig. 4.3a shows that, based on experimental results, the error in the multiply-and-accumulate current is at most $7\mu A$. Compared to the total multiply-and-accumulate current, this current deviates less than 1% from Cadence Spectre simulations. Furthermore, this error is well bellow the difference in LRS and HRS current, thus ensuring vector-matrix multiplication results are accurately read.

In terms of currents flowing through individual memristors, errors are in line with the multiply-and-accumulate results, the absolute errors accumulate to approximately the same multiply-and-accumulate error, and relative error seems comparable although slightly lower. This deviation can be explained by observing the individual error contributions of memristors in the LRS and HRS state. Due to differences in resistance, memristors in different states operate with different magnitudes of currents. And as can be seen by 4.5a and 4.6a, the error contributions from memristors in the LRS state is indeed higher that the error contributions from memristors in the HRS state. Furthermore, this relative error is in line with the relative error from the MAC current shown in Fig. 4.3b. Currents from memristors in the LRS state

are significantly higher than currents from memristors in the HRS state. Subsequently, LRS currents make up the majority of MAC current, and similarly contribute to most of the error.

Interestingly, memristors in the bottom left of the crossbar experience the most error, while being closest to the voltage sources. In contrast, memristors in the top right experience the least error, while they would experience the highest voltage drop due to line resistances. One possible explanation is due to a rise in error around $100mV$ in the memristor model. In the bottom right graph of Fig. 4.1 it can be seen that just after the $3s$ mark the relative error of the memristor model rises from around 10^{-6} to around 10^{-5}. This area of the graph coincides with the region where the applied voltage is around $100mV$. Therefore, the pattern in the heatmap could arise due to the error of the memristor model being lower for lower voltages, thus the crossbar having lower error in areas where the effects of line resistances are higher.

The validation results for the 64×64 crossbar experiment show similar results as to the 32×32 crossbar experiment. In general, error measurements for 64×64 are comparable at the bottom left of the crossbar while slightly better at the top right. With a larger crossbar, the top right memristors experience larger voltage drops due to line resistance, which reduces the simulation error compared to Cadence as discussed above.

Fig. 4.10a and 4.10b show odd behaviour not seen in the other heat maps: the error distribution seems to have a distinct diagonal band where it is lower than the surrounding area. Similar to the error patters of the other heat maps, this is likely due to the individual memristor error not being constant across different voltages. As the voltage across an individual memristor decreases, the error seems to decrease up to a certain point, after which it starts to increase again. However, an analysis of individual memristor error in Fig. 4.1 can not fully support this conclusion. Instead, error in individual memristors influences the voltage drop for later memristors, meaning this error could compound throughout a crossbar row. This compounding error then likely surpasses the reduction in error from lower voltages for memristors placed further from the sources.

In terms of execution time, the exact amount of speedup can not be determined. Nevertheless, with approximated Cadence Spectre measurements it appears that the C++ simulation is around two orders of magnitude faster for both 32×32 crossbars and 64×64 crossbars.

Multithreading

As mentioned in section 4.4.4, the results for the multithreading experiments are not directly comparable to the results of other experiments. However, there are some general trends observable in the graphs that are expected to remain true on different devices.

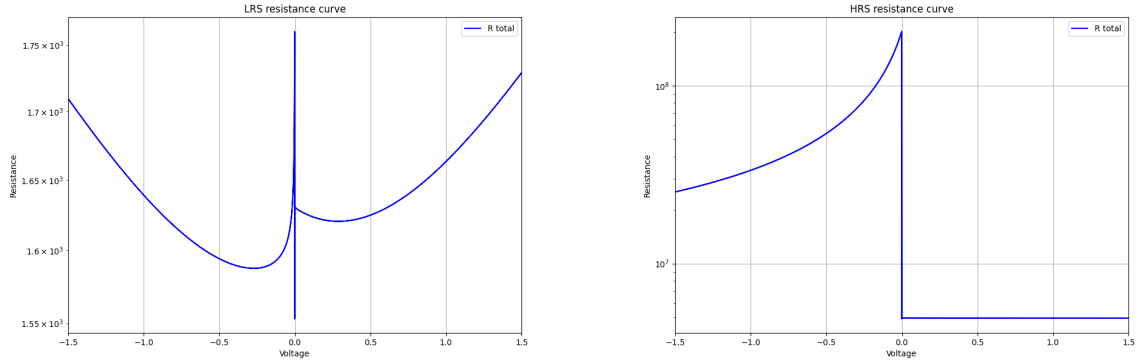
Both Fig. 4.11a and 4.11b show similar trends across the different threadpool sizes. First of all, a threadpool consisting of a single thread is significantly slower. In this configuration, the main thread would offload all the multithreading work to the single worker thread, and wait until that thread is finished. In terms of distributed work, this configuration is nearly identical to a configuration using no worker threads. Only one thread is busy at the time, thus the only difference relates to the overhead of managing the threadpool. Consequently, using one worker thread is slower.

Using two worker threads seems to be comparable to using no worker threads. This indicates that the threadpool related overhead is comparable to the work done by individual threads. This could be improved by decreasing the amount of necessary overhead, or by increasing the workload per thread.

Using three to five threads seems to be optimal. This range shows around a 30% decrease in terms of execution time compared using no threads. After 5 threads, the execution time seems to worsen significantly with each additional thread. At this point the threads are likely fighting for shared resources, namely the task queue. This further supports the possible improvement of increasing thread workload. Alternatively, a lock-free task queue could alleviate some of these issues.

4.5. Linear vs Non-linear Simulation

Up until this point, memristors were assumed to be non-linear devices. In general, this is true. However, other the simulation frameworks described in section 2.2.3 commonly approximate memristive devices



(a) LRS state

(b) HRS

Figure 4.12: Resistance curves of a JART VCM v1b var memristor without state changes

as linear. This section aims to explore the effects of non-linearity in crossbar simulation and compare this to a purely linear analysis.

4.5.1. Experimental Setup

For this analysis, the crossbar simulator developed in this thesis will be taken as the baseline truth for non-linear analysis. Linear analysis is performed using the same crossbar simulator configured such that any non-linear effects are removed. The two simulation methods can then be compared in order to determine how much linear analysis deviates from non-linear analysis.

As shown in Fig. 4.12a and 4.12b, the JART VCM v1b var memristor is a non-linear device. However, linear analysis requires it to be linear or to be approximated as such. In order to make this approximation, some operating point has to be chosen; a voltage around which the memristor is assumed to be linear. As discussed in section 3.4.3, this crossbar simulator supports three linear operating points: $0V$, supply voltage, or an user supplied operating point. For this analysis, the first two settings will be considered.

The complete experiment is split into two configurations for the two operating point settings. Both of those top-level configurations are further split into three more configurations for three different supply voltage levels: $0.1V$, $0.5V$, and $1.0V$. This results in six total sub-experiments, each consist of 100 linear and 100 non-linear simulations.

Each simulation entails applying voltage pulses to a crossbar in a similar manner as in section 4.4.1. Each voltage pulse has a width of $50\mu s$, a rise and fall time of $5\mu s$, and simulated with a $1\mu s$ time step. The wordlines to which these pulses are applied are randomized with a 50% ON/OFF ratio. The memristors in the crossbar are similarly randomized to be in LRS or HRS with a 50% ratio. Linear and non-linear simulations are randomized using the same seed to ensure accurate comparison. Finally, all line resistances are set to 1Ω.

4.5.2. Linear vs Non-linear Error

This section presents the results of the simulations described in the previous section, section 4.5.1. For this analysis, two results are considered: individual currents for memristors in the LRS state, and total accumulated currents on the bitlines. This analysis is meant to explore the differences in linear and non-linear simulation and how that impacts the readout of an crossbar operation. For this purpose, HRS current contributions are negligible.

Results are presented as the percentage deviation of linear analysis compared to non-linear analysis. This is calculated by taking the absolute difference between currents from a linear and non-linear simulation, and dividing by the non-linear current. These percentage errors are then averaged across the 100 simulations of each sub-experiment.

$0V$ Operating Point

Fig. 4.13a and 4.13b show the percentage errors between linear and non-linear simulation of LRS individual current and accumulated bitline current respectively. These figures correspond to the experiment with a $0V$ linear operating point and a supply voltage of $0.1V$. The percentage error of the LRS individual current seems to approximately be between 3.5% and 4.5%. Similarly, the percentage error of the bitline accumulated current seems to approximately be between 3.5% and 4.5%.

Fig. 4.13c and 4.13d show the percentage errors between linear and non-linear simulation of LRS individual current and accumulated bitline current respectively. These figures correspond to the experiment with a $0V$ linear operating point and a supply voltage of $0.5V$. The percentage error of the LRS individual current seems to approximately be between 3.2% and 4.2%. Similarly, the percentage error of the bitline accumulated current seems to approximately be between 3.2% and 4.2%.

Fig. 4.13e and 4.13f show the percentage errors between linear and non-linear simulation of LRS individual current and accumulated bitline current respectively. These figures correspond to the experiment with a $0V$ linear operating point and a supply voltage of $1.0V$. The percentage error of the LRS individual current seems to approximately be between 4.0% and 6.5%. Similarly, the percentage error of the bitline accumulated current seems to approximately be between 4.0% and 6.0%.

Supply Voltage Operating Point

Fig. 4.14a and 4.14b show the percentage errors between linear and non-linear simulation of LRS individual current and accumulated bitline current respectively. These figures correspond to the experiment with a linear operating point corresponding to the supply voltage and a supply voltage of $0.1V$. The percentage error of the LRS individual current seems to approximately be between 0.1% and 1.0%. Similarly, the percentage error of the bitline accumulated current seems to approximately be between 0.1% and 1.0%.

Fig. 4.14c and 4.14d show the percentage errors between linear and non-linear simulation of LRS individual current and accumulated bitline current respectively. These figures correspond to the experiment with a linear operating point corresponding to the supply voltage and a supply voltage of $0.5V$. The percentage error of the LRS individual current seems to approximately be between 0.2% and 1.2%. Similarly, the percentage error of the bitline accumulated current seems to approximately be between 0.2% and 1.2%.

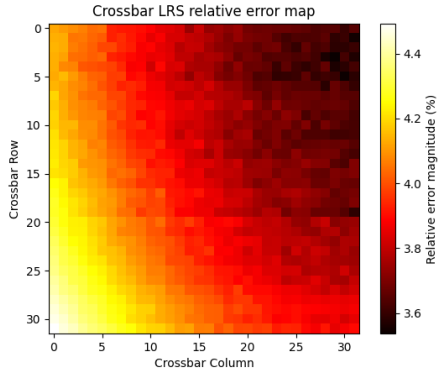
Fig. 4.14e and 4.14f show the percentage errors between linear and non-linear simulation of LRS individual current and accumulated bitline current respectively. These figures correspond to the experiment with a linear operating point corresponding to the supply voltage and a supply voltage of $1.0V$. The percentage error of the LRS individual current seems to approximately be between 0.5% and 2.5%. Similarly, the percentage error of the bitline accumulated current seems to approximately be between 0.5% and 2.5%.

4.5.3. 64x64

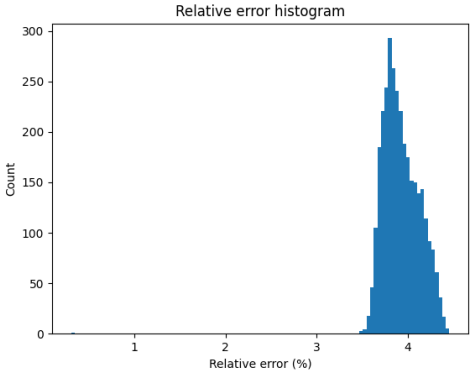
The experiments described in section 4.5.1 have all been on 32×32 crossbars. This section presents the results for similar experiment performed on 64×64 crossbars. All other parameters are identical to the experiment in section 4.5.1 except for the supply voltage, which has been limited to $0.1V$.

Fig. 4.15a and 4.15b show the percentage errors between linear and non-linear simulation of LRS individual current and accumulated bitline current respectively. These figures correspond to the experiment with a $0V$ linear operating point and a supply voltage of $0.1V$. The percentage error of the R_{ON} individual current seems to approximately be between 2% and 4.5%. Similarly, the percentage error of the bitline accumulated current seems to approximately be between 2% and 4%.

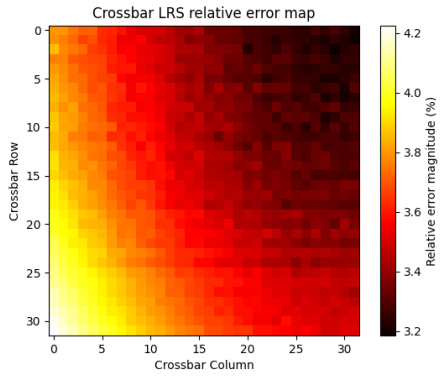
Fig. 4.15c and 4.15d show the percentage errors between linear and non-linear simulation of LRS individual current and accumulated bitline current respectively. These figures correspond to the experiment with a linear operating point corresponding to the supply voltage and a supply voltage of $0.1V$. The percentage error of the R_{ON} individual current seems to approximately be between 0.5% and 2.5%. Similarly, the percentage error of the bitline accumulated current seems to approximately be between 0.5% and 2.5%.



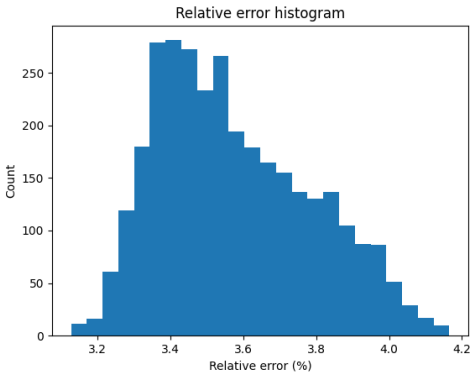
(a) LRS individual current error at $0.1V$ supply voltage



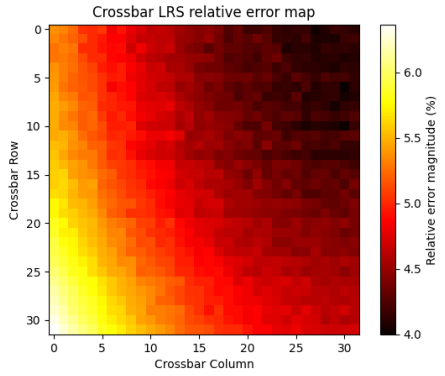
(b) Accumulated bitline current error at $0.1V$ supply voltage



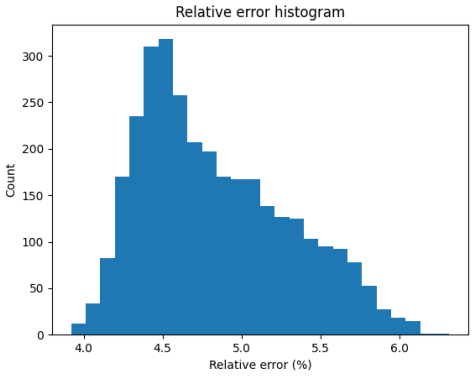
(c) LRS individual current error at $0.5V$ supply voltage



(d) Accumulated bitline current error at $0.5V$ supply voltage

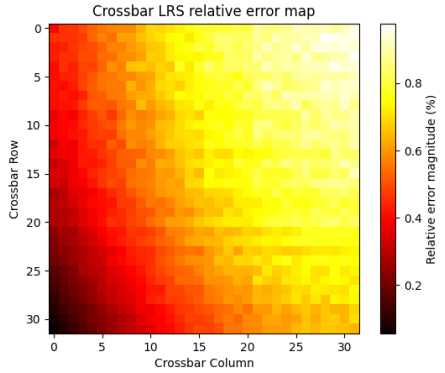


(e) LRS individual current error at $1.0V$ supply voltage

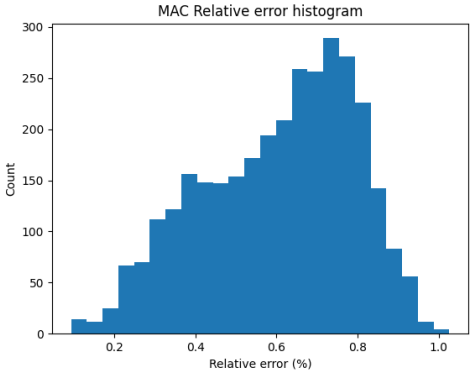


(f) Accumulated bitline current error at $1.0V$ supply voltage

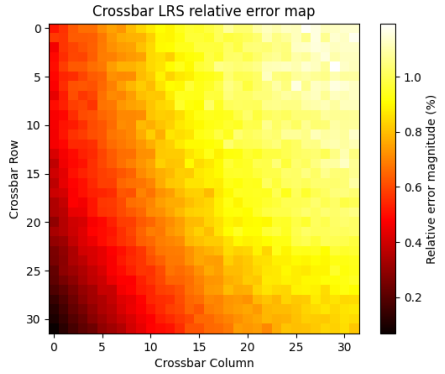
Figure 4.13: Percentage errors between linear and non-linear analysis using a $0V$ operating point



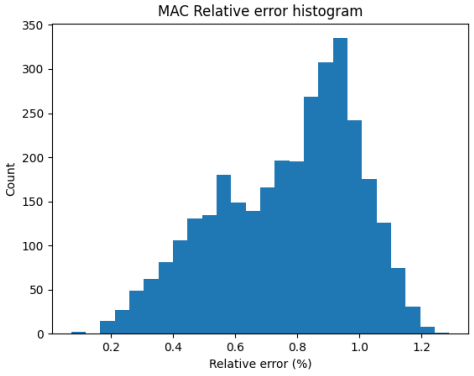
(a) LRS individual current error at $0.1V$ supply voltage



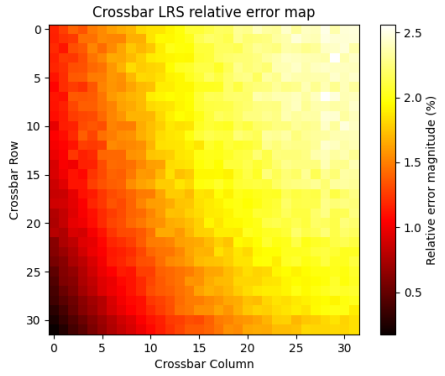
(b) Accumulated bitline current error at $0.1V$ supply voltage



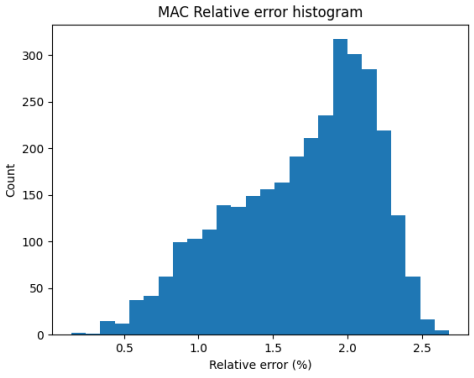
(c) LRS individual current error at $0.5V$ supply voltage



(d) Accumulated bitline current error at $0.5V$ supply voltage

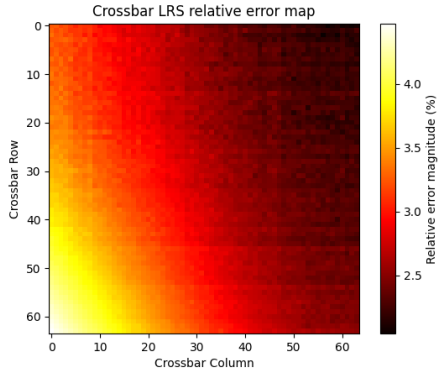


(e) LRS individual current error at $1.0V$ supply voltage

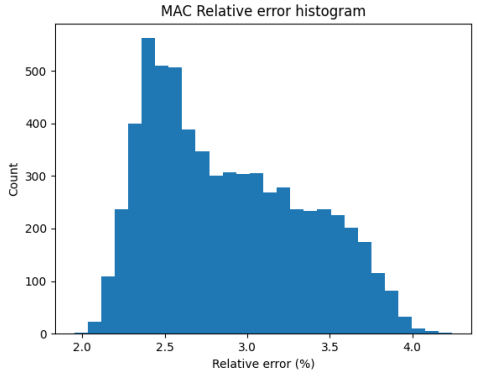


(f) Accumulated bitline current error at $1.0V$ supply voltage

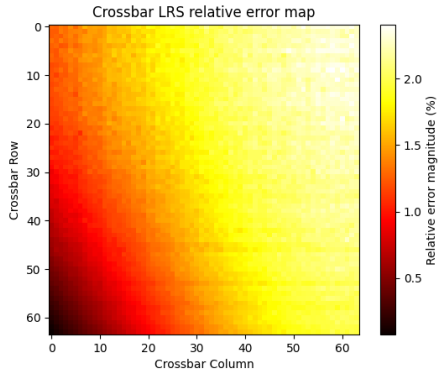
Figure 4.14: Percentage errors between linear and non-linear analysis using the supply voltage as the operating point



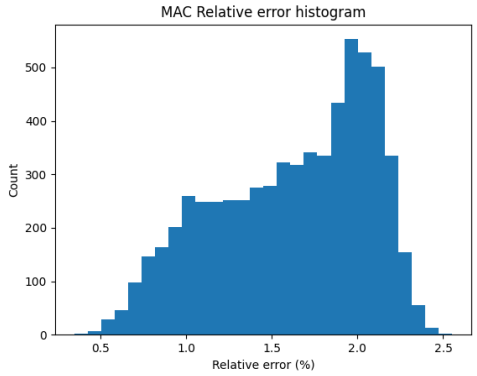
(a) LRS individual current error using a $0V$ operating point



(b) Accumulated bitline current error using a $0V$ operating point

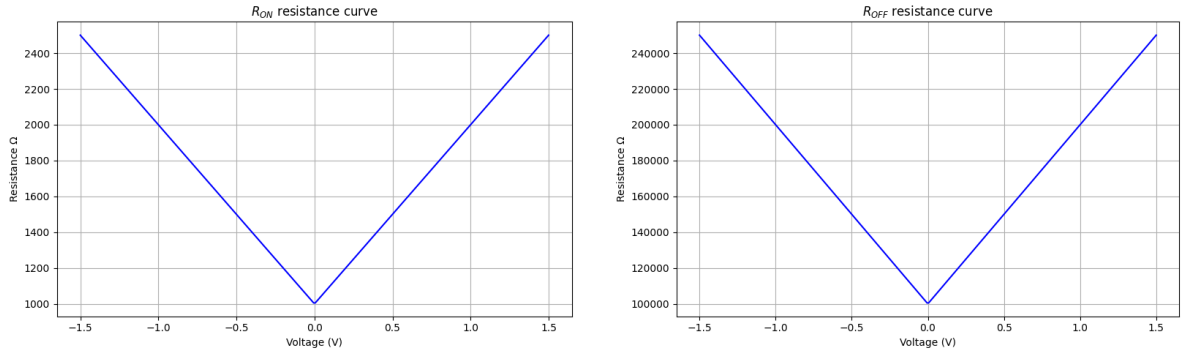


(c) LRS individual current error using the supply voltage as the operating point



(d) Accumulated bitline current error using the supply voltage as the operating point

Figure 4.15: Percentage errors between linear and non-linear analysis at $0.1V$ supply voltage



(a) R_{ON} state **(b)** R_{OFF} state

Figure 4.16: Resistance curve of a fictitious nonlinear resistor

4.5.4. Fictitious Non-linear resistor

This section will further analyze the differences between linear and non-linear simulation by replacing the memristors with fictitious non-linear resistors. These fictitious resistors linearly scale their resistance with the supply voltage according to equation 4.1. For these experiments, α is taken as 1, R_{on} is taken as $1k\Omega$, and R_{off} is taken as $100k\Omega$. Fig.4.16a and 4.16b show the resistance curves for this device. All other experimental settings are identical to the experiment in section 4.5.1.

$$R = (1 + \alpha \cdot |V|) \cdot R_{base} \tag{4.1}$$

$0V$ Operating Point

Fig. 4.17a and 4.17b show the percentage errors between linear and non-linear simulation of R_{ON} individual current and accumulated bitline current respectively. These figures correspond to the experiment with a $0V$ linear operating point and a supply voltage of $0.1V$. The percentage error of the R_{ON} individual current seems to approximately be between 4% and 10%. Similarly, the percentage error of the bitline accumulated current seems to approximately be between 4% and 9%.

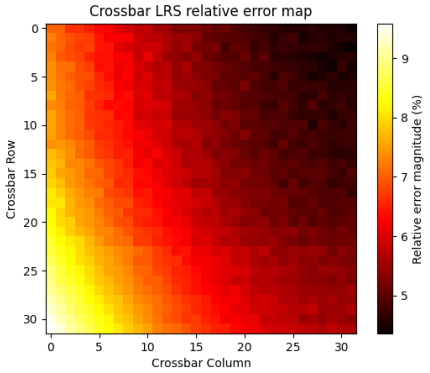
Fig. 4.17c and 4.17d show the percentage errors between linear and non-linear simulation of R_{ON} individual current and accumulated bitline current respectively. These figures correspond to the experiment with a $0V$ linear operating point and a supply voltage of $0.5V$. The percentage error of the R_{ON} individual current seems to approximately be between 25% and 50%. Similarly, the percentage error of the bitline accumulated current seems to approximately be between 25% and 45%.

Fig. 4.17e and 4.17f show the percentage errors between linear and non-linear simulation of R_{ON} individual current and accumulated bitline current respectively. These figures correspond to the experiment with a $0V$ linear operating point and a supply voltage of $1.0V$. The percentage error of the R_{ON} individual current seems to approximately be between 50% and 100%. Similarly, the percentage error of the bitline accumulated current seems to approximately be between 50% and 90%.

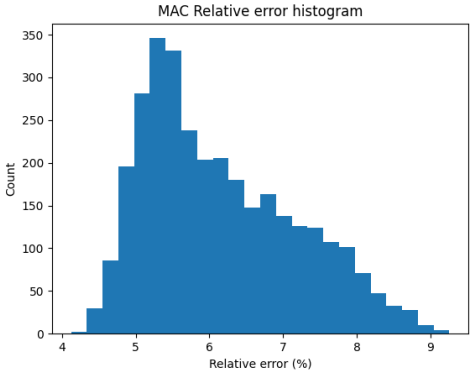
Supply Voltage Operating Point

Fig. 4.18a and 4.18b show the percentage errors between linear and non-linear simulation of R_{ON} individual current and accumulated bitline current respectively. These figures correspond to the experiment with a linear operating point corresponding to the supply voltage and a supply voltage of $0.1V$. The percentage error of the R_{ON} individual current seems to approximately be between 0.5% and 5%. Similarly, the percentage error of the bitline accumulated current seems to approximately be between 1% and 5%.

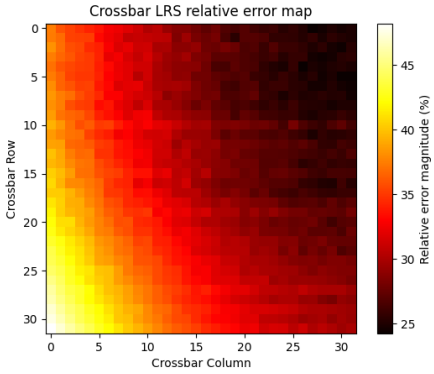
Fig. 4.18c and 4.18d show the percentage errors between linear and non-linear simulation of R_{ON} individual current and accumulated bitline current respectively. These figures correspond to the experiment with a linear operating point corresponding to the supply voltage and a supply voltage of $0.5V$. The percentage error of the R_{ON} individual current seems to approximately be between 2% and 16%.



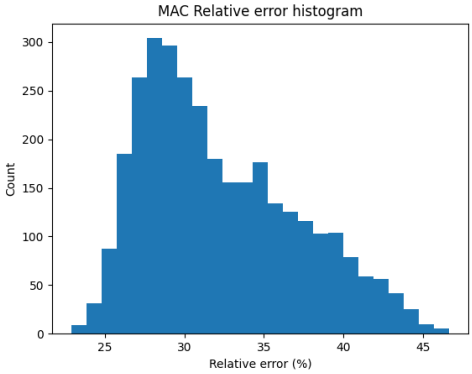
(a) R_{ON} individual current error at $0.1V$ supply voltage



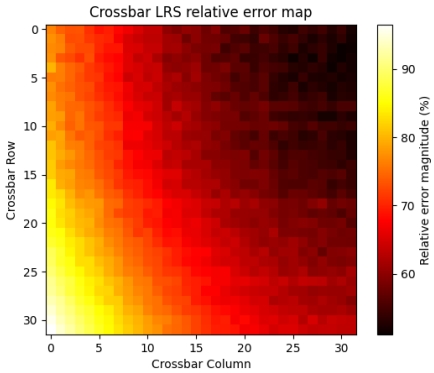
(b) Accumulated bitline current error at $0.1V$ supply voltage



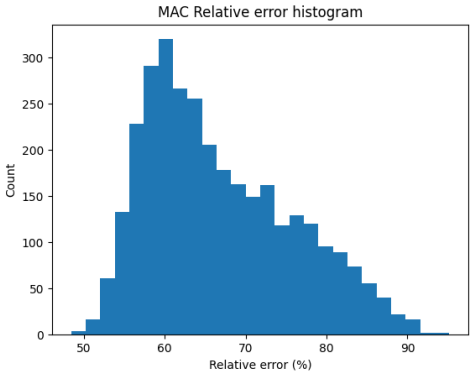
(c) R_{ON} individual current error at $0.5V$ supply voltage



(d) Accumulated bitline current error at $0.5V$ supply voltage



(e) R_{ON} individual current error at $1.0V$ supply voltage

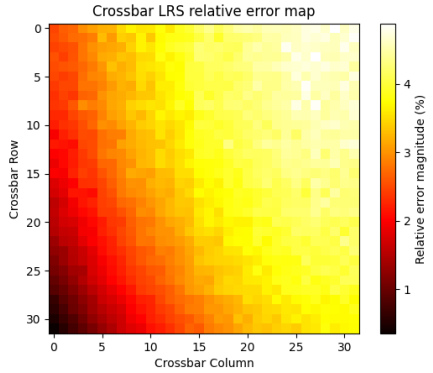


(f) Accumulated bitline current error at $1.0V$ supply voltage

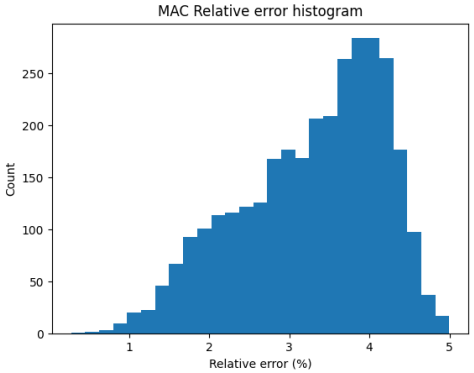
Figure 4.17: Percentage errors between linear and non-linear analysis a $0V$ operating point

Similarly, the percentage error of the bitline accumulated current seems to approximately be between 2% and 16%.

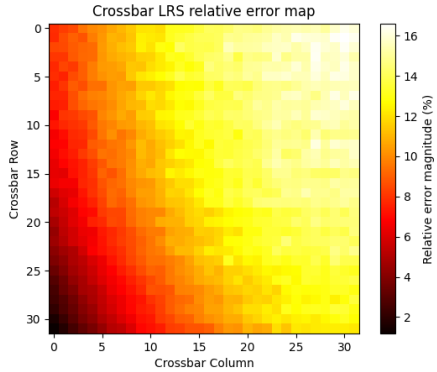
Fig. 4.18e and 4.18f show the percentage errors between linear and non-linear simulation of R_{ON} individual current and accumulated bitline current respectively. These figures correspond to the experiment with a linear operating point corresponding to the supply voltage and a supply voltage of $1.0V$. The percentage error of the R_{ON} individual current seems to approximately be between 2% and 22%. Similarly, the percentage error of the bitline accumulated current seems to approximately be between 2% and 24%.



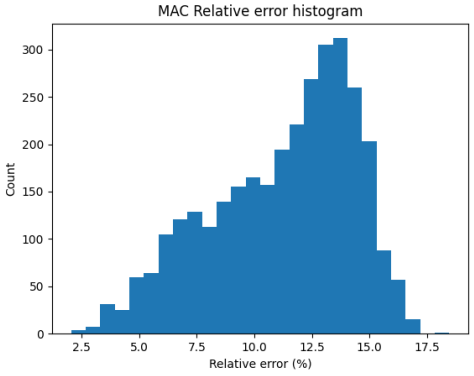
(a) R_{ON} individual current error at $0.1V$ supply voltage



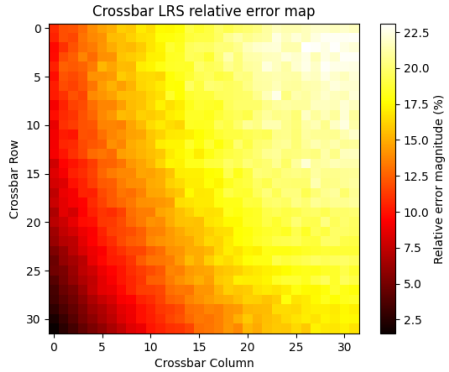
(b) Accumulated bitline current error at $0.1V$ supply voltage



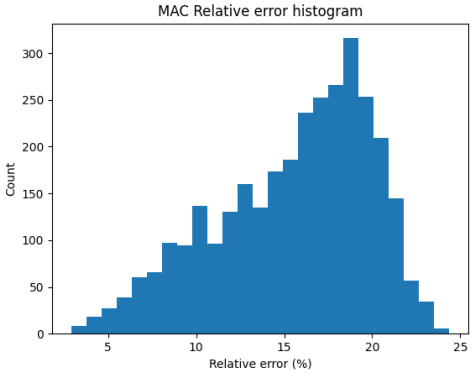
(c) R_{ON} individual current error at $0.5V$ supply voltage



(d) Accumulated bitline current error at $0.5V$ supply voltage



(e) R_{ON} individual current error at $1.0V$ supply voltage



(f) Accumulated bitline current error at $1.0V$ supply voltage

Figure 4.18: Percentage errors between linear and non-linear analysis using the supply voltage as the operating point

4.5.5. Discussion

A common approximation in memristive crossbar simulators is to assume memristive devices are linear. For example, MemTorch [33] determines the resistance of memristors independently of applied voltage when evaluating a memristive crossbar. Fig. 4.12a and Fig. 4.12b show this linear assumption not to be true; the resistance has a clear dependency on the applied voltage. However, a linear approximation for the JART VCM v1b var memristor [2] might not be entirely unreasonable. The LRS resistance, which predominantly influences the outcome of a crossbar operation, deviates by about 3% between the highest and lowest point in the $0V - 1V$ range.

Combined in a crossbar, however, errors of individual memristors influence each other and can compound to form larger deviations. An analysis of a full crossbar includes these interacting effects and could more accurately show the effects of the linear approximation of memristors. For this analysis, the non-linear simulator developed in this thesis is taken as the baseline solution. However, as discussed in section 4.4.5, compared to SPICE simulations this simulator is not perfectly accurate.

An important decision to make when approximating a non-linear device as linear is the choice of operating point: a voltage point around which the device is approximated as linear. This decision is implementation specific, thus this analysis considers common two cases: $0V$ and supply voltage.

Fig. 4.13a, 4.13c, and 4.13e show the percentage errors for a $0V$ operating point. Each graph shows the error to be largest in the bottom left and the smallest error in the top right, likely due to the location of the voltage sources and ground. Memristors in the bottom left of the crossbar are closest to both the voltage sources and ground, and thus will least be impacted by voltage drops due to line resistances. Top right memristors are consequently located furthest from sources and ground, thus experiencing the highest voltage drops. A higher voltage drops corresponds to a lower applied voltage, which in turn is closer to the operating point. Therefore, memristors placed top right of the crossbar least experience the effect of the linear approximation, while memristors placed bottom left experience it most.

Interestingly, a supply voltage of $0.5V$ seems to inhibit slightly lower error values compared to a supply voltage of $0.1V$. This is likely due to the shape of the resistance curve of LRS memristors. As can be seen in Fig. 4.12a, the resistance for an applied voltage of $0.5V$ is closer to the operating point of $0V$ than an applied voltage of $0.1V$.

Fig. 4.14a, 4.14c, and 4.14e show the percentage errors for an operating point equal to the supply voltage. Each graph shows a pattern opposite of the pattern observed at a $0V$ operating point: error values are largest for top right memristors, while smallest for bottom left memristors. This is likely due to the same reason as for the previous pattern: the location of the voltage sources, ground, and the effects of line resistance. As discussed previously, top right memristors are most effected by voltage drops due to line resistances and are therefore lowest compared to the supply voltage. In this experiment the supply voltage is also used as the operating point, thus these memristors experience most the effects of the linear approximation.

In the experiments for both operating points, the percentage error does not exceed 7%. This value is unlikely to significantly affect operations on small crossbars, but can have noticeable impact for larger crossbars. For example, if HRS current is approximated as $0A$ and LRS current at $60\mu A$ irregardless of voltage, then a 5% error results in a $3\mu A$ current deviation per LRS memristor. In order to avoid readout errors, the difference between the LRS and HRS current of a single memristor has to be detectable. In this example, the errors 20 LRS memristors in a single column would compound to $60\mu A$, after which readout errors are likely to occur. This example is idealized and does not include effects such as voltage drop, HRS current, or readout noise. However, it shows a low percentage errors could have a measurable impact, especially as crossbar size increases.

64x64

Fig. 4.15a and 4.15c show the percentage errors for a 64×64 crossbar with an operating point of $0V$ and the supply voltage respectively. Compared to the 32×32 experiment, using an operating point of $0V$ results in a similar error pattern with similar error values. Due to the increased crossbar size, top right memristors experience lower applied voltages, resulting in lower error values, while bottom left memristor errors seem nearly identical.

The 64×64 experiment where the operating point is set to the supply voltage similarly shows a similar pattern compared to its 32×32 counterpart. However, error values seem to increase significantly. A larger crossbar size inhibits larger voltage drops due to line resistances, which in turn further removes top right memristors from the operating point. Consequently, using the supply voltage as an operating point scales worse than a $0V$ operating point for increasing crossbar sizes. In contrast, the initial error when using a $0V$ operating point is higher. This suggests a tradeoff: using the supply voltage as an operating point is preferred for smaller crossbar sizes. However, at a certain point the scaling error overtakes the initial error of using a $0V$ operating point, making this choice preferable.

Fictitious Non-linear Resistor

While intended to be used with the JART VCM v1b var model [2], the simulator described in this thesis includes an abstract memristor class for the implementation of different models. Therefore, analysis of a fictitious non-linear resistor with exaggerated non-linear behaviour have been included.

Fig. 4.17a, 4.17c, 4.17e, 4.18a, 4.18c, and 4.18e each show similar error patters to their JART VCM v1b var counterparts, discussed in section 4.5.5. However, due the more extreme nature of the non-linearity, the error values themselves are significantly higher. At low voltages, these errors seem to stay below 10%, while at higher voltages error values can near 100%.

5 Conclusion

5.1. Summary

The aim of this thesis was to develop a RRAM crossbar model aimed at fast and precise simulation. To achieve this, a simulation program was implemented in C++ consisting of three sub-modules and with a clear interface, such that it can be used in larger simulation frameworks for CIM-based accelerators.

Experimental results show that the developed simulator offers a significant speedup compared to Cadence Spectre. Moreover, its accuracy is sufficiently low for the purposes of behavioral-level CIM simulations. Nevertheless, compared to Cadence Spectre simulations, several inaccuracies still persist which should be further investigated and resolved.

The modular structure of the simulation frameworks allows for straightforward maintainability and adjustments. Algorithmic improvements can be implemented in the associated sections without affecting other parts of the simulator. Furthermore, different memristor models can seamlessly be implemented using the abstract memristor class.

Compared to other simulation frameworks, the crossbar and memristor models in this framework used more closely resemble physical hardware. In particular, accurate non-linear simulation is a feature not encountered in related works.

Overall, this work presents a fast, accurate, and extendable simulation framework for RRAM crossbars. It offers a strong support tool for CIM-oriented circuit design and modeling.

5.2. Future work

This work focuses on computing output currents and resolving internal memristor states based on applied input voltages within the resistive crossbar. As such, several other simulation aspects are currently not included. Notably, this framework does not provide metrics on energy consumption. Future work could extend this model to calculate energy metrics based on the output of the current version of the simulator. Other prominent extensions include incorporating parasitic line capacitances and a propagation delay model to better capture the timing and interaction between crossbar elements.

Beyond functional extensions, several aspects of the simulator itself could be improved. Most notably, the current simulator could make more effective use of multithreading. While utilizing $3 - 5$ threads seems to be beneficial, any more significantly slows down execution time. It is also worth noting that the simulator heavily relies on mathematical solvers in several places. A more thorough study could reveal alternative solvers which are better tailored to the problem at hand, or solvers which benefits from parallelism.

Finally, the accuracy of the memristor model itself leaves room for improvement. One interesting alternative could be replacing this model with a piecewise-linear approximation based on a more accurate Cadence Spectre simulation. Such a model could be implemented as a precalculated look-up table, with intermediate values linearly interpolated during runtime.

References

[1] Arjun Tyagi and Shubham Sahay. *Assessing the Performance of Reinforcement Learning on Passive RRAM Crossbar Array*. 2023. DOI: 10.36227/techrxiv.22347277.

[2] Christopher Bengel et al. “Variability-Aware Modeling of Filamentary Oxide-Based Bipolar Resistive Switching Cells Using SPICE Level Compact Models”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 67.12 (2020), pp. 4618–4630. DOI: 10.1109/TCSI.2020.3018502.

[3] S. Menzel and C. Bengel. *JART VCM v1b var - Verilog-A*. URL: https://www.emrl.de/JART-files/JART%20VCM%201b%20veriloga-var.va.

[4] Dario Amodei and Danny Hernandez. *Ai and Compute | OpenAI*. 2018. URL: https://openai.com/index/ai-and-compute/.

[5] Timothy Prickett Morgan. *HPC market bigger than expected, and growing faster*. 2024. URL: https://www.nextplatform.com/2024/11/19/hpc-market-bigger-than-expected-and-growing-faster/.

[6] 2024. URL: https://www.hpcwire.com/2024/12/16/new-ultrafast-memory-boosts-improves-hpc-ai-workload-performance/.

[7] Amirali Boroumand et al. “Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks”. In: *Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks*. 2018, pp. 316–331. DOI: 10.1145/3173162.3173177.

[8] Onur Mutlu et al. “A modern primer on processing-in-memory”. In: *Computer Architecture and Design Methodologies* (2022), pp. 171–243. DOI: 10.1007/978-981-16-7487-7_7.

[9] Wm. A. Wulf and Sally A. McKee. “Hitting the memory wall”. In: *ACM SIGARCH Computer Architecture News* 23.1 (1995), pp. 20–24. DOI: 10.1145/216585.216588.

[10] Yuhao Ju, Yijie Wei, and Jie Gu. “A 65 nm General-Purpose Compute-in-Memory Processor Supporting Both General Programming and Deep Learning Tasks”. In: *IEEE Journal of Solid-State Circuits* 60.4 (2025), pp. 1500–1511. DOI: 10.1109/JSSC.2024.3453114.

[11] Sumon Kumar Bose, Deepak Singla, and Arindam Basu. “A 51.3-TOPS/W, 134.4-GOPS In-Memory Binary Image Filtering in 65-nm CMOS”. In: *IEEE Journal of Solid-State Circuits* 57.1 (2022), pp. 323–335. DOI: 10.1109/JSSC.2021.3098539.

[12] Debashis Panda, Paritosh Piyush Sahu, and Tseung Yuen Tseng. “A collective study on modeling and simulation of resistive random access memory”. In: *Nanoscale Research Letters* 13.1 (2018). DOI: 10.1186/s11671-017-2419-8.

[13] A. P. James and L. O. Chua. “Variability-Aware Memristive Crossbars—A Tutorial”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 69.6 (2022), pp. 2570–2574. DOI: 10.1109/TCSII.2022.3169416.

[14] L. Chua. “Memristor-The missing circuit element”. In: *IEEE Transactions on Circuit Theory* 18.5 (1971), pp. 507–519. DOI: 10.1109/TCT.1971.1083337.

[15] L.O. Chua and Sung Mo Kang. “Memristive devices and systems”. In: *Proceedings of the IEEE* 64.2 (1976), pp. 209–223. DOI: 10.1109/PROC.1976.10092.

[16] Dmitri B. Strukov et al. “The missing memristor found”. In: *Nature* 453.7191 (2008), pp. 80–83. DOI: 10.1038/nature06932.

[17] Dalibor Biolek et al. “Some fingerprints of ideal memristors”. In: *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2013, pp. 201–204. DOI: 10.1109/ISCAS.2013.6571817.

[18] Furqan Zahoor, Tun Zainal Azni Zulkifli, and Farooq Ahmad Khanday. “Resistive random access memory (RRAM): An overview of materials, switching mechanism, performance, multilevel cell (MLC) storage, modeling, and applications”. In: *Nanoscale Research Letters* 15.1 (2020). DOI: 10.1186/s11671-020-03299-9.

[19] H.-S. Philip Wong et al. “Phase Change Memory”. In: *Proceedings of the IEEE* 98.12 (2010), pp. 2201–2227. DOI: 10.1109/JPROC.2010.2070050.

[20] Dmytro Apalkov, Bernard Dieny, and J. M. Slaughter. “Magnetoresistive Random Access Memory”. In: *Proceedings of the IEEE* 104.10 (2016), pp. 1796–1830. DOI: 10.1109/JPROC.2016.2590142.

[21] Basma Hajri et al. “RRAM Device Models: A Comparative Analysis With Experimental Validation”. In: *IEEE Access* 7 (2019), pp. 168963–168980. DOI: 10.1109/ACCESS.2019.2954753.

[22] J. Joshua Yang et al. “Memristive switching mechanism for metal/oxide/metal nanodevices”. In: *Nature Nanotechnology* 3.7 (2008), pp. 429–433. DOI: 10.1038/nnano.2008.160.

[23] Matthew D. Pickett et al. “Switching dynamics in titanium dioxide memristive devices”. In: *Journal of Applied Physics* 106.7 (2009). DOI: 10.1063/1.3236506.

[24] Hisham Abdalla and Matthew D. Pickett. “SPICE modeling of memristors”. In: *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*. 2011, pp. 1832–1835. DOI: 10.1109/ISCAS.2011.5937942.

[25] Shahar Kvatinsky et al. “TEAM: ThrEshold Adaptive Memristor Model”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 60.1 (2013), pp. 211–221. DOI: 10.1109/TCSI.2012.2215714.

[26] Shahar Kvatinsky et al. “VTEAM: A General Model for Voltage-Controlled Memristors”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 62.8 (2015), pp. 786–790. DOI: 10.1109/TCSII.2015.2433536.

[27] Ximeng Guan, Shimeng Yu, and H.-S. Philip Wong. “A SPICE Compact Model of Metal Oxide Resistive Switching Memory With Variations”. In: *IEEE Electron Device Letters* 33.10 (2012), pp. 1405–1407. DOI: 10.1109/LED.2012.2210856.

[28] Kanika Monga et al. “Design of In-Memory Computing Enabled SRAM Macro”. In: *2022 IEEE 19th India Council International Conference (INDICON)*. 2022, pp. 1–4. DOI: 10.1109/INDICON56171.2022.10039958.

[29] Shubham Jain et al. “RxNN: A Framework for Evaluating Deep Neural Networks on Resistive Crossbars”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.2 (2021), pp. 326–338. DOI: 10.1109/TCAD.2020.3000185.

[30] Rui Xie et al. “A Fast Method for Steady-State Memristor Crossbar Array Circuit Simulation”. In: *2021 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*. 2021, pp. 62–64. DOI: 10.1109/ICTA53157.2021.9661817.

[31] Mohammed E. Fouda, Ahmed M. Eltawil, and Fadi Kurdahi. “Modeling and Analysis of Passive Switching Crossbar Arrays”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 65.1 (2018), pp. 270–282. DOI: 10.1109/TCSI.2017.2714101.

[32] An Chen. “A Comprehensive Crossbar Array Model With Solutions for Line Resistance and Nonlinear Device Characteristics”. In: *IEEE Transactions on Electron Devices* 60.4 (2013), pp. 1318–1326. DOI: 10.1109/TED.2013.2246791.

[33] Corey Lammie et al. “MemTorch: An Open-source Simulation Framework for Memristive Deep Learning Systems”. In: *Neurocomputing* (2022). ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2022.02.043. URL: https://www.sciencedirect.com/science/article/pii/S0925231222002053.

[34] Pai-Yu Chen, Xiaochen Peng, and Shimeng Yu. “NeuroSim: A Circuit-Level Macro Model for Benchmarking Neuro-Inspired Architectures in Online Learning”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.12 (2018), pp. 3067–3080. DOI: 10.1109/TCAD.2018.2789723.

[35] Xiaochen Peng et al. “DNN+NeuroSim: An End-to-End Benchmarking Framework for Compute-in-Memory Accelerators with Versatile Device Technologies”. In: *2019 IEEE International Electron Devices Meeting (IEDM)*. 2019, pp. 32.5.1–32.5.4. DOI: 10.1109/IEDM19573.2019.8993491.

[36] Xiaochen Peng et al. “DNN+NeuroSim V2.0: An End-to-End Benchmarking Framework for Compute-in-Memory Accelerators for On-Chip Training”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.11 (2021), pp. 2306–2319. DOI: 10.1109/TCAD.2020.3043731.

[37] Malte J. Rasch et al. “A Flexible and Fast PyTorch Toolkit for Simulating Training and Inference on Analog Crossbar Arrays”. In: *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. 2021, pp. 1–4. DOI: 10.1109/AICAS51828.2021.9458494.

[38] Christopher Bengel et al. *JART VCM v1 Verilog-A Compact Model - User Guide*. 2022. URL: https://www.emrl.de/JART-files/User_Guide_for_the_JART_VCM_v1_Compact_Model.pdf.

[39] *Eigen*. URL: https://eigen.tuxfamily.org/index.php?title=Main_Page.

A

Schottky Current Derivative

This chapter outlines the derivation of the derivative of $I_{scottky}(V)$ with respect to V for the purposes of finding the roots of a separate function using Newton's method, as described in section 3.1.3.

$I_{schottky}(V)$ is a piecewise function split along $V = 0$, as described in equations A.1 and A.2.

$$I_{Schottky,V>0}(V) = AA^*T^2 \exp(\frac{-e\phi_{B_n}(V)}{k_BT}) \cdot (\exp(\frac{eV}{k_BT}) - 1) \tag{A.1}$$

$$I_{Schottky,V<0}(V) = -A\frac{A^* \cdot T}{k_B}\sqrt{\pi W_{00} e \cdot (-V + \frac{\phi_{Bn}(V)}{\cosh^2(\frac{W_{00}}{k_BT})}} \cdot \exp(\frac{-e\phi_{Bn}(V)}{W_0})(\exp(\frac{-eV}{\epsilon'}) - 1) \tag{A.2}$$

Where $\phi_{Bn}(V)$ is described as in equation A.3.

$$\phi_{Bn}(V) = \phi_{Bn0} - e\sqrt[4]{\frac{e^3 z v_O N_{disc}(\phi_{Bn0} - \phi_n - V)}{8\pi^2 \epsilon^3_{\phi_B}}} \tag{A.3}$$

$\phi_{Bn}(V)$ has additional guards to limit to non-negative real values, as described in equations A.4 and A.5.

$$\phi_{Bn}(V) = \phi_{Bn0} \; if \; V \geq \phi_{Bn0} - \phi_n \tag{A.4}$$

$$\phi_{Bn} = 0 \; if \; \phi_{Bn} < 0 \tag{A.5}$$

Any other parameters do not depend on V thus are assumed to be constants for the sake of this derivation. A description of each parameter can be found in appendix C.

In order to derive $\frac{I_{schottky}(V)}{dV}$, each of the function segment differentiated separately. The derivative for $I_{schottky,V>0}(V)$ is derived as follows:

$$\frac{dI_{schottky,V>0}}{dV} = AA^*T^2(a'(V) \cdot b(V) + a(V) \cdot b'(V)) \tag{A.6}$$

where:

$$a(V) = \exp(\frac{-e\phi_{Bn}(V)}{k_BT}) \tag{A.7}$$

$$b(V) = \exp(\frac{eV}{k_BT}) - 1 \tag{A.8}$$

The derivative of equations A.7 and A.8 is as follows:

$$\frac{da(V)}{dV} = \exp(\frac{-e\phi_{Bn}(V)}{k_BT}) \cdot \frac{-e\phi'_{Bn}(V)}{k_BT} \tag{A.9}$$

$$\frac{db(V)}{dV} = \exp(\frac{eV}{k_BT}) \cdot \frac{e}{k_BT} \tag{A.10}$$

Combining these into equation A.6 leads to:

$$\frac{dI_{schottky,V>0}}{dV} = AA^*T^2(\exp(\frac{-e\phi_{Bn}(V)}{k_BT})\cdot\frac{-e\phi'_{Bn}(V)}{k_BT}\cdot(\exp(\frac{eV}{k_BT})-1)+\exp(\frac{-e\phi_{Bn}(V)}{k_BT})\cdot\exp(\frac{eV}{k_BT})\cdot\frac{e}{k_BT}) \tag{A.11}$$

$$AA^*T^2 \cdot \frac{e}{k_BT} \cdot \exp(\frac{-e\phi_{Bn}(V)}{k_BT}) \cdot (\exp(\frac{eV}{k_BT}) - \phi'_{Bn}(V) \cdot (\exp(\frac{eV}{k_BT}) - 1)) \tag{A.12}$$

The derivative for $I_{schottky,V<0}(V)$ is derived as follows:

$$\frac{dI_{schottky,V<0}(V)}{dV} = -A\frac{A^* \cdot T}{k_B}\sqrt{\pi W_{00}}e\cdot(c'(V)\cdot d(V)\cdot e(V)+c(V)\cdot d'(V)\cdot e(V)+c(V)\cdot d(V)\cdot e'(V)) \tag{A.13}$$

Where:

$$c(V) = \sqrt{-V + \frac{\phi_{Bn}(V)}{\cosh(\frac{W_{00}}{k_BT})}} \tag{A.14}$$

$$d(V) = \exp(\frac{-e\phi_{Bn}(V)}{W_0}) \tag{A.15}$$

$$e(V) = \exp(\frac{-eV}{\epsilon'}) - 1 \tag{A.16}$$

The derivative of equations A.14, A.15, and A.16 is as follows:

$$\frac{dc(V)}{dV} = \frac{1}{2} \cdot (-V + \frac{\phi_{Bn}(V)}{\cosh^2(\frac{W_{00}}{k_BT})})^{-\frac{1}{2}} \cdot (\frac{\phi'_{Bn}(V)}{\cosh^2(\frac{W_{00}}{k_BT})} - 1) \tag{A.17}$$

$$\frac{dd(V)}{dV} = \frac{-e}{W_0} \cdot \exp(\frac{-e\phi_{Bn}(V)}{W_0}) \cdot \phi'_{Bn}(V) \tag{A.18}$$

$$\frac{de(V)}{dV} = \frac{-e}{\epsilon'} \cdot \exp(\frac{-eV}{\epsilon'}) \tag{A.19}$$

Finally, both $I_{schottky,V>0}(V)$ and $I_{schottky,V<0}(V)$ depend on $\phi_{Bn}(V)$, the derivative of which is as follows:

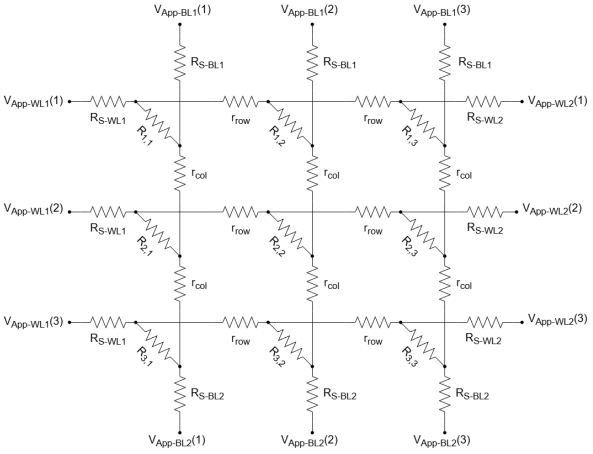
$$\frac{d\phi_{Bn}(V)}{dV} = \frac{e}{4} \cdot \sqrt[4]{\frac{e^3 z_{v_O} N_{disc}}{8\pi^2 \epsilon^3_{phi_B}}} \cdot (\phi_{Bn0} - \phi_n - V)^{-\frac{3}{4}} \tag{A.20}$$

If either guard A.4 or guard A.5 is triggered, the derivative is instead 0.

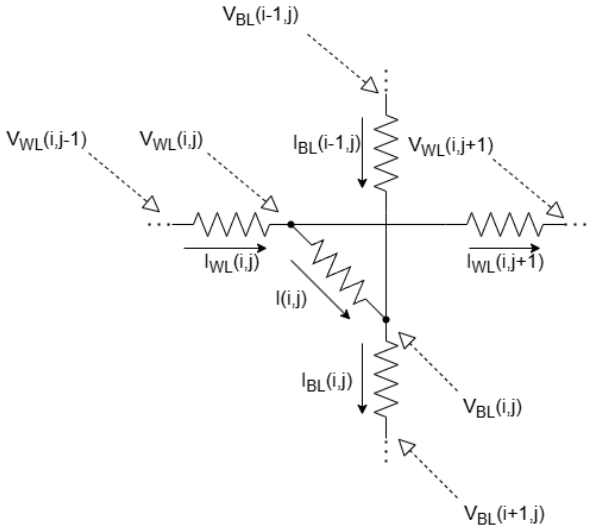
B

3x3 crossbar example

This chapter outlines a 3×3 example of the crossbar simulation method developed by [32] and outlined in section 3.2. For this example, the crossbar shown in Fig. B.1a will be used, where a crossbar node is defined as in Fig. B.1b.



(a) 3×3 crossbar circuit diagram



(b) Crossbar node circuit diagram

First Kirchhoff's current equations are applied to each node in the crossbar. These equations are subsequently rewritten in terms of voltages and reordered in a more convenient form. The result of this process is shown in equations B.1, B.2, B.3, B.4, B.5, and B.6. The full derivation for these equations is further explained in section 3.2.1.

$$V_{WL}(i,j-1)\cdot\frac{-1}{R_{WL}}+V_{WL}(i,j)\cdot(\frac{1}{R(i,j)}+\frac{2}{R_{WL}})+V_{WL}(i,j+1)\cdot\frac{-1}{R_{WL}}+V_{BL}(i,j)\cdot\frac{-1}{R(i,j)}=0$$
$$\text{for } 1\leq i\leq m \text{ and } 2\leq j\leq n-1 \quad \text{(B.1)}$$

$$V_{WL}(i,j)\cdot(\frac{1}{R_{S-WL1}}+\frac{1}{R(i,j)}+\frac{1}{R_{WL}})+V_{WL}(i,j+1)\cdot\frac{-1}{R_{WL}}+V_{BL}(i,j)\cdot\frac{-1}{R(i,j)}=V_{APP-WL1}(i)\cdot\frac{1}{R_{S-WL1}}$$
$$\text{for } 1\leq i\leq m \text{ and } j=1 \quad \text{(B.2)}$$

$$V_{WL}(i,j-1)\cdot\frac{-1}{R_{WL}}+V_{WL}(i,j)\cdot(\frac{1}{R_{S-WL2}}+\frac{1}{R(i,j)}+\frac{1}{R_{WL}})+V_{BL}(i,j)\cdot\frac{-1}{R(i,j)}=V_{APP-WL2}(i)\cdot\frac{1}{R_{S-WL2}}$$
for $1 \leq i \leq m$ and $j = n$ (B.3)

$$V_{WL}(i,j)\cdot\frac{-1}{R(i,j)}+V_{BL}(i-1,j)\cdot\frac{-1}{R_{BL}}+V_{BL}(i,j)\cdot(\frac{1}{R(i,j)}+\frac{2}{R_{BL}})+V_{BL}(i+1,j)\cdot\frac{-1}{R_{BL}}=0$$
for $2 \leq i \leq m-1$ and $1 \leq j \leq n$ (B.4)

$$V_{WL}(i,j)\cdot\frac{-1}{R(i,j)}+V_{BL}(i,j)\cdot(\frac{1}{R_{S-BL1}}+\frac{1}{R(i,j)}+\frac{1}{R_{BL}})+V_{BL}(i+1,j)\cdot\frac{-1}{R_{BL}}=V_{APP-BL1}\cdot\frac{1}{R_{S-BL1}}$$
for$i = 1$ and $1 \leq j \leq n$ (B.5)

$$V_{WL}(i,j)\cdot\frac{-1}{R(i,j)}+V_{BL}(i-1,j)\cdot\frac{-1}{R_{BL}}+V_{BL}(i,j)\cdot(\frac{1}{R_{S-BL2}}+\frac{1}{R(i,j)}+\frac{1}{R_{BL}})=V_{APP-BL2}\cdot\frac{1}{R_{S-BL2}}$$
for $i = m$ and $1 \leq j \leq n$ (B.6)

These equations can be stacked in an equation of the form:

$$G * V = E \tag{B.7}$$

Where:

$$V = [V_{WL}(1,1)\cdots V_{WL}(1,n),\cdots,V_{WL}(m,1)\cdots V_{WL}(m,n) \\ V_{BL}(1,1)\cdots V_{BL}(1,n),\cdots,V_{BL}(m,1)\cdots V_{BL}(m,n)]^T \tag{B.8}$$

$$G = G_{ABCD} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \tag{B.9}$$

$$E = \begin{bmatrix} E_W \\ E_B \end{bmatrix} \tag{B.10}$$

The resulting A, B, C, and D submatrices for a 3×3 crossbar are shown in equations B.13, B.11, B.12, and B.14 respectively. The resulting E_W and E_B subvectors for a 3×3 crossbar are shown in equation B.15 and B.16 respectively.

$$B = \begin{bmatrix} \frac{-1}{R(1,1)} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{-1}{R(1,2)} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{-1}{R(1,3)} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{-1}{R(2,1)} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{-1}{R(2,2)} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{-1}{R(2,3)} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R(3,1)} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R(3,2)} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R(3,3)} \end{bmatrix} \tag{B.11}$$

$$C = \begin{bmatrix} \frac{-1}{R(1,1)} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{-1}{R(1,2)} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{-1}{R(1,3)} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{-1}{R(2,1)} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{-1}{R(2,2)} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{-1}{R(2,3)} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R(3,1)} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R(3,2)} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R(3,3)} \end{bmatrix} \tag{B.12}$$

(B.13)

$$
A = \begin{bmatrix}
\frac{1}{R_{S-WL1}} + \frac{1}{R(1,1)} + \frac{1}{R_{WL}} & \frac{-1}{R_{WL}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{-1}{R_{WL}} & \frac{1}{R(1,2)} + \frac{1}{R_{WL}} + \frac{2}{R_{WL}} & \frac{-1}{R_{WL}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{-1}{R_{WL}} & \frac{1}{R_{S-WL2}} + \frac{1}{R(1,3)} + \frac{1}{R_{WL}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{R_{S-WL1}} + \frac{1}{R_{WL}} + \frac{1}{R_{WL}} & \frac{-1}{R_{WL}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{-1}{R_{WL}} & \frac{1}{R(2,2)} + \frac{1}{R_{WL}} + \frac{1}{R_{WL}} & \frac{-1}{R_{WL}} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{-1}{R_{WL}} & \frac{1}{R_{WS-WL2}} + \frac{1}{R(2,3)} + \frac{1}{R_{WL}} & \frac{-1}{R_{WL}} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{-1}{R_{WL}} & \frac{1}{R_{S-WL1}} + \frac{1}{R(3,1)} + \frac{1}{R_{WL}} & \frac{-1}{R_{WL}} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R_{WL}} & \frac{1}{R(3,2)} + \frac{1}{R_{WL}} + \frac{2}{R_{WL}} & \frac{-1}{R_{WL}} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R_{WL}} & \frac{1}{R_{S-WL2}} + \frac{1}{R(3,3)} + \frac{1}{R_{WL}} & \frac{-1}{R_{WL}} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R_{WL}} & \frac{1}{R_{WL}} + \frac{1}{R_{WL}} & \frac{-1}{R_{WL}} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R_{WL}} & \frac{1}{R_{WL}} + \frac{1}{R_{WL}} & \frac{-1}{R_{WL}} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R_{WL}} & \frac{1}{R_{WL}} + \frac{1}{R_{WL}}
\end{bmatrix}
$$

(B.14)

$$
D = \begin{bmatrix}
\frac{1}{R_{S-BL1}} + \frac{1}{R(1,1)} + \frac{1}{R_{BL}} & \frac{-1}{R_{BL}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{-1}{R_{BL}} & \frac{1}{R_{S-BL1}} + \frac{1}{R(1,2)} + \frac{1}{R_{BL}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{-1}{R_{BL}} & \frac{1}{R_{S-BL1}} + \frac{1}{R(1,3)} + \frac{1}{R_{BL}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{R(2,1)} + \frac{2}{R_{BL}} & \frac{-1}{R_{BL}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{-1}{R_{BL}} & \frac{1}{R(2,2)} + \frac{2}{R_{BL}} & \frac{-1}{R_{BL}} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{-1}{R_{BL}} & \frac{1}{R(2,3)} + \frac{1}{R_{S-BL2}} + \frac{1}{R_{BL}} & \frac{-1}{R_{BL}} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{-1}{R_{BL}} & \frac{2}{R_{BL}} + \frac{1}{R(3,1)} + \frac{1}{R_{BL}} & \frac{-1}{R_{BL}} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R_{BL}} & \frac{1}{R_{S-BL2}} + \frac{1}{R(3,2)} + \frac{1}{R_{BL}} & \frac{-1}{R_{BL}} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R_{BL}} & \frac{1}{R_{S-BL2}} + \frac{1}{R(3,3)} + \frac{1}{R_{BL}} & \frac{-1}{R_{BL}} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R_{BL}} & \frac{1}{R_{BL}} + \frac{1}{R_{BL}} & \frac{-1}{R_{BL}} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R_{BL}} & \frac{1}{R_{BL}} + \frac{1}{R_{BL}} & \frac{-1}{R_{BL}} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-1}{R_{BL}} & \frac{1}{R_{BL}}
\end{bmatrix}
$$

$$\begin{bmatrix} \frac{V_{APP-WL1}(1)}{R_{S-WL1}} \\ 0 \\ \frac{V_{APP-WL2}(1)}{R_{S-WL2}} \\ \frac{V_{APP-WL1}(2)}{R_{S-WL1}} \\ 0 \\ \frac{V_{APP-WL2}(2)}{R_{S-WL2}} \\ \frac{V_{APP-WL1}(3)}{R_{S-WL1}} \\ 0 \\ \frac{V_{APP-WL2}(3)}{R_{S-WL2}} \end{bmatrix} \tag{B.15}$$

$$\begin{bmatrix} \frac{V_{APP-BL1}(1)}{R_{S-BL1}} \\ 0 \\ \frac{V_{APP-BL1}(2)}{R_{S-BL1}} \\ \frac{V_{APP-BL1}(3)}{R_{S-BL1}} \\ 0 \\ 0 \\ 0 \\ \frac{V_{APP-BL2}(1)}{R_{S-BL2}} \\ \frac{V_{APP-BL2}(2)}{R_{S-BL2}} \\ \frac{V_{APP-BL2}(3)}{R_{S-BL2}} \end{bmatrix} \tag{B.16}$$

C

Memristor Model Paremeters

[38]

Symbol	Description	Value	Unit
π	Pi	3.1415927	
e	Charge of an electron	1.6022e-19	C
k_B	Boltzmann's constant	1.38065e-23	J/K
ϵ_0	Permittivity of a vacuum	8.65419e-12	F/m
A^*	Effective Richardson constant	6.01e5	A/m^2K^2
m^*	Electron rest mass	9.10938e-31	kg
zv_O	Oxygen vacancy charge number	2	

Table C.1: Memristor model constants

Symbol	Description	Default	Range	Unit
T_0	Ambient temperature	293	[100:500]	K
ϵ_s	Static Permittivity	17	[10:25]	-
$\epsilon_{\phi_{\text{Bn0}}}$	Permittivity related to the Schottky effect	5.5	[1:10]	-
ϕ_{Bn0}	Nominal Schottky barrier height	0.18	[0.1:1.5]	eV
ϕ_n	Energy level difference between the Fermi level in the oxide conduction band edge	0.1	[0.1:ϕ_{Bn0}]	eV
μ_n	Electron Mobility	4e-6	[1e-6:1e-5]	$\text{m}^2/(\text{Vs})$
$N_{\text{disc,max}}$	Maximum oxygen vacancy concentration in the disc	20	[0.001;1100]	$10^{26}/\text{m}^3$
$N_{disc,min}$	Minimum oxygen vacancy concentration in the disc	0.008	[0.0001;100]	$10^{26}/\text{m}^3$
N_{init}	Initial oxygen vacancy concentration in the disc	0.008	[0.0001;1000]	$10^{26}/\text{m}^3$
N_{plug}	Initial oxygen vacancy concentration in the plug	20	[0.001;100]	$10^{26}/\text{m}^3$
a	Ion hopping distance	0.25e-9	[1e-10;1e-9]	m
v_0	Attempt frequency	2e13	[1e10;1e14]	Hz
ΔW_{A}	Activation energy for ion hopping	1.35	[0.8;1.5]	eV
R_{th0}	Thermal resistance	1e7	[1e6;2e7]	W/K
r_{det}	Radius of the filament	45e-9	[5e-9;100e-9]	m
l_{cell}	Length of the filament	3	[2;5]	nm
l_{det}	Length of the disc	0.4	[0.1;l_{cell}]	nm
$R_{\text{theff,scaling}}$	Scaling factor fo the thermal resistance for gradual RESET	0.27	[0.1;1]	-
R_0	Line resistance for the current of 0 A	719.2437		Ω
$R_{\text{th,line}}$	Thermal resistance of the lines	90471.47		W/K
α_{line}	Temperature coefficient in the lines	3.92e-3		1/K
A	Filament area	$\pi * r_{det}^2$		m^2
cv_O	Average concentration of the plug and disc regions	$\frac{N_{plug}+N_{disc}}{2}$		$10^{26}/\text{m}^3$

Table C.2: Memristor model parameters [38]

D

GitHub

Link to GitHub repository: https://github.com/bsmeele/master-thesis

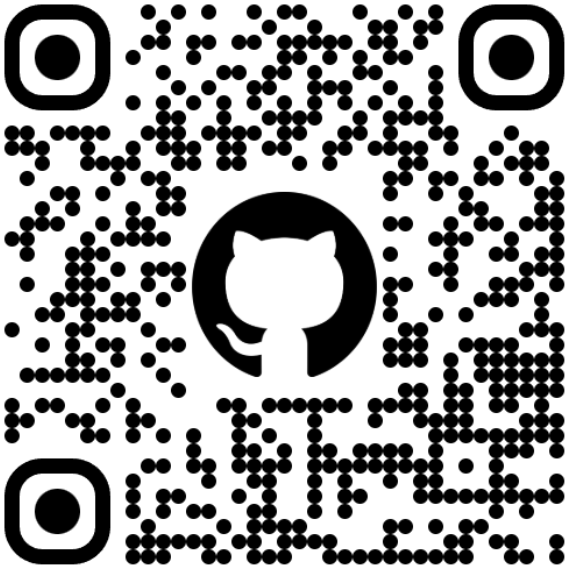


Figure D.1: QR-code to GitHub repository