Spring 1-1-2012

Memristor-based Reservoir Computing

Manjari S. Kulkarni
Portland State University

10.15760/etd.899

Memristor-based Reservoir Computing

by

Manjari S. Kulkarni

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Thesis Committee:
Christof Teuscher, Chair
Dan Hammerstrom
Roy Kravitz

Portland State University
2012

Abstract

In today's nanoscale era, scaling down to even smaller feature sizes poses a significant challenge in the device fabrication, the circuit, and the system design and integration. On the other hand, nanoscale technology has also led to novel materials and devices with unique properties. The memristor is one such emergent nanoscale device that exhibits non-linear current-voltage characteristics and has an inherent memory property, i.e., its current state depends on the past. Both the non-linear and the memory property of memristors have the potential to enable solving spatial and temporal pattern recognition tasks in radically different ways from traditional binary transistor-based technology. The goal of this thesis is to explore the use of memristors in a novel computing paradigm called "Reservoir Computing" (RC). RC is a new paradigm that belongs to the class of artificial recurrent neural networks (RNN). However, it architecturally differs from the traditional RNN techniques in that the pre-processor (i.e., the reservoir) is made up of random recurrently connected non-linear elements. Learning is only implemented at the readout (i.e., the output) layer, which reduces the learning complexity significantly. To the best of our knowledge, memristors have never been used as reservoir components. We use pattern recognition and classification tasks as benchmark problems. Real world applications associated with these tasks include process control, speech recognition, and signal processing. We have built a software framework, RCspice (Reservoir Computing Simulation Program with Integrated Circuit Emphasis), for this purpose. The framework allows to create random memristor networks, to simulate and evaluate them in Ngspice, and to train the readout layer by means of Genetic Algorithms (GA). We have explored reservoir-related parameters, such as the network connectivity and the reservoir size along with the GA parameters.

Our results show that we are able to efficiently and robustly classify time-series patterns using memristor-based dynamical reservoirs. This presents an important step towards computing with memristor-based nanoscale systems.

Acknowledgements

Completion of this thesis has been a journey with unforgettable moments of stress, anxiety and final moments of my thesis defense has molded into sense of accomplishment and success. I take this opportunity to thank all those who have provided invaluable support and guidance throughout this journey.

First and foremost I would like to thank my adviser, Professor Christof Teuscher, for generously providing guidance on the thesis topic, continuously encouraging me even when at times the results were not as expected, and for all the patience and support given to me since day one for my thesis completion. I would also like to thank Professor Dan Hammerstrom and Professor Roy Kravitz for serving on my defense committee. Your insights and comments are very much appreciated.

I would like to sincerely thank my husband for his support throughout my Master's at Portland State University. Support of my family and their interest in "Reservoir Computing and Memristor" has led to various discussions that have enabled me to think and understand this topic in a different light.

This work was partly sponsored by the National Science Foundation, (grant number 1028378) and the Sigma Xi Grants-in-Aid of Research program.

Contents

List of Tables

List of Figures

1

Overview

1.1 Introduction and Motivation

Ongoing technology scaling not only results in increased silicon and system complexity but also in nanoscale devices with interesting new properties. The current *International Technology Roadmap for Semiconductors* (ITRS) [8], highlights advancements in non-silicon nanoelectronic devices, which include carbon nanotube field-effect transistors (FETs), graphine nanoribbion FETs, nanowires and molecular electronics. In the area of memory devices, the research is focused on Ferroelectric FET memories (capacitance based), spin torque, nanomechanical and nanoionic, and redox reaction based memories (resistance based) [8].

The fabrication and circuit design of these nanoscale devices is a challenge as it is increasingly difficult to control their exact orientation and assembly. Designing complex circuit topologies using these nanodevices or materials requires a major shift from conventional design techniques and fabrication methods to new techniques. The semiconductor industry is considering exploring alternative ways of computing, for example by implementing stochastic computing techniques [8] and using self-assembly processes instead of top-down lithography [9]. A hybrid architecture approach that uses conventional silicon with non-conventional nanoscale storage devices, and application-based reconfigurable nanoelectronic circuits is discussed in [10, 11]. An adaptive programming technique of *randomly assembled computer* (RAC) built from diodes as a computational element is explored in [12].

Liquid State Machines (LSMs) and *Echo State Network* (ESN) are biologically-inspired computation architectures, which are explored in [13, 14].

Applications for nanoscale devices range from signal processing, low power reconfigurable logic to non-volatile logic [8]. Another application consists in implementing dense circuit architectures that can mimic a certain brain functionality. This particular application is possible due to a novel nanodevice, which is functionally equivalent to a 'synapse' [15].

1.2 Challenges

Although complementary metal-oxide-semiconductor (CMOS) technology has well-established design and fabrication techniques, it is also facing challenges with device reliability, lifetime and an ongoing increase in non-recurring engineering (NRE) fabrication cost as a result of technology scaling [8, 10]. Nanoelectronic, as an emerging technology naturally comes with a set of challenges due to the lack of well-established design and fabrication methodologies. The challenges can be summarized as follows:

1. At nanoscale, it is less likely that every device fabricated will have the exact predefined tolerance and precision. Hence, one of the challenges lies in developing low-cost and tolerance-driven fabrication techniques, which can be applied to a wide range of nanoscale devices.

2. Designing circuit topologies and architectures to tolerate high variation due to device fabrication techniques [10].

3. Defining new computing techniques that will help to harness the novel device characteristics.

4. Defining accurate device characteristic models which allow multi-level simulation and abstraction [8].

5. Integration complexity in nanoelectronic results due to hierarchical and hybrid architecture approach. Thus, defining a suitable interconnect technology compatible across platforms is a challenge [8].

6. Keeping power consumption to a minimum and employing effective heat dissipation techniques.

1.3 Research Questions

Based on the challenges outlined in Section 1.2, we formulated the following research questions:

1. Using self-assembly as one of the nanoscale fabrication techniques, researchers have demonstrated the fabrication of two terminal nanodevices [10]. These individual self-assembled devices can be assembled to form a random network topology. If we are given such a random self-assembled network of two terminal nanodevices, how can we extract meaningful computation from such a network?

2. Which computational architecture should be implemented to explore the computational capability of emerging nanodevices?

3. Which application areas can benefit from a random network of nanodevices?

1.4 Our Approach in a Nutshell

To design optimal circuits, well-established evolutionary computing techniques have been used in areas ranging from analog filter circuit design [6] to evolving memristor-based circuits [16]. In recent years, a novel computing technique called *Reservoir Computing* (RC) [14] has been explored for real time computation with dynamical systems. This approach uses randomly generated networks as the compute core. The uniqueness of this approach is that the compute core, i.e., the network, does not have to be trained for specific tasks. Rather, the network dynamics are interpreted by a simple output layer. This approach allows performing computations with circuits that do not require a well-defined topology and also enables spatial computing [13, 14].

My contributions in this thesis are as following:

1. We proposed that memristors can be used as a dynamical element in reservoir computing (RC).
2. We experimentally showed that a memristor-based reservoir can be used to perform non-trivial tasks, such as pattern recognition.
3. We developed *RCspice*, a Matlab-based framework to implement memristor based reservoir computing (see Chapter 3 and 4 for details).
4. We use Ngspice [17] simulator to simulate memristor-based reservoirs.
5. We optimized the reservoir performance by exploring reservoir parameters (see Chapter 7.1 for details).
6. We performed experiments to evaluate the proposed memristor-based reservoir with different network sizes (see Chapter 7.1 for details).
7. We performed various pattern recognition experiments with different inputs signals, such as triangular-square, frequency modulated, and amplitude modulated signals. Our experiments showed that the memristor reservoir is able to distinguish between signal variation in the input (see Section 7.2, 7.4 and 7.5 for details).
8. We performed an associative memory experiment that demonstrated learning behavior related to specific input events (see Section 7.6 for details).
9. We published a paper titled "Evolving nanoscale associative memories with memristors" at the IEEE NANO 2011 conference [16].

1.5 Thesis Organization

In this thesis we implement RC architecture for exploring the computation abilities of random memristor networks. The thesis organization is as follows:

We start with the introduction, Chapter 2.2 gives background information on the *reservoir computing* (RC) by establishing its links to *Artificial Neural Networks* (ANNs). The literature review under Section 2.3 gives a brief introduction to the RC architecture. Section 2.4 gives an overview of our choice of a novel nanoscale device, the memristor, and states its properties that makes it an ideal candidate for RC.

Chapter 3 describes our Matlab-based *Reservoir Computing Simulation Program with Integrated Circuit Emphasis* (RCspice) simulation framework. The framework's sub-modules are described in the Chapter 4. In the Chapter 5 we describe the reservoir evaluation using Ngspice and the genetic algorithms.

Chapter 7 explains the experiments. Section 7.1 shows the parameter exploration relating to the memristor reservoir and the genetic algorithms. Sections 7.2, 7.4 and 7.5 explain the experiments conducted for the pattern recognition experiment. Section 7.6 and 7.7 covers the associative memory experiment and the logic computation experiment respectively. The final Chapter 8 concludes the thesis and discusses future work.

2

Background

2.1 Non-classical Computing Paradigms

2.2 Artificial Neural Networks

The field of neurocomputing is inspired from the very idea of our brain's ability to process information. The fundamental computing units of our nervous system are *neurons* and the *synapses* are the channels via which they communicate to other neurons. Our brain is a dynamic computational core, i.e., it is not wired for a specific task but in fact, reusing, rearranging and modifying the given existing brain structure gives rise to varied computational ability [18, 19].

The area of *Artificial Neural Networks* (ANNs), or simply *neural networks*, is inspired from our brain's computational ability. ANNs mimic biological neural networks that are capable of performing numerous computational tasks. There exist various mathematical models that approximate the behavior of the basic biological computing unit (i.e., the neuron). In ANN's terminology, each mathematical modeled neuron is called a 'unit' [20]. One of the first artificial neuron model proposed by McCulloch-Pitts is a simple two-state neuron model [21]. More realistic integrate-and-fire models, also called spiking neuron models, are designed to describe and predict biological processes (a more detailed description of these models can be found in [18, 19]).

A single 'unit' can transform an incoming signal into an output signal, but solving computational tasks requires arranging the units in a particular network topology. Neural circuits in our brain are recurrently connected and the information processing is typically of temporal nature (i.e., the outcome due to a particular stimuli is its integration over a time period) [22]. Thus, in ANNs, the units are arranged to form networks, such as the feedforward and recurrent neural networks (RNNs). In the feedforward network, the flow of information takes place only in one direction (i.e., connections are directed only from the input to the output). In RNNs, the network topology forms feedback connections. An important characteristic of RNNs is that due to the feedback associated with the network topology, they develop internal temporal dynamics (i.e., memory).

ANNs can solve tasks ranging from basic logical operations, such as AND, OR and XOR to more complex tasks, such as associative computation, pattern recognition and content addressable memory [18, 19, 21].

2.3 Reservoir Computing

Recently, two new computing paradigms were introduced: *Liquid State Machines* (LSMs) [23] by Wolfgang Maass and the *Echo State Network* (ESN) [14] by Herbert Jaeger. Both models represent a new class of computing models inspired by RNNs. The overarching term for these paradigms is *Reservoir Computing* (RC) [14,23,24].

The major difference between these two architectures is the mathematical modeling of the basic computing 'unit' (i.e., neuron). In LSMs, the 'unit' is modeled as a *spiking integrate-and-fire* neuron while the ESN architecture implements 'unit' as sigmoid (i.e., as *tanh* transfer function). The spiking neuron model closely resembles the spiking nature of biological neurons and thus retains the essential neuron-behavior. Hence, LSM applications are mainly focused to provide a biologically-plausible architecture for generic cortical microcircuits computations [24, 25]. While *tanh* being a non-linear function, ESN finds applications in a number of engineering tasks [14, 24]. Although both these architectures find their applications in different research areas, an important attribute is that the computation is performed on a large non-linear dynamic recurrent network. These dynamical networks need not be strictly RNNs, but any medium which contains dynamical properties can be used for implementing LSMs and ESNs. Section 2.3.3 gives an overview of the various dynamical mediums used as reservoirs. The following Sections (2.3.1 and 2.3.2) will give a brief overview of the LSM and ESN architectures respectively.

2.3.1 Liquid State Machines

Liquid State Machines (LSMs) are a novel computational framework recently introduced in [23]. The computing core is a recurrent neural network (RNN). The computing core is referred to as the 'liquid'; the term liquid is a symbolic representation of the RNN's dynamic nature. In the 'liquid', each computing 'unit' is defined as a mathematical model of *spiking integrate-and-fire* neuron. These neuron models closely resemble functioning of the biological neurons. Thus, the LSM framework is capable of performing real-time computations on time-varying inputs. Its applications are mainly focused in the area of neural microcircuits [13,23]. The LSM's architecture can be divided into three distinct sections namely; (i) the pre-processor unit called as the 'liquid', (ii) the liquid's state space (i.e., memory), and (iii) the processing layer called the 'readout layer'.

A diagrammatic representation of the LSM framework is shown in Figure 2.1. The striking feature of the LSM's architecture is that the 'liquid' itself does not have to be trained for particular tasks, but only the readout layer is trained to extract the time-varying information from the liquid. Maass experimentally shows that the readout layer can be trained to perform a temporal pattern recognition task [23].

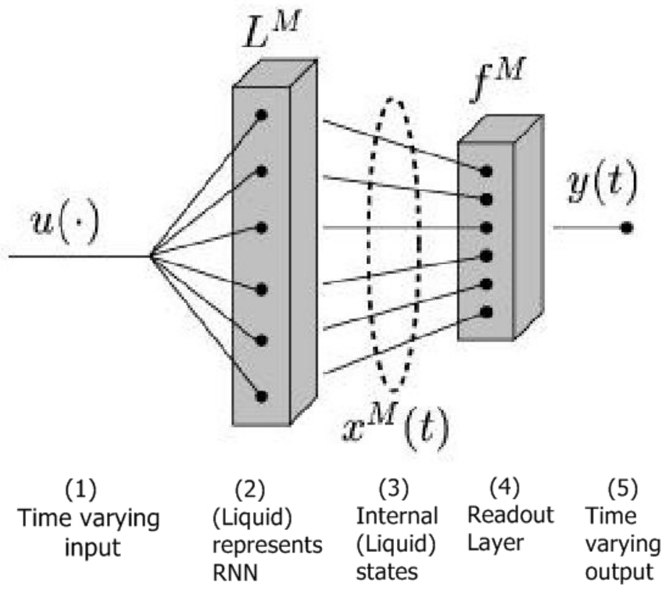


Figure 2.1: Architecture of a *Liquid State Machine*. The input u(.)(1), is a time varying input applied to the liquid L^M(2). The internal liquid states at time(t) are represented by x^M(t)(3). These states are transformed by a readout layer f^M(4) to produce output y(t)(5) [23].

2.3.2 Echo State Network

Echo State Network (ESN) is a recently introduced architecture based on recurrent neural networks (RNN) [14]. The ESN is architecturally similar to LSM and can similarly be divided into three distinct sections; (i) the pre-processor unit called the 'reservoir', (ii) the internal state space (i.e., memory) of the reservoir and, (iii) the output layer or the 'readout' layer. Each computing 'unit' in the 'reservoir' is defined as mathematical model of *tanh* function. Similar to LSMs, the readout layer is trained for a given task. The architecture of the *Echo State Network* is similar to LSM as represented in Figure 2.1.

2.3.3 Types of Reservoirs

After a brief introduction to the LSM and the ESN architectures (see Section 2.3.1 and 2.3.2 for details), two questions that remain to be answered are:

1. What makes a good dynamical reservoir?
2. What tasks can be solved by a *liquid* or *reservoir*?

This section answers the above questions by summarizing recently published articles that use unique dynamical *reservoirs or liquids* (i.e., compute cores) for implementing LSMs or ESNs. Fernando *et al.* in [2] literally used water as a 'liquid'. Water as dynamical medium is a natural agent, which incorporates information over time without the use of any mathematical model required to store information over time. Figure 2.2 shows water as a unique pre-processor. Here, the water is placed in a glass tank, which is simulated using an electric motor. This motor action causes ripples in the water, which are read using the optical setup and are transformed by the output layer to solve tasks, such as XOR and the speech recognition [2]. In contrast to water as a liquid, one of the recent publication [26] implements a hard-liquid, i.e., a general purpose mixed-mode artificial neural network as a *Application Specific Integrated Circuit* (ASIC) is configured as a liquid. In [27], Jones *et al.* implement a LSM using a model of the *Gene Regulation Network* (GRN) of Escherichia Coli as a 'liquid'. Photonic reservoir computing is implemented by configuring a network of coupled *Semiconductor Optical Amplifiers* (SOA) as a 'reservoir' in [28]. One recently published article [29] implements *optoelectronic reservoir computing.*

Figure 2.2: In [2], Fernando *et al.* used water as a pre-processor ('liquid') for LSM implementation. (Source: [3])

From the recent flurry of publications in the area of reservoir computing, the platforms that can be used as *reservoir* or *liquid* range from ANN, optoelectronic, optics, real water to VLSI [2,14,26–30]. Reservoir computing is capable of solving non-trivial tasks like speech recognition and robot control [30].

As seen from the above applications, reservoir computing is a powerful computational tool for performing complex real time computations on continuous input streams. Their performance measure is based on two properties, separation and approximation. If LSM or ESN, have different internal states for two different input sequences then the liquid or the reservoir is said to have the *separation property*. The distance between different states is generally measured using eucidian distance [23] or using hamming distance [20]. This property determines how well can a liquid classify between inputs with different input history. An *approximation property* is the measure of the readout capability to produce a desired output from the given liquid states [14,23].

2.4 Memristor

This section presents a brief overview of the nanoscale memristor device, which we use as a building block in this thesis. *Memristor*, short for memory resistor [31], is a passive two-terminal circuit element which was theoretically postulated in 1971 by Professor Leon Chua. He also demonstrated memristive behavior using active circuits in his seminal paper on memristors [31].

The theory of the electrical circuits deals with three fundamental circuit elements namely resistor (R), capacitor (C) and inductor (L) which are defined using relationships between the four fundamental variables, namely current (i), voltage (v), charge (q) and magnetic flux (ϕ). We all are familiar with the relationship between these four variables: (i) the charge is defined by the time integral of the current (ii) the flux is defined by the time integral of the voltage (iii) R is defined by v/i (iv) C is defined by q/v (v) L is defined by v = L (di/dt). From the symmetry point of view, Chua in [31] put forward the missing relationship between (q) and (ϕ) as shown in Equation 2.1. The relation between the four fundamental variables is shown in Figure 2.3.

In 2008, the first physical device with a memristive property was realized by HP [15]. This device is a 40nm cube of titanium dioxide (TiO_2) sandwiched between platinum conducting plates. An external voltage is applied across these two conducting plates. The device structure is composed of two layers, the upper half of the device is a TiO_2 layer, which is devoid of 0.5 percentage of its oxygen vacancies (TiO_{2-x}). These mobile vacancies makes the region more conductive representing R_{on} and the lower half has a perfect 2:1 oxygen to titanium ratio making it a perfect insulator representing R_{off}. Thus, the device varies its internal resistance based on the doping vacancy distribution [15]. A representation of HP's memristor

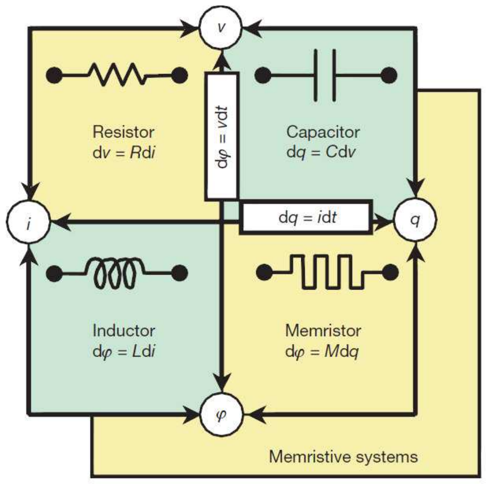


Figure 2.3: Four fundamental variables, charge, flux, voltage and current defining the circuit elements resistance, memristance, inductor and capacitor. The memristor is equivalent to the resistor when the rate of change of flux ϕ with respect to charge q is constant. (Source: [4]).

is shown in Figure 2.4.

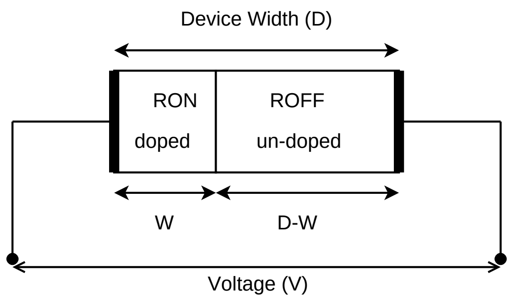


Figure 2.4: Representation of the TiO_2 memristor device. The doped region represents low resistance R_{on} and the un-doped region represents high resistance R_{off}. (Source: redrawn from [4].)

HP's memristor belongs to a broader class of nonlinear dynamical systems called memristive system [32] that follows a more generalized definition where the memristor voltage is dependent on the variation in the doped region (w) at a given point in time(t) (see Equation 2.2), as opposed to the more specific definition based on flux and charge relation as defined by Chua (see Equation 2.1), in which memristance M is a function of q. For HP's memristance definition, w is the physical quantity responsible for change in the internal state of the device. The *memory property* of the memristor is due to the charge that has passed through it defined by its effective resistance M(q) [15]. Similar to resistance, *memristance* is measured in ohms (Ω).

$$d\phi = M dq \Leftrightarrow v = M(q)i \tag{2.1}$$

$$v(t) = R(w)i \tag{2.2}$$

Strukov *et al.* in [4] demonstrated the current-voltage $I-V$ hysteretic behavior. This is shown in Figure 2.5. Sinusoidal voltage Vsin(ω_0t) across the memristor device causes nonlinear change in the current. The change in the applied voltage across the device causes the boundary between the R_{on} and R_{off} regions to change, which is due to the charged dopant drift [4], thus changing the device conductivity. The memristor characteristics are frequency dependent, as shown in Figure 2.5. As the frequency is increased from ω_0 to $10\omega_0$, the hysteresis characteristics is no longer valid and the device operates in a linear regime.

This passive two-terminal nanoscale device with a nonlinear characteristics can be integrated with the current CMOS technology. Thus, memristors find applications in the area of non-volatile memory [4], programmable logic arrays [32], analog computation and specific scientific research is concentrated on using memristors to mimic synapses for cognitive computing [15]. Pershin *et al.* [7] have demonstrated associative memory behavior using a simple neural network with memristors as synapse.

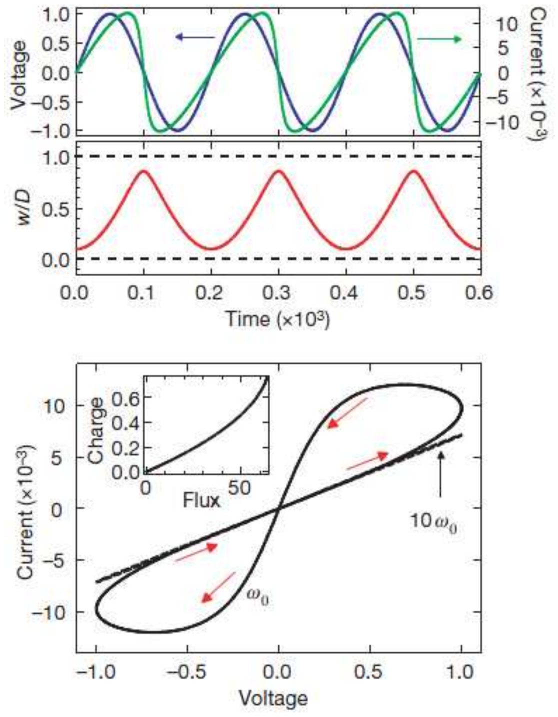


Figure 2.5: Memristor's hysteretic $I-V$ characteristic. (Source: [4]). (top) Nonlinear change in the current with respect to the applied voltage Vsin(ω_0t). (middle) The change in the internal state is measured as the ration of the doped width w to total device width D. (bottom) Frequency dependent hysterysis $I-V$ curve. The memristor is a frequency-dependent device, it can be seen that as the frequency changes from ω_0 to $10\omega_0$, it shows linear characteristics.

In this thesis we explore the memristor's nonlinear characteristics for different applications using a reservoir computing approach. For this purpose we use memristor *Simulation Program with Integrated Circuit Emphasis* (SPICE) model, which is described in Section 2.4.1.

2.4.1 Modeling Memristor using SPICE

To go beyond the theoretical concepts of the memristive systems, one requires a simulation model that can well approximate the physical characteristics of the memristor device. Rák *et al.* and Biolek *et al.* in [5, 33] have recently published

Simulation Program with Integrated Circuit Emphasis (SPICE) memristor models. Although both authors follow the published mathematical equations from [4], the memristor model defined in [5] takes into account the simulation stability allowing an average SPICE engine to handle multiple sub-circuit definitions [5]. Hence, in our thesis we use the Ngspice compatible memristor model defined by R*á*k and Cserey [5].

This SPICE memristor model has memristance range from 10 Ω to 1 KΩ, with the initial state being the conducting state, i.e., memristance of 10 Ω. To observe the model's current-voltage $I - V$ characteristics, we applied an input sinusoid of 300Hz with an amplitude of 0.5V, similar to that in [5]. Figure 2.6, shows the obtained simulated results. The (bottom) plot shows the non-linear current characteristics and (top) shows the change in the memristance. The device memristance increases non-linearly with the positive going input voltage and reaches its maximum (1 KΩ), i.e., non-conducting state. To switch back to the conducting state (minimum resistance), an input of the opposite polarity is required. During the switching event the memristor model stays in saturation, this is due to the effect of the high electrical fields in the thin-film of the memristor [5]. Figure 2.7 shows the hysteresis $I - V$ characteristics.

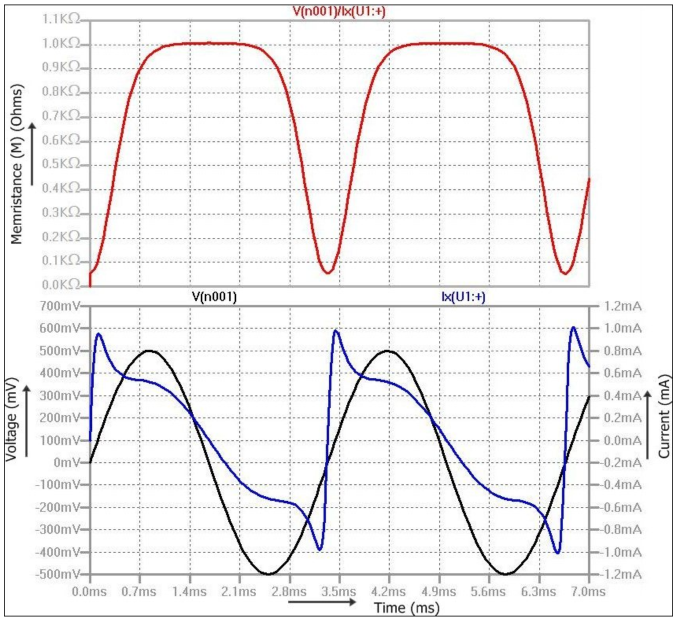


Figure 2.6: Simulation of the memristor SPICE model [5] with a sinusoid of 300Hz and 0.5V amplitude. (top) Change in the memristance with respect to the applied input. Memristance increases for a positive going input and decrease for a negative going input. (bottom) Nonlinear change in the current with respect to the applied sinusoid input.

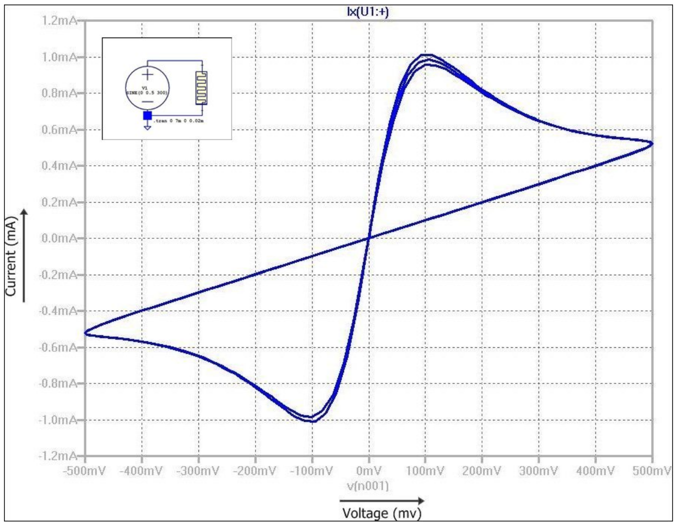


Figure 2.7: Memristor's hysteretic $I-V$ characteristic for applied sinusoid of 300Hz and 0.5V amplitude.

3

Simulation Framework

To implement the memristor-based reservoir computing architecture that will be presented in Chapter 4, we need a simulation framework that will allow us to generate memristor reservoirs and interpret their results. Since there is no commercial or open source software available for this purpose, we have developed a Matlab-based simulation framework called the *Reservoir Computing Simulation Program with Integrated Circuit Emphasis* (RCspice) for this purpose. The framework consists of $9,000$ lines of the code. Figure 3.1 shows the framework overview. The implementation details are described in the following Section 3.1.

The framework implementation is divided into the following three parts:

1. *Base Framework*: The base structure of the framework is implemented in Matlab [34].

2. *Simulation Engine (SE)*: Ngspice [17] is used as a simulation engine for the transient analysis of the reservoir.

3. *Evolutionary Engine (EE)*: Genetic Algorithms (GAs) are used to train the output layer of the reservoir. We use a Matlab-based GA toolbox [1].

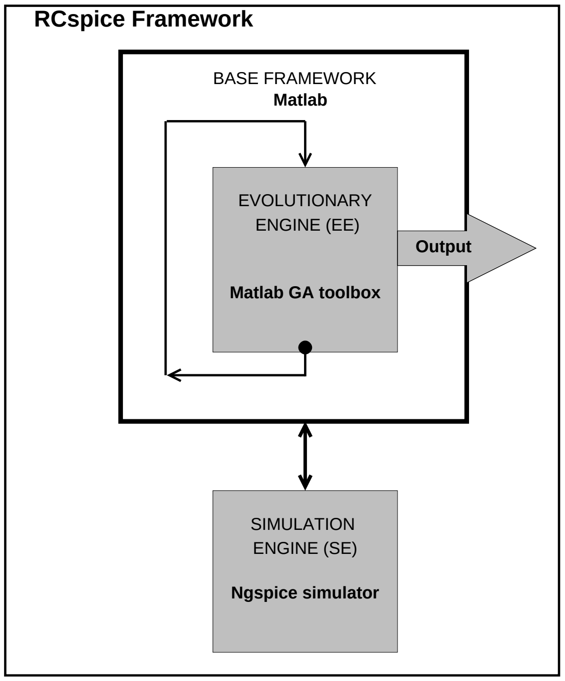


Figure 3.1: *RCspice* Framework overview.

3.1 Main Modules of the Framework

The core of our *RCspice framework* implementation is to define functions for an input layer, memristor reservoir and a readout layer and training algorithm (see Section 4 for details). *main_LSM* is the RCspice framework's main execution script. The details are as follows:

main_LSM: Responsible for setting directory and file paths, initializing the framework attributes and defining the user interface. Requires the user to input the required reservoir (network) size N.

- *setup_LSM*: Function adds path for the working, evolutionary engine and simulation engine directory.

- *funcInit_LSM*: Initializes framework attributes (see the main_LSM structure definition).

- *funcGenerate_LSM*: Creates a random network of N nodes. This function creates an adjacency matrix defining network connectivity and a component matrix defining the components in the network.

- *funcSetAttribute_LSM*: Set network attributes (see the main_LSM structure definition).

- *funcGenerateNetlist_LSM*: Generates a network netlist.

- *funcOutputLayer_LSM*: Defines the interface between the *simulation* and *evolutionary* engine.

funcGenerate_LSM Generates a reservoir (network) of memristors (see Section 4.2 for implementation details).

- *genIndividualGraph_LSM*: A high level function that initializes variables for the functions listed below.

- *funcNet_LSM*: Gets the pre-defined network configuration structure which defines the network components used and the template adjacency matrix size.

- *templateAdj_LSM*: Creates an empty N×N adjacency matrix, which will represent the network connectivity.

- *cell*: Cell is a Matlab construct [34]. This function takes network size N as an input argument and creates an empty N×N cell matrix. This matrix stores the component values for the network defined using the adjacency matrix.

- *addEdge_LSM*: Adds a random edge to the empty network adjacency matrix created using the function *templateAdj_LSM* and assigns a component name and a value to the edge. In our implementation, a memristor is assigned as a edge component and the value is the SPICE subcircuit definition.

- *isBiconnectGraph_LSM*: Checks if the network is bi-connected using Theorem 4.1.

funcGenerateNetlist_LSM: Defines subfunctions for the *simulation* engine (see Section 5.1 for implementation details)

- *selTaskFile*: Defines an experiment to be performed. Requires a user input to choose an experiment number from the given list of experiments.

- *createNGspiceFile_LSM*: This function takes the input arguments as an adjacency and component matrix to creates a Ngspice compatible network netlist.

- *perl*: This function is an interface to the Ngspice simulator (*simulation engine*). It invokes a perl script, which performs an transient analysis on the network netlist.

- *graphviz_LSM*: Creates a graph representing structural network information. This function takes the input argument as an adjacency matrix representing network connectivity and invokes *Graphviz*, a graph visualization software [35]. All the networks used in this thesis are presented in Section 6.

funcOutputLayer_LSM: Defines subfunctions for the interface between the *simulation* and the *evolutionary* engine.

- *load*: Loads the network transient data generated by the *simulation engine* (Ngspice simulator).
- *interpolateData*: Performs interpolation on the network transient data.
- *mainGA*: *Evolutionary engine's* main function.

mainGA: Defines subfunctions for the *evolutionary engine* (see Section 5.2 for implementation details).

- *InitGA*: Defines a GA structure which initializes the genetic algorithm parameters (an example GA structure is shown in the structure list for mainGA).
- *createParameterLog*: Creates a log file for the genetic algorithm parameters using the pre-defined GA structure.
- *createChrom*: Creates an initial (parent) population by using the *crtbase* function defined in [1].
- *decode_Chrom*: Decodes the initial population to real strings using the *bs2rv* function defined in [1].
- *objlsm1*: Calculates the actual output for a given population.
- *Fitness*: Evaluates the actual output against the target output for the entire population using Equation 5.2. This measures the raw performance (objective values) of the individuals in a population.
- *ranking*: Ranks individuals according to their objective values [1].

- *select*: Performs selection of individuals from a population to create a new (offspring) population [1].
- *recombin*: Recombination operator used to create population diversity.
- *mut*: This operator mutates each individual of the current population with a given probability defined in the GA structure [1].
- *reins*: This operator inserts offspring into the current population, replacing parents with offspring and returning the resulting population [1]. We use a fitness-based reinsertion.
- *createFitnessLog*: Creates a log of the best, worst and the average individual in the population for the number of generation count defined in the GA structure.
- *createPlotFile*: Creates a readout output data log for the best individual in the population for number of generation count defined in the GA structure.

Plot: Functions are used to plot the fitness and the readout layer output.

- *plotFitness*: Plots the fitness data logged by the *createFitnessLog* function.
- *plotOutput*: Plots the readout output data logged by the *createPlotFile* function.

Structures, variables defined for main_LSM: Structures are defined for the network and the readout layer configuration settings.

- *LSM_NETWORK*: This structure stores network_id, number of network components and number of network nodes.

- *LSM_READOUT_CONFIG*: This structure stores information for the readout layer, i.e., number of readout taps, connection type and the probability of connecting network nodes to the readout layer.
- *LSM_DATA*: This structure stores information for the reservoirs transient analysis. This includes, readout data points, target data points, time and length of data.
- *LSM_READOUT_CONFIG*: This structure stores information for the various activation functions for the readout layer.
- *READOUT_TAP*: A variable that defines number of output points in the readout layer.

4

Memristor-based Reservoir Computing Architecture

In our thesis we propose to implement memristor-based reservoir computing. This novel approach to design, train and analyze dynamical networks will help us to explore the dynamical property of the memristor. A block diagram of our approach is shown in Figure 4.1. The three main modules are; (I) input layer, (II) memristor reservoir and, (III) readout layer. The *genetic Algorithm* (GA) block represented by module (IV) is used to train the readout layer. The following subsections (4.1, 4.2.1, 4.3 and 4.4) describe the implementation details.

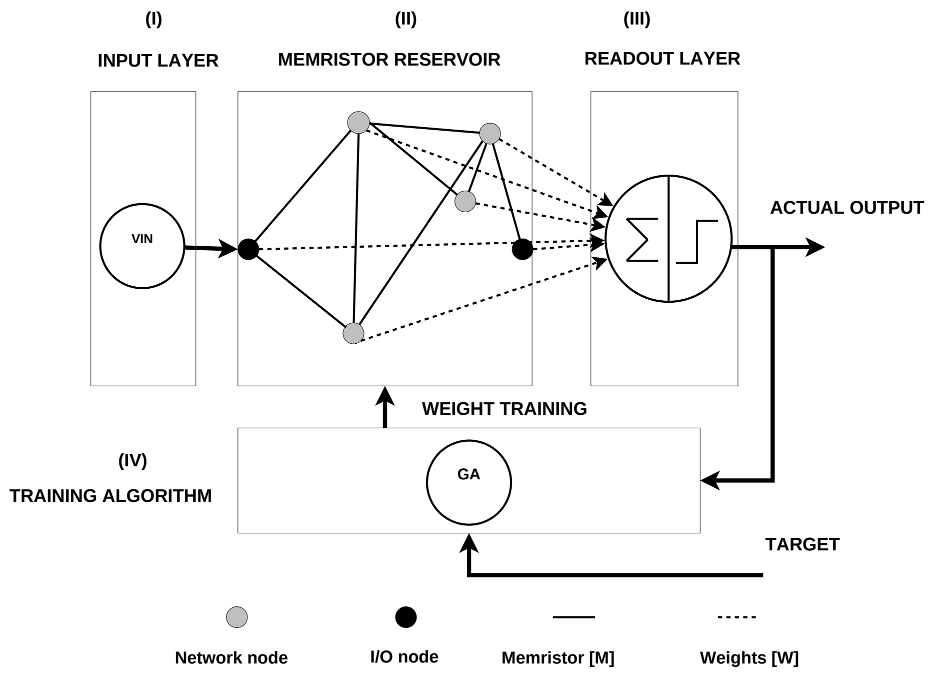


Figure 4.1: Architecture overview of memristor-based reservoir computing. (I) The input layer, (II) memristor reservoir and (III) readout layer. GAs are used for training the weights of the readout layer.

4.1 Input Layer

The input layer is used to distribute the inputs to the reservoir. The input definition depends on a particular task to be solved and the type of reservoir. In one of the examples mentioned in Section 2.3.3, where the reservoir is taken to be real water, the inputs to such a system are defined to be electrical motors that create a wave pattern on the surface of the water. In our implementation of memristor-based reservoirs, we define reservoir inputs as an independent voltage source which can generate *sine*, *pulsed*, *square*, and piecewise linear *pwl* waves or a combination thereof.

4.2 Memristor Reservoir

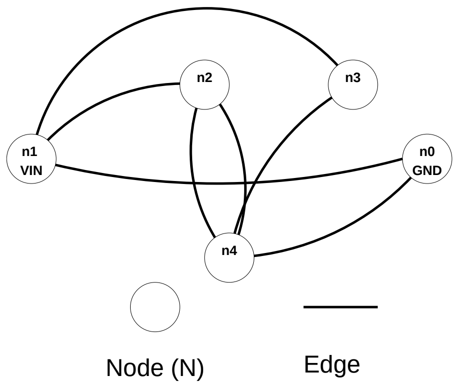
4.2.1 Graph Based Approach for Representing Memristor Reservoirs

In our framework, the *reservoir* is implemented as a network of memristors, which we will henceforth call *memristor-reservoir*. Memristor being a two terminal passive element, a network of memristors represent an *electrical circuit*. We define a memristor-reservoir as an *undirected bi-connected multigraph* [6]. Defining electrical circuits as graph allows us to discover structural properties like, connectivity and complexity of a circuit.

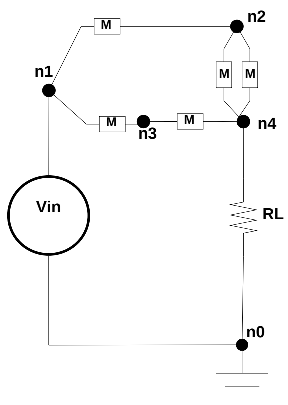
A graph $G(V, E)$ is defined as a set of vertices or nodes (V) and edges (E). A *multigraph* is a graph with multiple (i.e., parallel) edges. A *bi-connected* graph is sometimes referred as a 2-connected graph. A bi-connected graph is defined as a connected graph with no articulation point, i.e., the graph will remain connected even if a node is removed.

An example of an undirected bi-connected multigraph representing a memristor-reservoir is shown in Figure 4.2a. In this pictorial representation, the circle ◯ represents a graph node and an edge (__) is represented by the line between two nodes.

Figure 4.2a shows a graph representation of the corresponding memristor reservoir in Figure 4.2b. Here, graph nodes represent memristor terminals and edges represent memristor elements (M). For example, $n1$ is an input node (Vin) and $n0$ is a ground (Gnd) node. An edge between these two nodes represents a voltage source (Vin). Similarly, an edge between $n1$ and $n3$ represents a memristor element.



(a) 5-node graph.



(b) 5-node memristor network.

Figure 4.2: A 5-node undirected bi-connected graph and its equivalent memristor network.

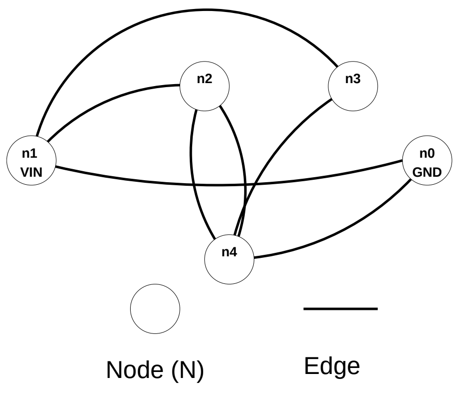
4.2.2 Adjacency Matrix Representation

A graph can be represented as a matrix. An incident or adjacency matrix is commonly used for graph representation [18]. In our implementation, we represent the memristor-reservoir by using an adjacency matrix.

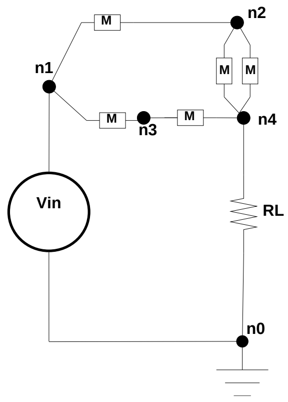
An adjacency matrix is a square matrix in which each row and column represents a graph node. In general, an N-node reservoir can be represented by a $N \times N$ adjacency matrix. Here, we consider the graph example shown in Section 4.2.2. Figure 4.3a represents a 5-node graph (i.e., $N\{0, 1, 2, 3 \text{ and } 4\}$). Its equivalent memristor network and adjacency matrix representation are shown in Figure 4.3b and Figure 4.4 respectively. In an adjacency matrix presence of an edge between two nodes is represented by 1. For example, in Figure 4.2a, an edge between $n1$ and $n2$ is represented by a 1 on the second row and first column (i.e., $Adj_{row,col}$ = $Adj_{2,1}$ = 1). A graph with parallel edges is represented by a number greater than one. In our representation, we do not allow self-loop, which means that the diagonal is 0.

4.2.3 Template Structure for Memristor Reservoir

After defining the adjacency matrix, we define a fixed template structure as shown in [6], which defines input and output reservoir nodes. Figure 4.5 represents the template structure with an input node Vin and an output node $Vout$, terminated with a 1KΩ load resistance RL. The equivalent adjacency matrix representing this template circuit is shown in Figure 4.6 and its equivalent memristor-reservoir template circuit representation is shown in Figure 4.5. This template adjacency matrix defination is used to generate the memristor-reservoir. Further details are described in Section 4.2.4.



(a) 5-node graph.



(b) 5-node memristor network.

Figure 4.3: A 5-node undirected bi-connected graph and its equivalent memristor network.

$$Adj = \begin{pmatrix} \mathbf{0} & 0 & 0 & 0 & 0 \\ 1 & \mathbf{0} & 0 & 0 & 0 \\ 0 & 1 & \mathbf{0} & 0 & 0 \\ 0 & 1 & 0 & \mathbf{0} & 0 \\ 1 & 0 & 2 & 1 & \mathbf{0} \end{pmatrix}$$

Figure 4.4: A 5×5 adjacency matrix representation of the 5-node graph shown in Figure 4.3a.

4.2.4 Generating a Memristor Reservoir

In this section we describe the generation of a random memristor network topology by using the template adjacency matrix (see Section 4.2.3). The reservoir representation has two key components as follows:

1. An adjacency matrix (Adj) representing both the memristor-reservoir connectivity and size.

2. A component matrix (E) based on the adjacency matrix.

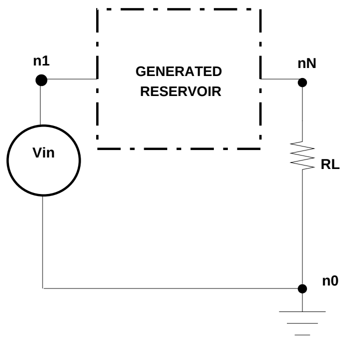


Figure 4.5: Template circuit for generating the memristor-reservoir. (Source: adapted from [6])

$$Template\ Adj = \begin{pmatrix} \mathbf{0} & 0 & 0 \\ Vin & \mathbf{0} & 0 \\ Rl & 0 & \mathbf{0} \end{pmatrix}$$

Figure 4.6: A 3×3 template adjacency matrix representation.

For example, Figure 4.7c and Figure 4.7d represents an adjacency matrix and its corresponding component matrix. Here, $Adj_{32} = 1$ corresponds to $E_{32} = M$. We represent component matrix using components Vin, Rl and M.

Figure 4.7 shows an example on how to generate a 5-node memristor reservoir. The steps are as follows:

1. We use a pre-defined template adjacency matrix as shown in Figure 4.7a (see Section 4.2.3 for details).

2. The user defines the number of nodes N (e.g., $N = 2$) to be inserted in the 3×3 template adjacency matrix. For N=2, an empty adjacency matrix of size (3+N)×(3+N) i.e., 5×5 is created using the 3×3 template adjacency matrix (see Figure 4.7a and 4.7b).

3. Using the 5×5 empty adjacency matrix, we add an edge between two randomly chosen nodes. For example, if random nodes 2 and 1 are chosen, the

skeleton adjacency matrix then becomes ($Adj_{21} = 1$).

4. After addition of an edge graph's bi-connectivity is checked by using *theorem 4.1* [6, 36]. Bi-connectivity theorem is applied to ensure a close loop network formation.

5. We add random edges until the adjacency matrix represents a bi-connected graph. Figure 4.7c shows the an adjacency matrix representation of a bi-connected graph, which defines reservoir connectivity.

6. Circuit representation is shown in Figure 4.7f, for $N = 2$, the memristor count MC is 5.

An equivalent component matrix is represented in Figure 4.7d. Figure 4.7f, shows an equivalent memristor reservoir defined using adjacency and component matrix. Formation of the memristor reservoir is shown in Figure 4.8. *Algorithm 1* summarizes the above steps.

Algorithm 1 Generating a random memristor network

```
1: Generate a template adjacency matrix $T$
2: Define number of network nodes $N$ to be inserted
3: Create an empty network adjacency matrix $A$
4: check for bi-connectivity using Theorem 4.1
5: while not bi-connected do
6:     randomly addEdge to matrix $A$
7: end while
```

Theorem: bi-connected multigraph [36]

A degree sequence of ‘p’ nodes $d_{1,2,..p}$ for graph G has a bi-connected realization, if and only if Π is graphical.

$$\begin{aligned} &\Pi = (d_1, d_2, ...d_p) with (d_1 \ \geq d_2 \ \geq ...d_p) \\ &d_p \ \geq 2 \\ &\sum_{i=1}^{p} d_i \ \geq 2(p - 2 + d_1) \end{aligned} \tag{4.1}$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

(a)

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

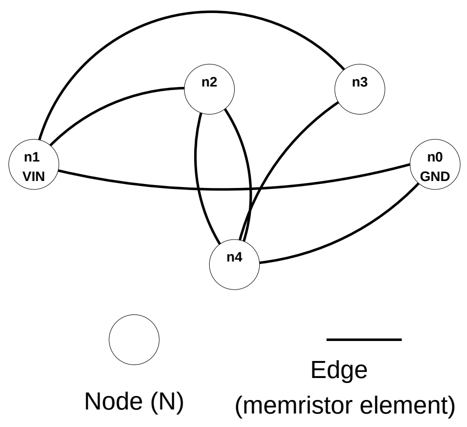
(b)

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 2 & 1 & 0 \end{pmatrix}$$

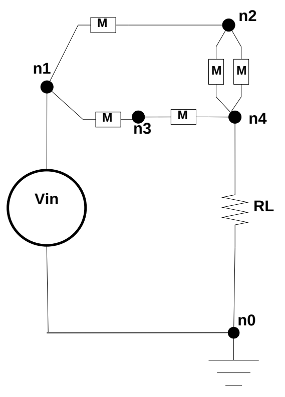
(c)

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ \mathrm{Vin} & 0 & 0 & 0 & 0 \\ 0 & \mathrm{M} & 0 & 0 & 0 \\ 0 & \mathrm{M} & 0 & 0 & 0 \\ \mathrm{Rl} & 0 & [\mathrm{M},\mathrm{M}] & \mathrm{M} & 0 \end{pmatrix}$$

(d)



(e)



(f)

Figure 4.7: Steps for generating a memristor reservoir by using an adjacency matrix. (a) A 3×3 template adjacency matrix T. (b) The template adjacency matrix expanded to add N user defined nodes. Here, $(N = 2)$ creating a 5×5 adjacency matrix. (c) Using procedure in step (3), random edges are added between the row and columns of the adjacency matrix representing memristor element. (d) An equivalent component matrix define components used. (e) Graph representation of the adjacency matrix in sub-figure *(e)*. (f) Fully connected memristor reservoir with component values.

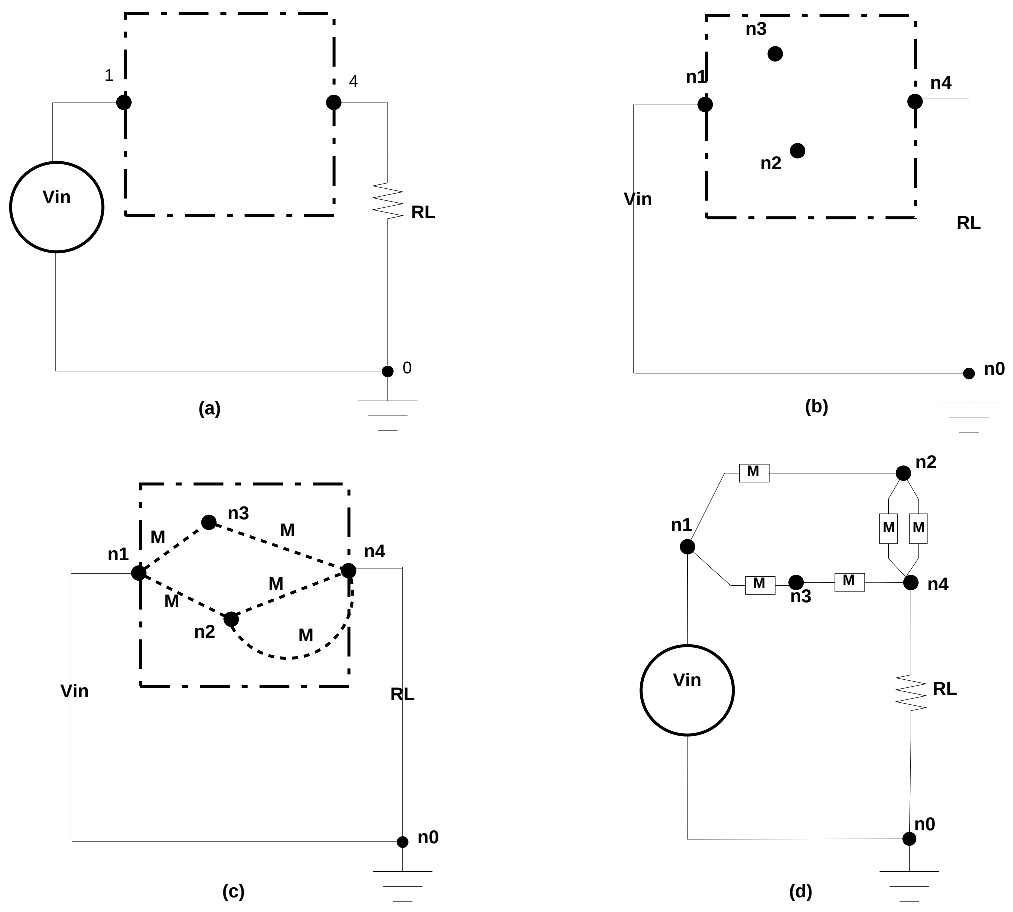


Figure 4.8: A diagramatic representation of 5-node memristor reservoir. (a) Template structure with input voltage source Vin and output load resistance RL. (b) Graph representation of the template structure. Nodes nodes $n3$ and $n2$ are inserted, N is 2. (c) Random edges are added between nodes. (d) Circuit representation of the generated random reservoir with edges representing memristor M. Figure 4.7c shows the adjacency matrix and Figure 4.7e shows graph representation for this example. (f) Circuit representation, for $N = 2$, number of memristor count MC is 5. (Source: adapted from [6])

4.3 Readout Layer

In Section 2.4 we have seen the non-linear and memory characteristic of the memristor. This implies that the internal states are stored as a change in the memristance value. This state change occurs with respect to the magnitude and polarity of the applied voltage as a function of time. Due to its internal state change, the voltage at a time $t + 1$ depends on the voltage at the previous time t.

The readout layer *y(t)* is implemented as a simple function f that maps the memristor reservoir states *x(t)* for nodes N, with weights W and bias B. Equation 4.2 defines the mathematical expression of the readout layer. Here, W_i, x_i and B represent weight, state and the bias for node i in an N-node reservoir. The linear function of the readout layer is used because of the non-temporal nature [20], which makes learning simple. To achieve an average state activity at a particular point in time, an activation function $f(.)$ is implemented. Generally, an activation function $f(.)$ can be *tanh*, *sigmoid*, *step* or *sign* function [20]. The final output obtained from the readout is $y_{response}$(t).

$$\begin{aligned} y(t) &= \sum_{i=1}^{N} f(x_i(t) \times W_i + B) = \sum_{i=1}^{N} f(V_i(t) \times W_i + B) \\ y_{response}(t) &= f(y(t)) \end{aligned} \tag{4.2}$$

4.4 Training Algorithm

Sections 4.1, 4.2 and, 4.3 describe the three main modules of the RCspice framework, *i.e.,* input layer, reservoir and the readout layer. As a next step, the readout layer needs to be trained to extract meaningful computation from the reservoir.

The core computational capability of the reservoir computing (RC) architecture lies in its high-dimensional state space formed due to its non-linear and dynamical elements. Hence, the readout layer maps reservoir states *x(t)* to y_{target}(t) as a linear combination of the reservoir states *x(t)* at time t as defined in Equation 4.2. The output layer is trained for the reservoir nodes, weight and the bias values that give optimum results. We use Genetic Algorithms (GAs) [1] for the weight, bias and node training. Implementation details for the GA are described in Section 5.2.

5

Evaluation Methodology

5.1 Reservoir Evaluation using Ngspice

After generating a N-node reservoir using the steps described in Section 4.2.4, we perform transient analysis using Ngspice. Transient analysis is performed by transforming the adjacency matrix of the reservoir into an NGspice netlist. The netlist defines a list of components (in our case memristors, voltage source and load resistance) and the nodes (or nets) that connect them together. Figure 5.1 shows an example on how to generate a netlist. The steps are as follows:

1. Create an adjacency matrix for the reservoir connectivity using *Algorithm 1*. An example adjacency matrix is shown in Figure 5.1a. Due to symmetry property of the adjacency matrix, the lower triangular matrix is used to define the netlist.

2. Create a component matrix defining the reservoir components as shown in Figure 5.1b.

3. The adjacency matrix, which describes the reservoir connectivity and the component matrix, which describes the reservoir components is passed onto a Matlab function in the base framework that creates a netlist compatible with Ngspice [17]. Figure 5.1c shows an example netlist.

4. Figure 5.1d shows the memristor circuit corresponding to the extracted netlist.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 2 & 1 & 0 \end{pmatrix}$$

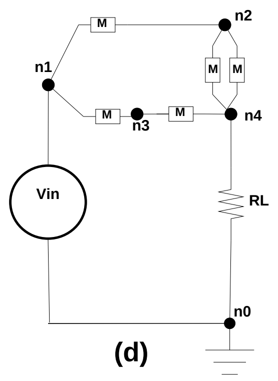
(a)

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ \mathrm{Vin} & 0 & 0 & 0 & 0 \\ 0 & \mathrm{M} & 0 & 0 & 0 \\ 0 & \mathrm{M} & 0 & 0 & 0 \\ \mathrm{Rl} & 0 & [\mathrm{M},\mathrm{M}] & \mathrm{M} & 0 \end{pmatrix}$$

(b)

```
*MAIN CIRCUIT
VIN 1 0 PULSE(1 1 2NS 2NS 50NS 100NS)
X0 1 2 memristor
X1 1 3 memristor
X2 2 4 memristor
X3 2 4 memristor
X4 4 3 memristor
RL 4 0 1KOHM
```

(c)



(d)

Figure 5.1: An example of the netlist formation. (a) Adjacency matrix representing the reservoir connectivity. (b) Component matrix defining the components used. (c) Ngspice netlist extracted from the component matrix. (d) Reservoir representation of the extracted netlist.

After creating the netlist, the next step is to simulate it using Ngspice [17]. The netlist is passed onto the simulation engine (see Figure 3.1) for transient analysis. Since the reservoir represents a dynamical electrical network, each node has time-varying voltage levels $V_1, V_2, ...V_N$, which are evaluated in the readout layer. Section 5.2, describes the readout layer training using the genetic algorithm.

5.2 Readout Training using Genetic Algorithms

The time varying node voltages extracted during transient analysis (see Section 5.1), represent memristor states as a function of network connectivity and time varying input signal. The readout layer *y(t)* is implemented as a simple function that maps these instantaneous memristor network states *x(t)* with corresponding weights and bias values using Equation 4.2. Reservoir nodes along with corresponding weights and bias values that contribute towards the final output are determined during training, which is done using Genetic Algorithm (GA). An example for evaluating the input response for the maximum generational count is shown in Figure 5.2.

Algorithm 2 outlines the pseudo-code for the GA implementation. The following subsections provide details about the implementation of the steps outlined in *Algorithm 2*.

Algorithm 2 Genetic algorithm for readout training

```
1: gen = 0
2: Initialize population
3: Evaluate initial population using objective function
4: while gen ≤ MAXGEN do
5:     Calculate fitness values for the entire population
6:     Select individuals for breeding
7:     Recombine individuals (crossover)
8:     Mutate individuals
9:     Evaluate offspring using objective function
10:    Reinsert offspring into population
11:    gen++
12: end while
```

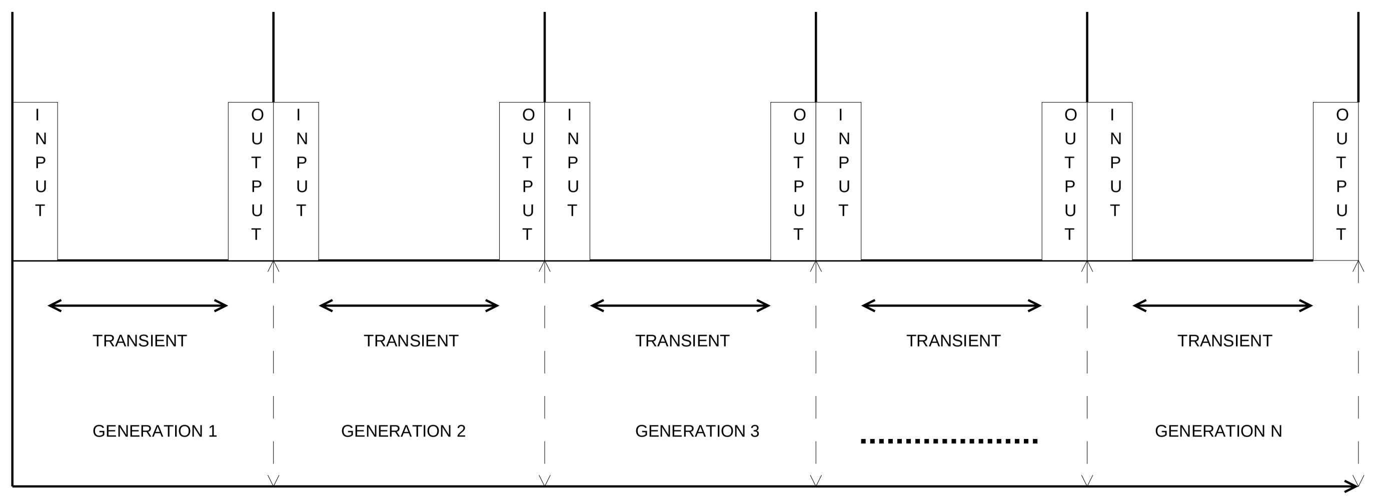


Figure 5.2: An input signal is evaluated at every generation. The transient time is task specific. The genetic algorithm evaluates the output response at the end of the transient analysis. The output response is evaluated with respect to the evolved weights, nodes and the bias values.

5.2.1 Population Representation and Initialization

Genetic algorithms (GAs) operates on a number of potential solutions. A set of potential solutions is termed a *population* and each potential solution in the population is represented as a *chromosome* [1]. We represent a potential solution (i.e., chromosome) as a set of three variables, i.e., reservoir node n represents the node at the voltage V_n, which corresponds to the memristor state x_n(t) for that node, the corresponding weight W_n at node n and a bias B. The bias value is *only* mapped for the entire reservoir and not for individual reservoir nodes.

The variables in the *chromosome* can be encoded solely as binary-strings or integers or floating point numbers or in any combination thereof. In our case, the reservoir node is encoded as an integer value while the weight and bias are encoded as 8-bit binary strings. The number of bits used to encode determine the precision level of the variables. For a N-node reservoir, $n_1, n_2, ...n_N$ represents individual reservoir nodes and $W_1, W_2, ...W_N$, are its corresponding weight. Thus, to represents variables as binary and integer strings, *createChrome* function defined in the base framework is invoked.

The mapping of an individual node and its corresponding weight onto a node-weight chromosome NW is shown in Table 5.1. An example to encode n_2 (i.e., node 2) from a N-node reservoir, its corresponding weight and bias value is shown in Table 5.1.

The population is defined as a matrix of size ($Nind \times Lind$) as shown in Figure 5.3. Here, the number of individuals $Nind$ in the population is represented by number of matrix rows and $Lind$ is the length of an individual, represented by number of matrix columns.

The node, weight and bias variables are encoded onto a chromosome as strings

Variables	**Node (n)**	**Weight (W)**	**Bias (B)**
Encoding	Integer	Binary	Binary
Encoding length	1	8-bit	8-bit
Chromosome example	2	10101111	11101011
Total length	**9**		**8**
	NW chromosome		B chromosome

Table 5.1: An example of encoding node, weight and bias variables into a chromosome.

of 0s and 1s. These variables need to be decoded to be passed onto the readout layer, which is described as a linear combination of these three variables (see Equation 4.2). This equation is evaluated to find the reservoir output $y(t)$. The next Section 5.2.2 describes decoding of these variables.

$$
Population = \begin{pmatrix} NW_{1,1} & NW_{1,2} & \dots & NW_{1,N} & B \\ NW_{2,1} & NW_{2,2} & \dots & NW_{2,N} & B \\ NW_{3,1} & NW_{3,2} & \dots & NW_{3,N} & B \\ \vdots & & & & \\ NW_{Nind,1} & NW_{Nind,2} & ... & NW_{Nind,N} & B \end{pmatrix} \begin{array}{c} IND_{1,Lind} \\ IND_{2,Lind} \\ ... \\ IND_{Nind,Lind} \end{array}
$$

Figure 5.3: Population matrix of size *Nind* $\times$ *Lind*. The number of rows corresponds to the number of individuals, the number of columns from (1 to $N-1$) represents the node-weight chromosome, and the N^{th} column is the bias chromosome.

5.2.2 Decoding the Chromosomes

The node, weight and bias variable are mapped onto a chromosome as strings of 0s and 1s. These variables take a random values from a predefined variable range defined in Table 5.2. To convert binary strings to decoded values is done using *decodeChrome* function defined in base framework. An example of encoded and decoded variables is shown in Table 5.3.

An example of the representing population encoded in strings of binary and integer values is shown in Figure 5.4. Here, population is defined for a 5-node reservoir and $Nind$ is taken to be 50. The length of each individual $Lind$ depends on the number of reservoir nodes, length of NW and bias chromosome. The total length of an individual $Lind$ for a 5-node reservoir is 53 (see Equation 5.1). The decoded population matrix representing variables as integers and real numbers is shown in Figure 5.5.

$$Lind = [(reservoirsizeN \times lengthofNWchromosome) + lengthofBchromosome] \tag{5.1}$$

Variables	**Node (n)**	**Weight (W)**	**Bias (B)**
Type	Integer	Real	Real
Range	[1 to No. of reservoir nodes (N)]	[0 to 5]	[-2.5 to 2.5]

Table 5.2: Decoded chromosome range and type definition.

Variables	**Node (n)**	**Weight (W)**	**Bias (B)**
Encoded chromosome	5	10100110	11010101
Decoded chromosome	5	3.843	1.0
Total length	**9**		**8**
	NW chromosome		B chromosome

Table 5.3: An example of decoded chromosome representing decoded values for variables node, weight and bias.

$$
Population = \begin{array}{c} \begin{pmatrix} \mathbf{0}01111000 & \mathbf{2}01100011 & \dots & \mathbf{5}01001101 & \mathbf{00101010} \\ \mathbf{3}11111011 & \mathbf{1}11001001 & \dots & \mathbf{4}00010101 & \mathbf{10101001} \\ \mathbf{4}10101101 & \mathbf{2}01010000 & \dots & \mathbf{2}01100000 & \mathbf{00111101} \\ \vdots & & & & \\ \mathbf{1}00111001 & \mathbf{5}11101101 & \dots & \mathbf{2}10110000 & \mathbf{11010101} \end{pmatrix} \end{array} \begin{array}{c} IND_{1,53} \\ IND_{2,53} \\ \dots \\ IND_{50,53} \end{array}
$$

Figure 5.4: An example of an encoded population matrix for a 5-node reservoir. First [1 to $N-1$] columns represents the NW chromosome. In each 9-bit NW chromosome, the 1^{st}-bit is node n and bit [2 to 8] represents weight W. The N^{th} column represents the 8-bit bias B.

$$
Population = \begin{pmatrix} [\mathbf{1}\ 1.5686] & [\mathbf{2}\ 1.2941] & \dots & [\mathbf{5}\ 2.3137] & [\mathbf{-1.500}] \\ [\mathbf{3}\ 3.3921] & [\mathbf{1}\ 2.7843] & \dots & [\mathbf{4}\ 4.9019] & [\mathbf{1.539}] \\ [\mathbf{4}\ 3.9411] & [\mathbf{2}\ 1.8823] & \dots & [\mathbf{2}\ 1.2549] & [\mathbf{-1.696}] \\ \vdots & & & & \\ [\mathbf{1}\ 2.4313] & [\mathbf{5}\ 3.5686] & \dots & [\mathbf{2}\ 4.3725] & [\mathbf{1.0}] \end{pmatrix} \begin{array}{c} IND_1 \\ IND_2 \\ \dots \\ IND_{50} \end{array}
$$

Figure 5.5: An example decoded population matrix for a 5-node reservoir. The first [1 to $N-1$] columns represents NW chromosome. From the 9-bit NW chromosome, the 1^{st}-bit decoded as an integer representing reservoir node n and binary bits from [2 to 8] represents the node weight W, which is decoded to be a real value. The N^{th} column represents 8-bit bias B decoded to be a real value.

5.2.3 Fitness Function

The objective or raw fitness function is used to provide a measure of how individuals IND in a given population perform for a given problem. An individual's performance is measured against a given target function. This is evaluated using the function defined in Equation 5.2, called the *squared error* function. Here, A_i is the actual value of the readout layer, T_i is the required target value, and $Nind$ is the population size. A_i is calculated by using the decoded values of weight (W), bias (B) and reservoir size (N) (see Section 5.2.2) and passing it to Equation 4.2.

The error value obtained using Equation 5.2 is the raw fitness of an individual.

The raw fitness values are calculated for the entire population and are passed on to the ranking function [1]. It uses a non-linear ranking scheme and ranks on the scale from 0 to 2. The fitness values are based on an individual's raw fitness value with respect to the the entire population. For the fitness assignment, we implement a minimization strategy, i.e., an individual with minimum error is assigned highest fitness (scale=2) and the one with maximum error is given the lowest fitness (scale=0). Table 5.4 shows an example of non-linear ranking scheme.

$$A(i) = \sum_{i=1}^{N} f(x_i(t) \times W_i + B) = \sum_{i=1}^{N} f(V_i(t) \times W_i + B)$$

$$Fitness\,(Error) = \sum_{j=1}^{Nind} (A_j - T_j)^2 \tag{5.2}$$

Raw fitness (Error)	Rank [0 to 2]
1 (Min Error)	2.00 (Highest rank)
2	1.66
3	1.38
4	1.15
5 (Max Error)	0.95 (Lowest rank)

Table 5.4: An example of non-linear ranking scheme used for evaluating raw fitness values.

5.2.4 Population Selection and Diversity

As a next step, after determining the fitness of each individual in the original population, the selection function [1] selects individuals for reproduction on the basis of their level of fitness. The selected individuals are typically fraction of the original population. In the GA, the basic operator for producing new individuals are mutation and crossover. We implement a multi-point crossover and the mutation rate is set to 0.05.

5.2.5 Population Reinsertion and Termination

In the selection phase of the genetic algorithm, since only a fraction of the original population is selected than the size of the original population the selected individuals (i.e., offsprings) have to be reinserted into the old population. We implement a fitness-based reinsertion scheme with a reinsertion rate of 80%. In this scheme the least fit individuals in the original population are replaced by the fraction of selected individuals. This selection and reinsertion procedure allows only the fittest individuals to propagate through the generational loop. An example of the reinsertion scheme is shown in Tabel 5.5 [1]. Reinsertion ensures population diversity.

This procedure continues for every generational loop until the termination condition is satisfied. In our implementation, we choose to terminate the GA when a specified generational count *MAXGEN* is reached (see Algorithm 2).

Original Population	Fitness 1
1	**21**
2	**22**
3	**23**
4	24
5	25
6	26
7	27
8	28

Selected Population	Fitness 2
11	**31**
12	**32**
13	**33**
14	**34**
15	**35**
16	36

New Population	New Fitness
1	21
2	22
3	23
15	35
14	34
13	33
12	32
11	31

Table 5.5: An example of fitness-based reinsertion scheme in which offspring replace the least fit parents with 80% of reinsertion rate. Here, the number of individuals for the original population is 8 and the selected population is 6. For the reinsertion rate of 80%, a total of 5 individuals from the selected population replace the least fit individuals in the original population. The new fitness values are copied according to the inserted selected population [1].

6

Memristor Reservoirs

This section presents the different memristor reservoirs used for our experiments. Each reservoir is created using the steps described in Section 4.2. Each reservoir can have a different *Memristor Count* (MC) even though the number of nodes is defined to be N. This is because during formation of the reservoir, we define the number of parallel edges allowed between two nodes (see Section 3.1). Table 6.1 summerizes the reservoirs used in our experiments and their associated memristor count.

6.1 6-Node Reservoirs

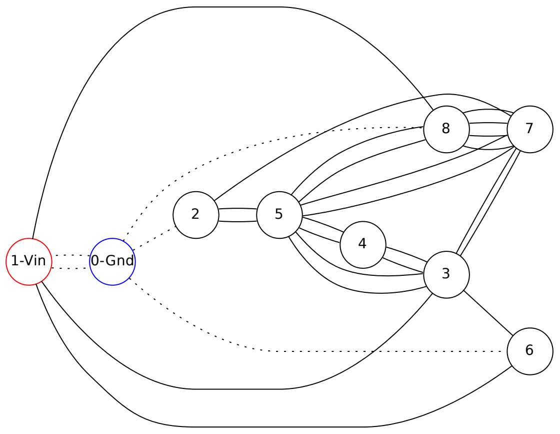


Figure 6.1: **6-node reservoir 1**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

Reservoir Size N	**Memristor Count** MC	**Figure Reference**
6	21	Fig. 6.1
	19	Fig. 6.2
	14	Fig. 6.3
	13	Fig. 6.4
	14	Fig. 6.5
10	30	Fig. 6.6
	25	Fig. 6.7
	40	Fig. 6.8
	25	Fig. 6.9
	33	Fig. 6.10
15	36	Fig. 6.11
	39	Fig. 6.12
	35	Fig. 6.13
	35	Fig. 6.14
	66	Fig. 6.15
30	76	Fig. 6.16
	110	Fig. 6.17
	73	Fig. 6.18
	81	Fig. 6.19
	101	Fig. 6.20
40	93	Fig. 6.21
	143	Fig. 6.22
	156	Fig. 6.23
	176	Fig. 6.24
	139	Fig. 6.25

Table 6.1: Reservoir size and the memristor count (MC).

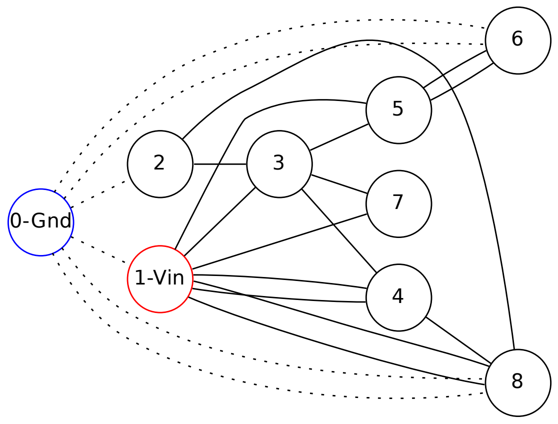


Figure 6.2: **6-node reservoir 2**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

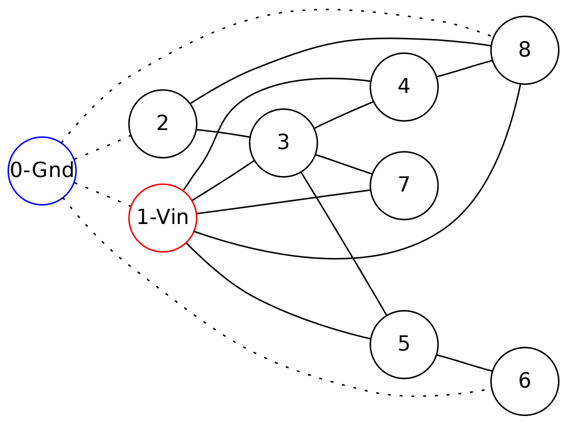


Figure 6.3: **6-node reservoir 3**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

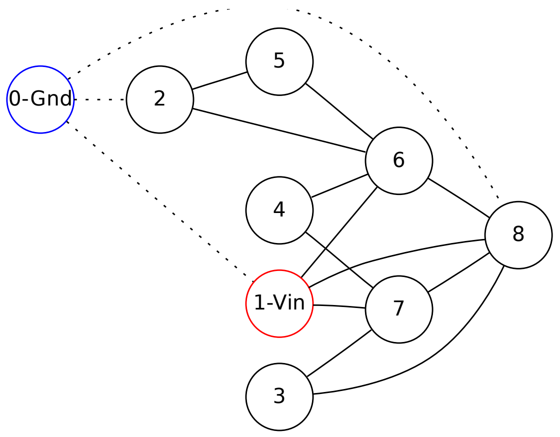


Figure 6.4: **6-node reservoir 4**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

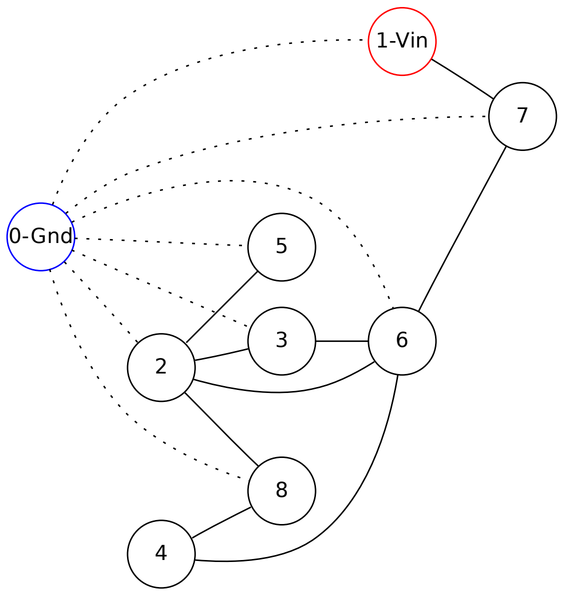


Figure 6.5: **6-node reservoir 5**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

6.2 10-Node Reservoirs

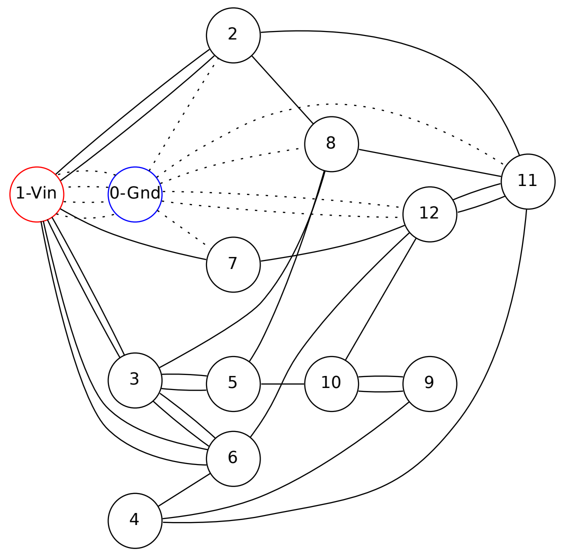


Figure 6.6: **10-node reservoir 1**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

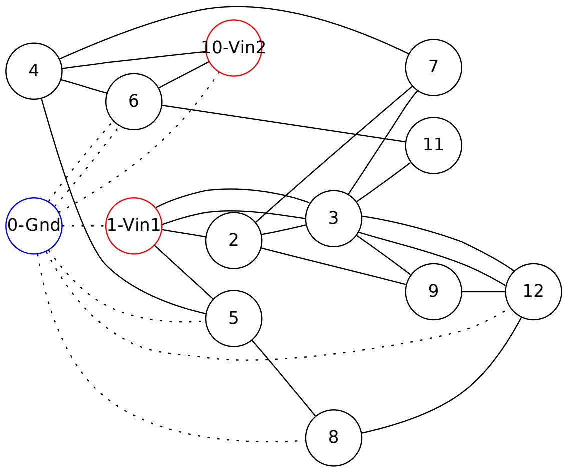


Figure 6.7: **10-node reservoir 2**. The red node represents two inputs (VIN1 and VIN2) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

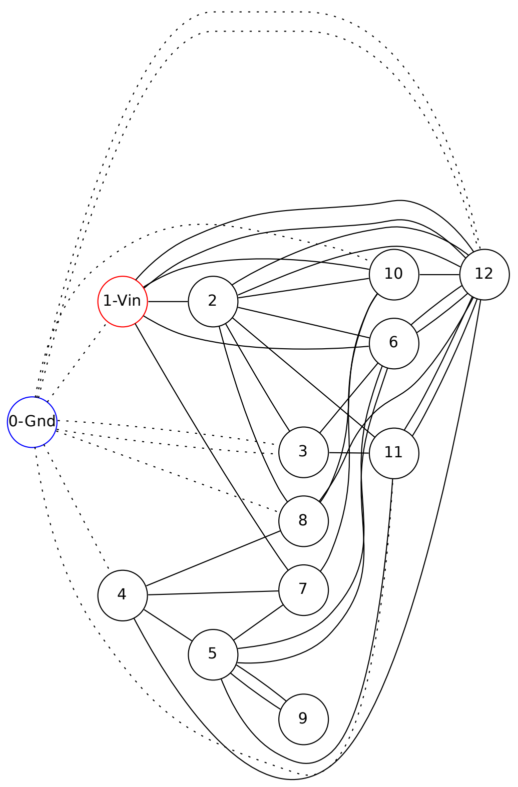


Figure 6.8: **10-node reservoir 3**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

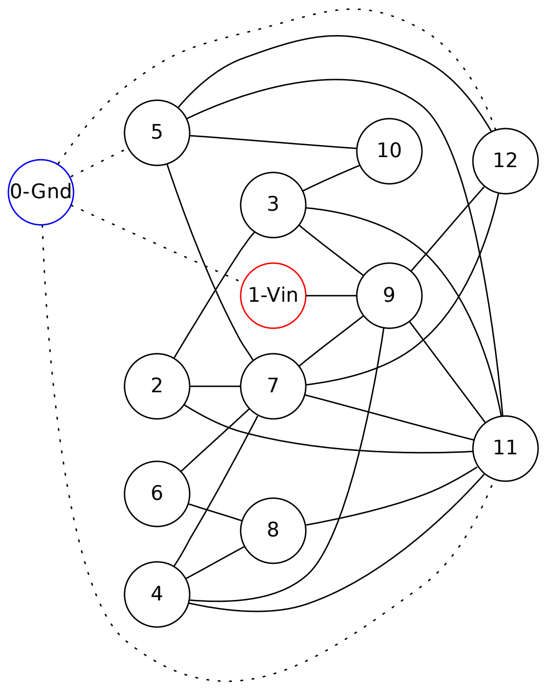


Figure 6.9: **10-node reservoir 4**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

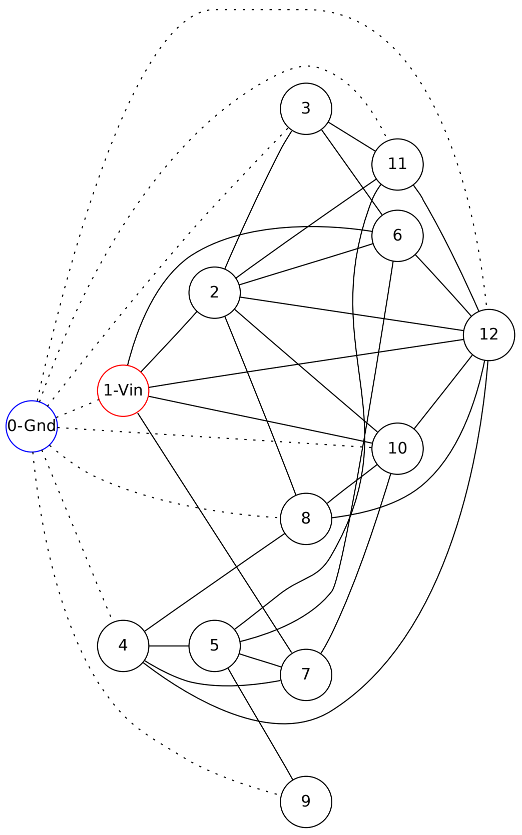


Figure 6.10: **10-node reservoir 5**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

6.3 15-Node Reservoirs

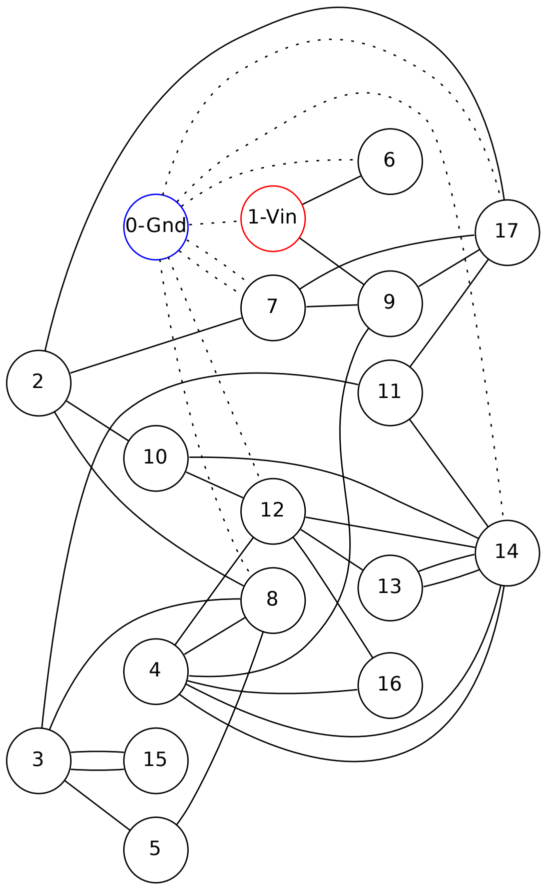


Figure 6.11: **15-node reservoir 1**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

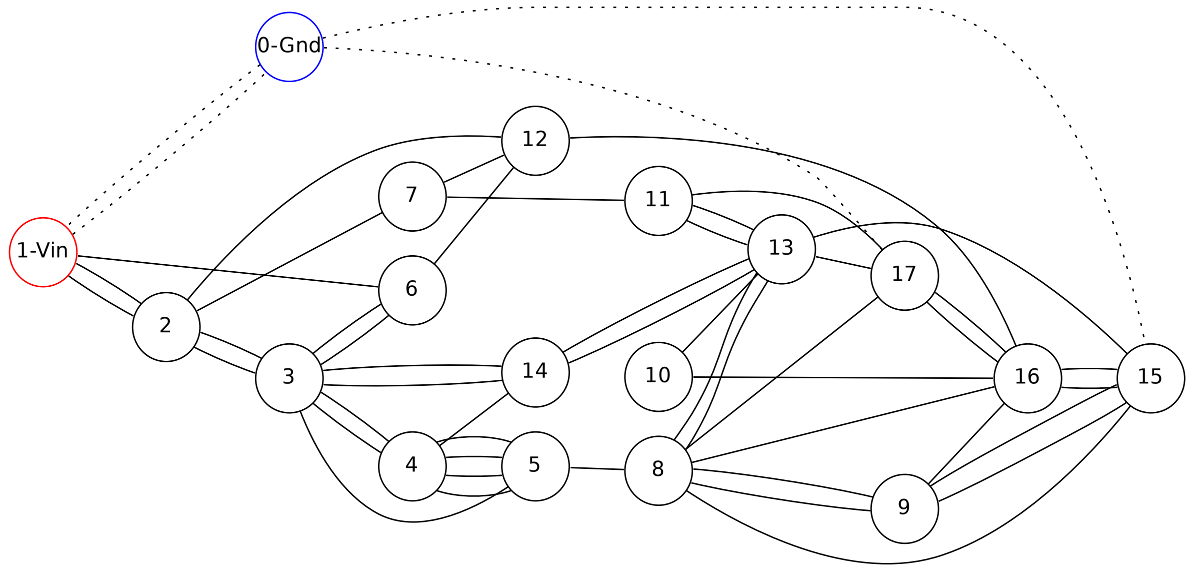


Figure 6.12: **15-node reservoir 2**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

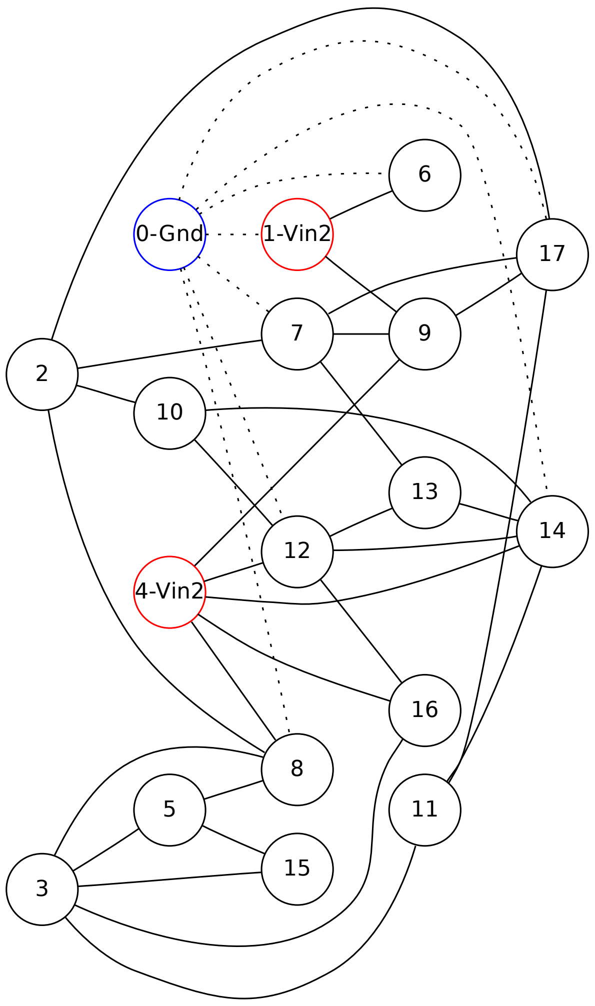


Figure 6.13: **15-node reservoir 3**. The red node represents two the input signals (VIN1 and VIN2) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

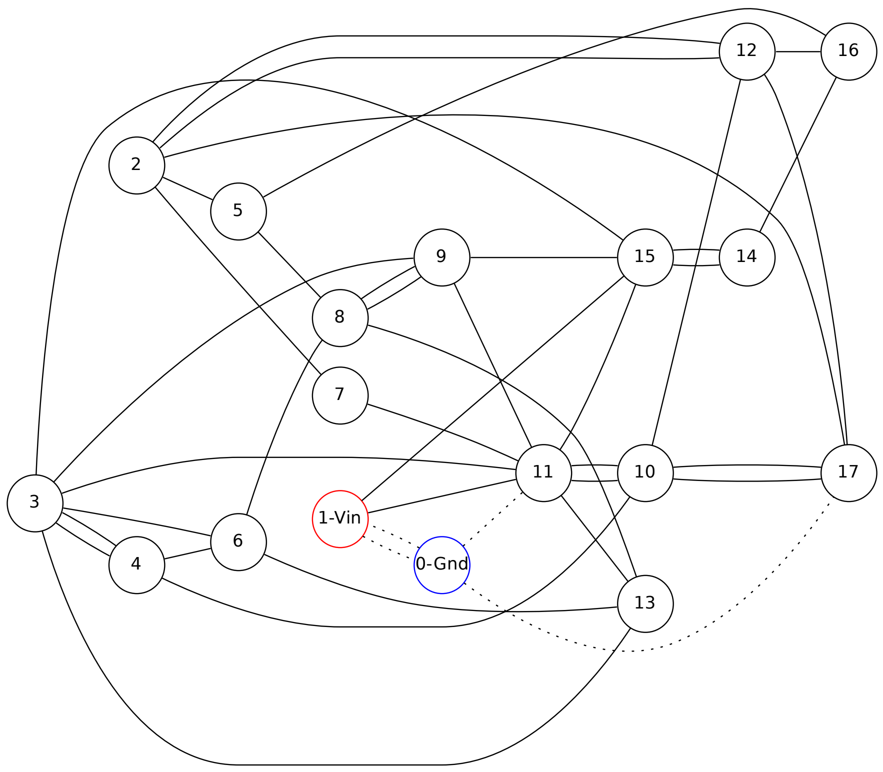


Figure 6.14: **15-node reservoir 4**. The red node represents the inputs (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

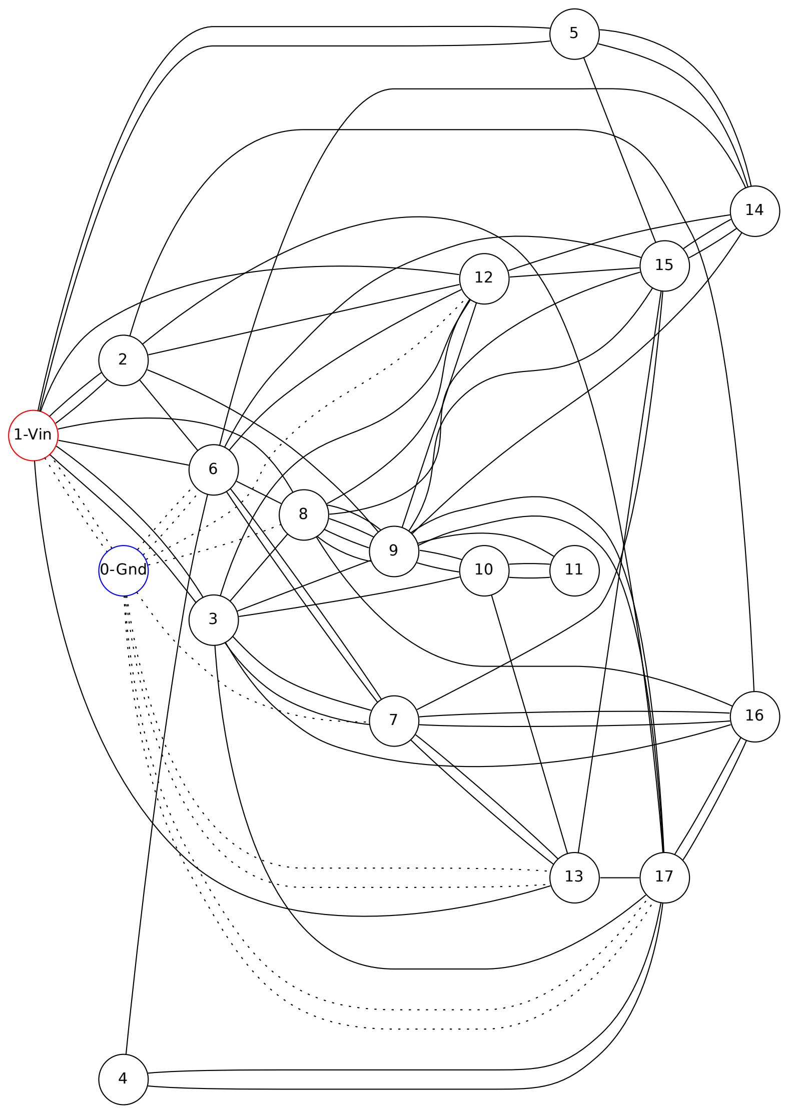


Figure 6.15: **15-node reservoir 5**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

6.4 30-Node Reservoirs

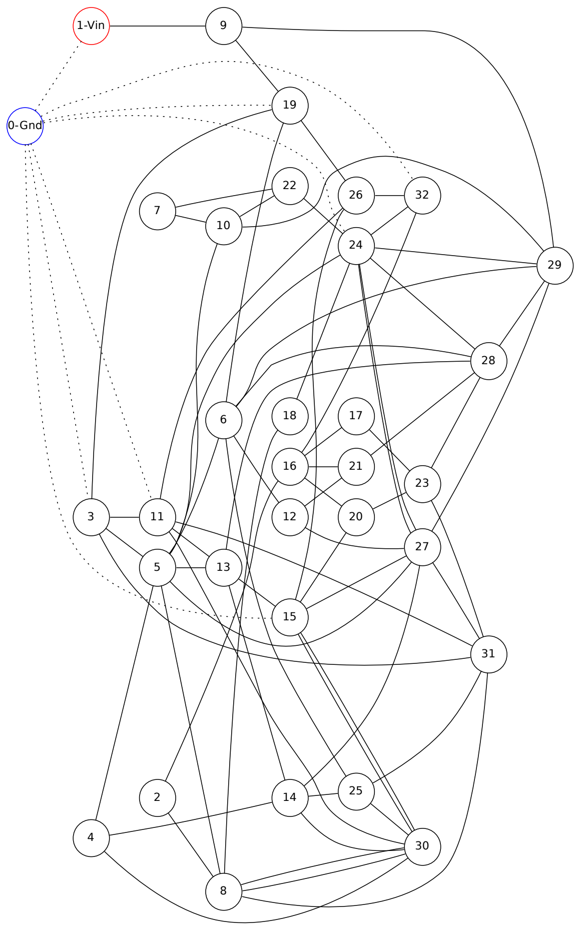


Figure 6.16: **30-node reservoir 1**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

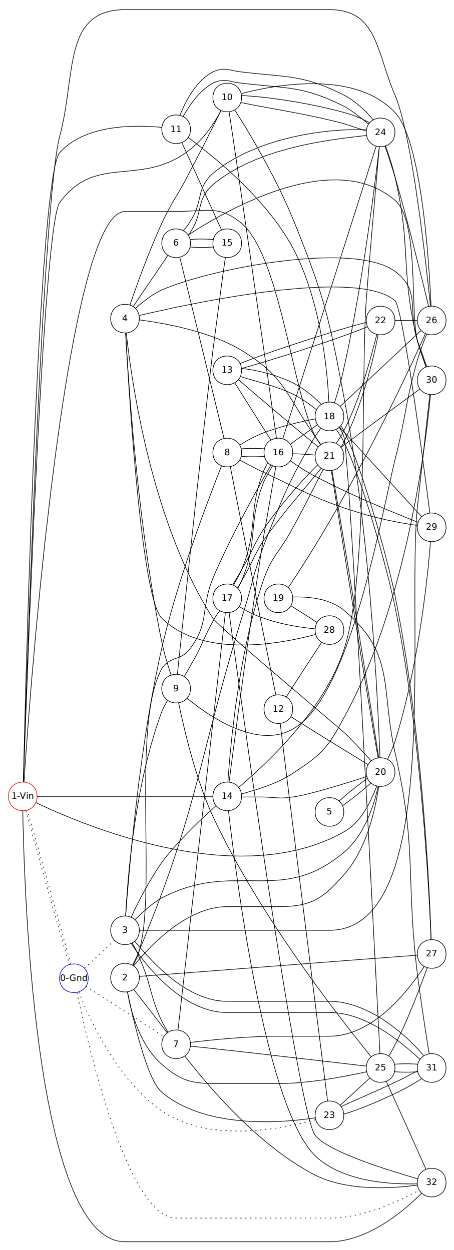


Figure 6.17: **30-node reservoir 2**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

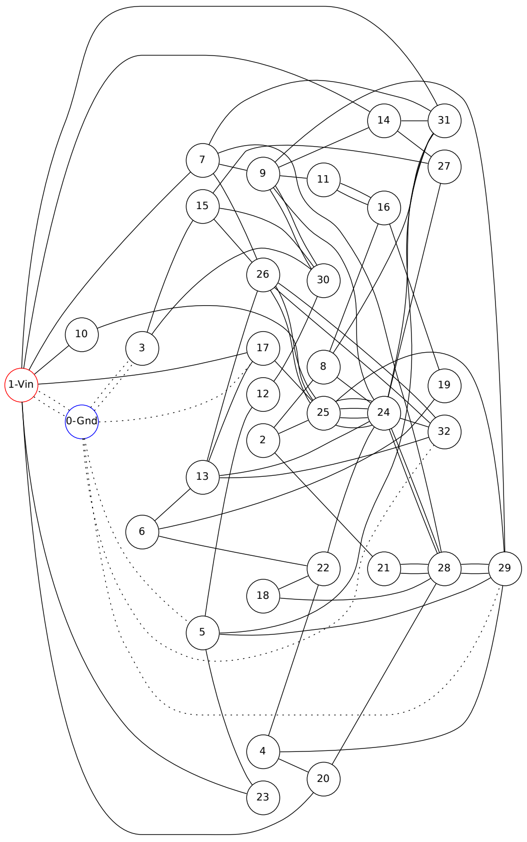


Figure 6.18: **30-node reservoir 3**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

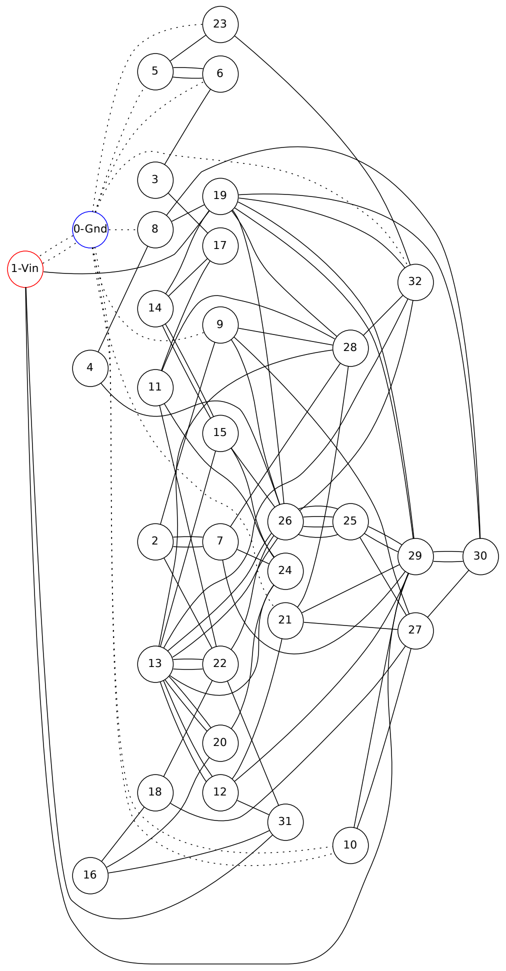


Figure 6.19: **30-node reservoir 4**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

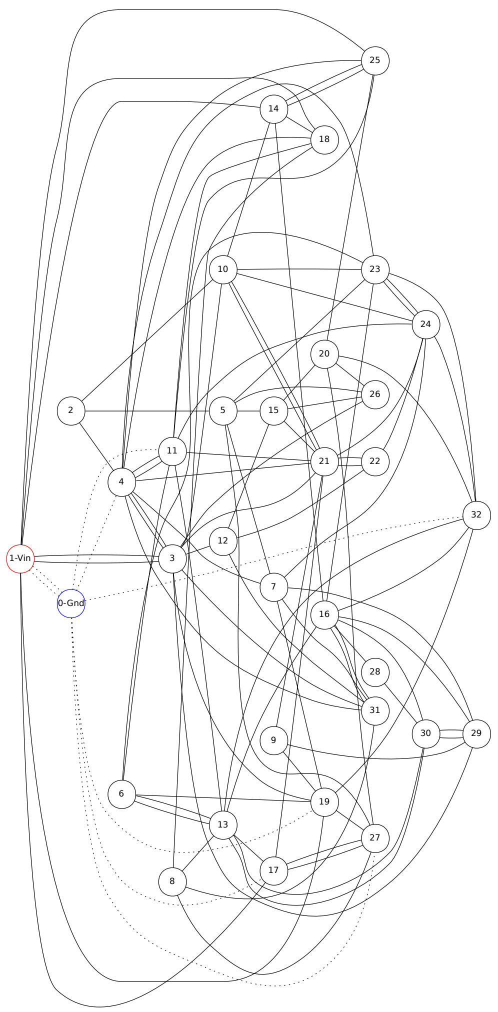


Figure 6.20: **30-node reservoir 5**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

6.5 40-Node Reservoirs

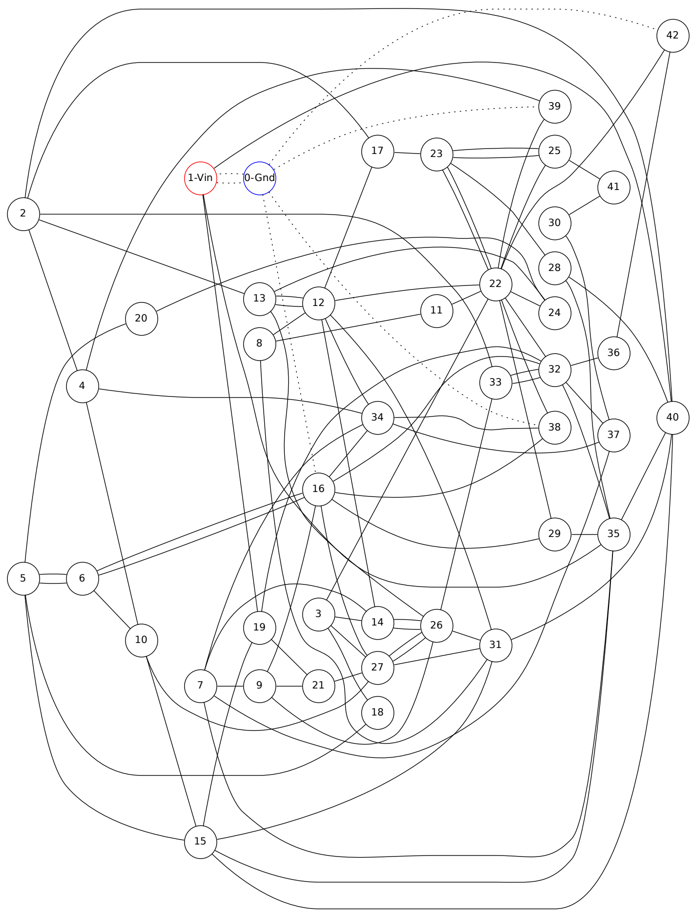


Figure 6.21: **40-node reservoir 1**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

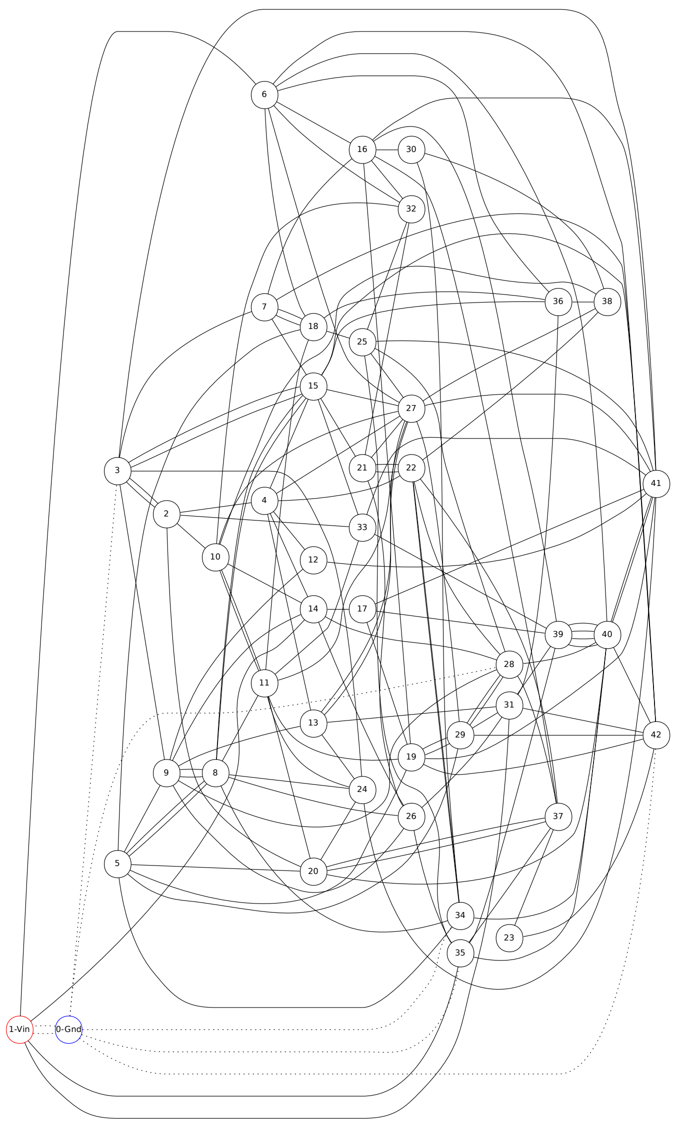


Figure 6.22: **40-node reservoir 2**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

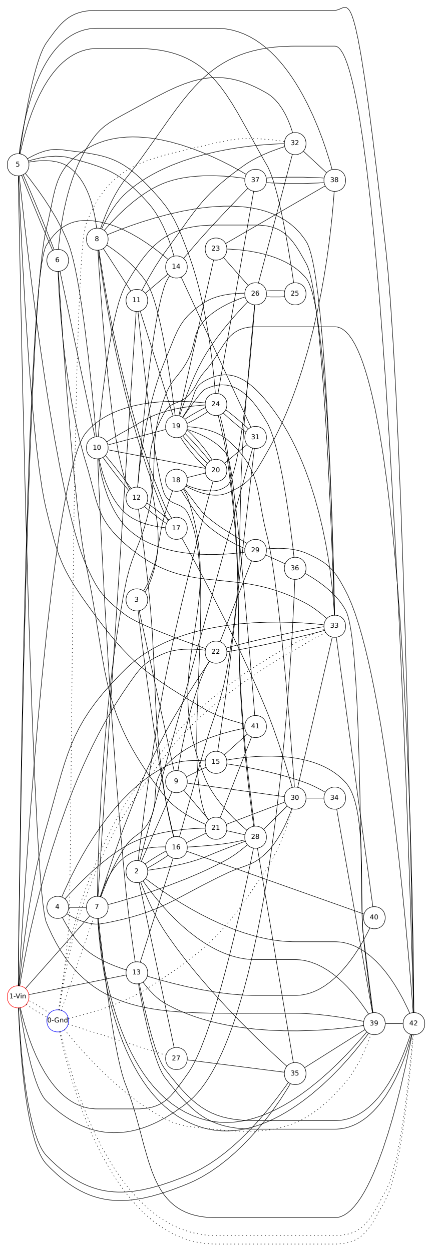


Figure 6.23: **40-node reservoir 3**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

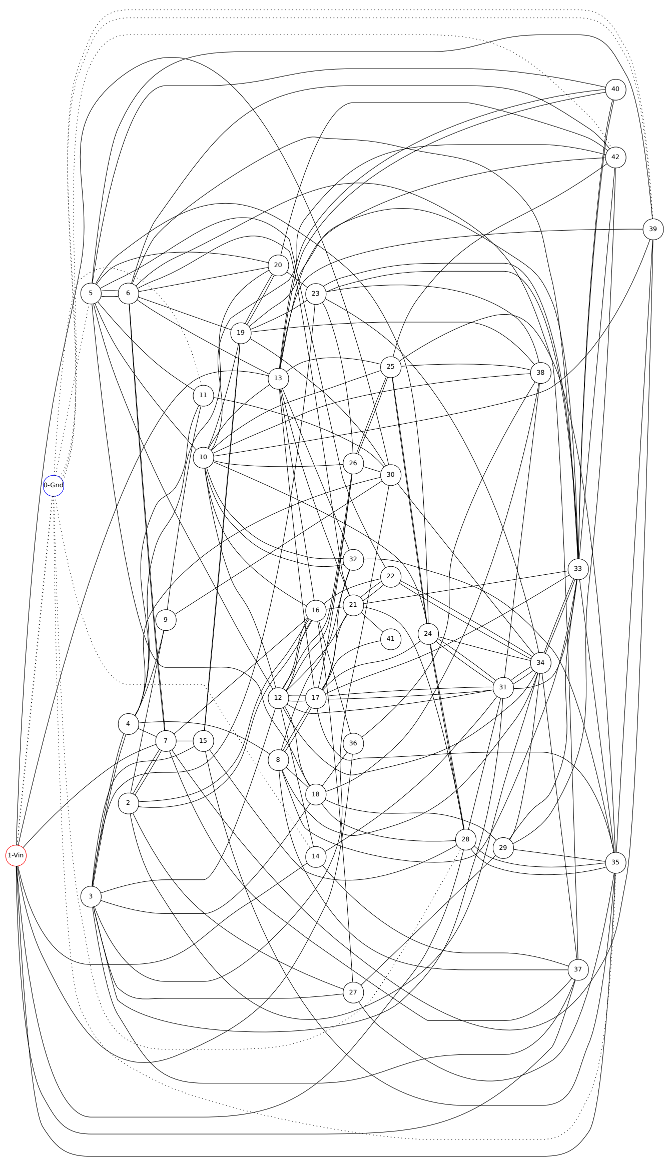


Figure 6.24: **40-node reservoir 4**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

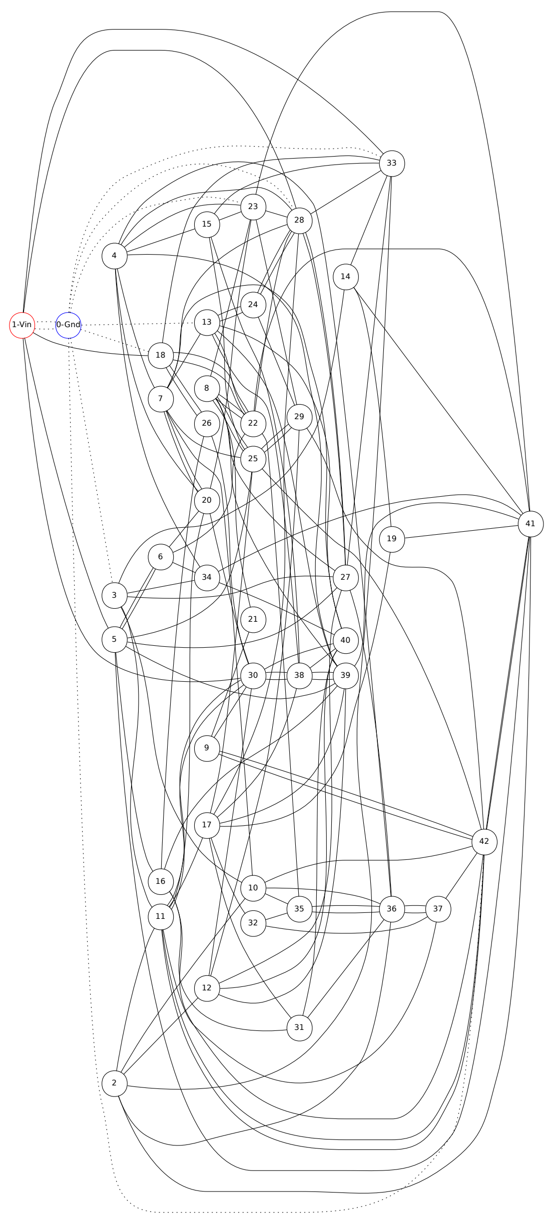


Figure 6.25: **40-node reservoir 5**. The red node represents the input node (VIN) and the blue node is the ground node (GND). The dotted and solid lines represent memristors.

7

Experiments and Results

This section presents an overview of the experiments performed.

In our first experimens (see Section 7.1), we focus on exploring the reservoir size as well as the genetic algorithm parameters. Reservoir size is chosen as one of the parameters because we would like to address the dynamic property of the memristors for different reservoir topologies. The second set of parameters belong to the genetic algorithm. These parameters play an important role to find the best solution in the large search space. The aim of our first experiment in Section 7.1 is to find the combined parameter set that gives optimal results.

Our first real experiment is a pattern recognition experiment for a triangular-square wave input pattern. Vandoorne *et al.* in [28] demonstrated the potential of photonic reservoir computing using a triangular-square benchmark task. We also performed an experiment on the variation in the input amplitude for different reservoir sizes for this task (see Section 7.3).

To explore the memristor's state change property (memristance), which depends on the change in the input amplitude and frequency, we simulate the reservoirs with realistic inputs, which include, amplitude and frequency-modulated signals. A pattern recognition experiment is performed for these signals (see Sections 7.4 and 7.5)

As a next experiment, we explore the memristor's unique memory property. Pershin *et al.* [7] experimentally demonstrated an associative memory behavior. We use the same experimental setup as a benchmark to demonstrate the associative memory behavior with reservoir computing (see Section 7.6).

Memristors not only find applications as a memory element, but they have also been capable of performing logical operations [37]. We demonstrate the basic logical operations, i.e., AND, OR and XOR (see Section 7.7).

7.1 Experiment 1: Reservoir and GA Parameter Exploration

In our first experiment, we explore parameters related to the memristor reservoir (i.e., reservoir size) N and the *genetic algorithm* (GA) (*i.e., population size, mutation rate and reinsertion rate*).

Vandoorne *et al.* in [28] demonstrate photonic reservoir computing using the non-trivial triangular-square signal as a pattern recognition benchmark task. We use the same triangular-square signal as a benchmark task for this experiment. The triangular-square pattern used in this experiment is shown in Figure 7.1 (top). The bottom shows the expected readout response, which should converge to logic (-1) for the triangular wave and logic (+1) for the square wave signal.

Fitness (i.e., error) value and simulation time (i.e., the number of generations required to converge) are defined as a measure for a reservoir's performance. We implement fitness minimization scheme in our RCspice framework, i.e., minimum the fitness value better the reservoir's performance (see Section 5.2.3). The error is calculated as the difference between the *readout* and the pre-defined *target* value using the squared error function described in Equation (5.2). We use 6, 10, 15, 30 and 40 node reservoirs in this experiment. The reservoir figure reference is listed in the Table 7.1.

The initial genetic algorithm setting involved the changing the reinsertion rate. It is a scalar that defines the rate of reinsertion of offspring per subpopulation in the range [0, 1]. The initial reinsertion rate of 0.5 yielded in higher fitness values compared to other values. We varied this parameter to be a 0.8, our simulation results showed convergence towards minimum fitness. For the next parameter, population sizes of 100 and 50. The convergence results for both the population sizes did not vary due to the fitness evaluation using mean squared error function. We found that number of generations required to converge for the population size of 100 for 40-node and 30-node reservoirs was much longer due to larger search space. We have therefore chosen a population size of 50.

Next, we change the mutation rate. Table 7.1 summarizes the fitness obtained for different reservoir sizes. The 3D plot in Figure 7.2 shows the average fitness evaluated over 5 simulations. Here, the *x-axis* shows the mutation rate, the *y-axis* denotes the reservoir size N and the *z-axis* shows the average fitness (error). Figure 7.3 shows the fitness averaged over 5 simulations as a function of number of generations (i.e., simulation time) obtained for all reservoirs sizes.

Discussion: From the set of reservoirs used in this experiment, we observed that the higher node reservoirs, i.e., 30-node and 40-node reservoirs showed the maximum average fitness values and the number of generations required for convergence is greater than $7,500$. The smaller reservoirs i.e., 6-node, 10-node and 15-node reservoirs converge in less than $3,000$ generations. The higher fitness values can be attributed to the genetic algorithm search space, which is a function of reservoir size, weight, and bias values as well as reservoir topology. For the GA parameter set, a population size of 50 was found to be optimal, as it allowed for faster computation time with respect to the fitness calculation across all reservoir sizes in comparison with the population size of 100. Figure 7.2 shows that the mutation rates different from 0.05 do not result in minimum fitness values. Thus, considering the reservoir and GA parameters that contribute towards the final output, we conclude that the reservoir sizes 6, 10 and 15 gave the optimum results, i.e., minimum fitness values and generational count with the GA parameter set consists of a population size of 50, a mutation rate of 0.05, a crossover rate of 0.07 and a reinsertion rate of 0.8.

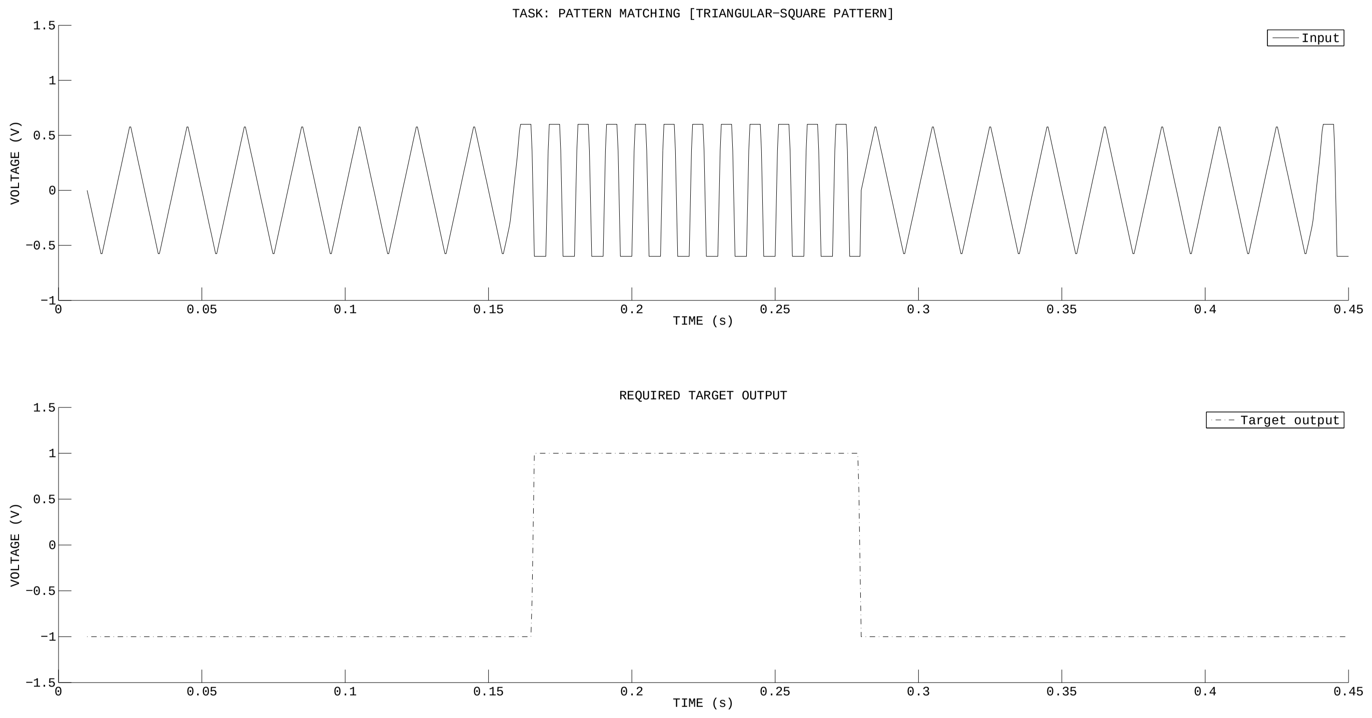


Figure 7.1: (Top) Input triangular-square wave signal; (Bottom) desired target response.

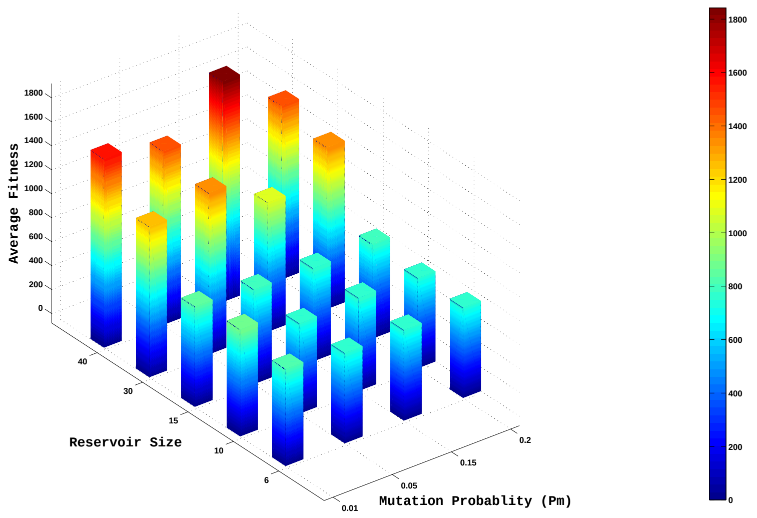


Figure 7.2: Fitness averaged over 5 simulations for the task shown in Figure 7.1. The *y-axis* shows the reservoir size N, the *x-axis* shows the mutation rate, and the *z-axis* shows the average fitness. Fitness plotted for population size of 50. The minimum fitness values are observed for the mutation rate of 0.05 across all reservoirs and the reservoirs of sizes 6, 10 and 15 showed minimum fitness.

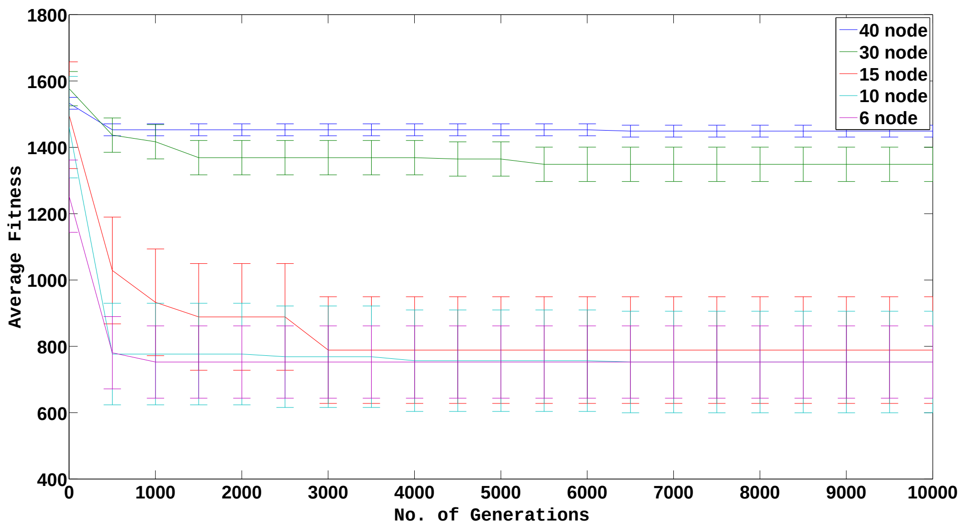


Figure 7.3: Best average fitness for population size 50, mutation rate 0.05 and reinsertion rate 0.8. Small size reservoirs with 6, 10 and 15 nodes, converge at a faster rate towards a minimum fitness value. The 30-node and 40-node reservoirs take more generation, i.e., simulation time to converge.

Reservoir Size N	Mutation Rate	Figure Reference	Average Fitness
6	0.01		808.80
	0.05	Fig. 6.1	772.79
	0.15		755.99
	0.2		752.80
10	0.01		892.00
	0.05	Fig. 6.6	847.20
	0.15		776.80
	0.2		752.80
15	0.01		836.80
	0.05	Fig. 6.12	989.80
	0.15		776.80
	0.2		788.80
30	0.01		1255.40
	0.05	Fig. 6.17	964.50
	0.15		1078.10
	0.2		1348.80
40	0.01		1575.36
	0.05	Fig. 6.21	1142.08
	0.15		1841.92
	0.2		1448.80

Table 7.1: Reservoir fitness averaged over 5 simulations for population size 50, reinsertion rate of 0.08.

7.2 Experiment 2: Pattern Recognition for Triangular-Square Signal

In Section 7.1 we explored the optimum parameters for the triangular-square input pattern [28]. Henceforth, we will use the optimal set of the genetic algorithm parameters (i.e., crossover rate = 0.07, mutation rate = 0.05 and a population size = 50) for all the experiments. Figure 7.1 (top) shows the input used for this experiment, which is a triangular-square signal with a voltage swing of ($\pm 0.6V$) and (bottom) shows the required target response. The expected readout response should converge to logic (-1) for a triangular wave and logic (+1) for a square wave. The input signal is defined using a *piecewise linear* (PWL) source defined in the Ngspice voltage source library [17].

We performed simulations using the RCspice framework with reservoir sizes of 6, 10, 15, 30 and 40. The aim of this experiment is to optimize the readout to minimize the error between the target and the readout response. Equation 5.2 is used to calculate the squared error (i.e., fitness). Table 7.2 lists the reservoirs used in this experiment, their corresponding fitness and memristor count.

Triangular-Square Patten Recognition Experiment				
Reservoir Size N	Memristor Count MC	Reservoir Figure Reference	Output Figure Reference	Fitness
40	93	Fig. 6.21	Fig. 7.5 (subplot A)	868.79
30	110	Fig. 6.17	Fig. 7.5 (subplot B)	760.79
15	39	Fig. 6.12	Fig. 7.5 (subplot C)	766.79
10	30	Fig. 6.6	Fig. 7.5 (subplot D)	766.80
6	21	Fig. 6.1	Fig. 7.5 (subplot E)	766.79

Table 7.2: Simulation results obtained for the triangular-square signal with amplitude of ($\pm 0.6V$).

Figure 7.5 shows the response for all the reservoirs used in this experiment. A representation of the readout for a 15-node reservoir used in this experiment

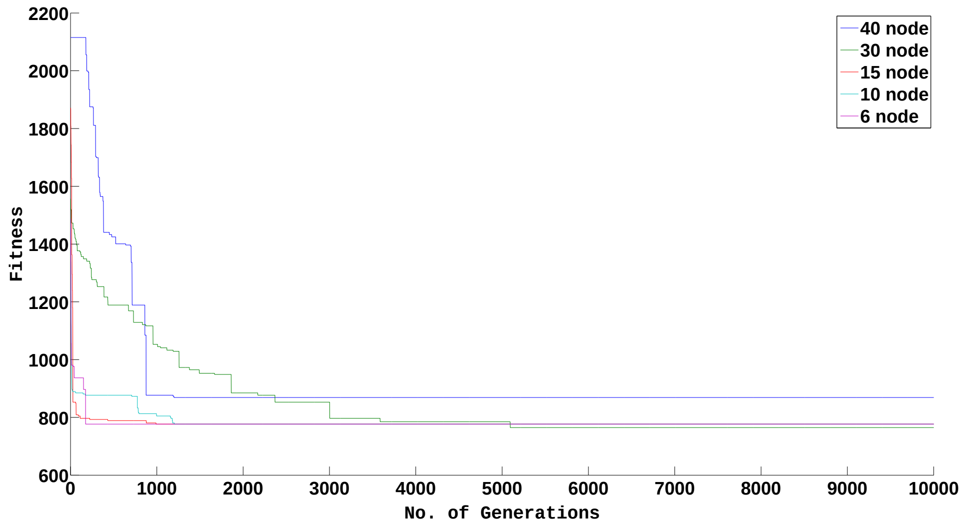


Figure 7.4: Fitness as a function of reservoir size. Higher node reservoirs require more generations to converge towards minimum fitness.

is shown in Figure 7.6. Here, the readout function Out maps the reservoir states x(t) for the nodes, with weights and bias values. The final response Y is obtained using a *sign* activation function, whose response is (-1) for a negative signal and $(+1)$ for a positive signal.

Figure 7.4 plots the fitness as a function of the number of generations required to converge for 6, 10, 15, 30 and 40 node reservoirs. It shows that the 40 and 30 node reservoirs requires more than $10,001$ generations to converge towards the minimum fitness.

Discussion: As seen from Figure 7.4 as the reservoir size increases, so does the number of generations required to converge. This is because the higher node reservoirs, i.e., 30-node and 40-node present a large search space for genetic algorithms to find the optimum weights, nodes and bias combination for minimizing the fitness.

The average power consumption for a memristor-reservoir is a function of size and therefore the number of memristors MC (i.e., cost). A memristor is a passive device, if its memristance does not change for a time-varying input, the memristor simply acts as a constant resistive element, which will consume power. The memristor count associated with the 6-node, 10-node and 15-node reservoirs is lower and hence, it is more likely that under varying input signals, most of the memristors will change their internal state (memristance) and hence, the average power consumed will be lower. A comparison of the average power consumed as a function of the topology remain to be investigated and are beyond the scope of this thesis.

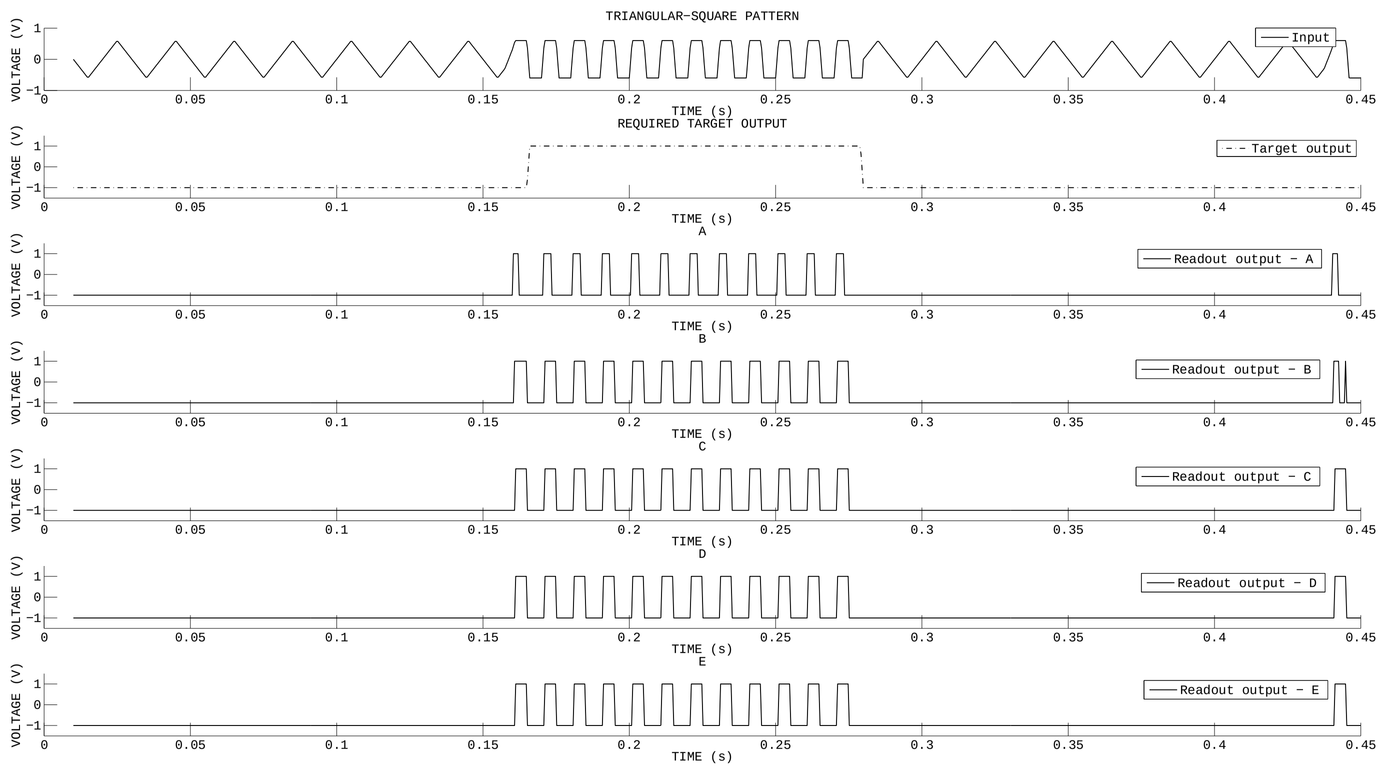


Figure 7.5: Simulation response for the reservoirs listed in Table 7.2. Describing the plot from (top to bottom). (*first*) Input to the reservoir; (*second*) the desired output response. Next, the readout responses for 40-node, 30-node, 15-node, 10-node and 6-node reservoirs are shown from A to E respectively. For the triangular input, the readout response for all the reservoirs converge to logic -1 and for the square input the response is seen as pulses of logic 1. For the 40-node readout response as shown in A, it can be observed that the response for the square input has a smaller pulse width as compared to rest of the reservoir responses. This is reflected as a higher mean squared error value of 868.79. Clearly, all the reservoirs are able to distinguish between the square and the triangular input.

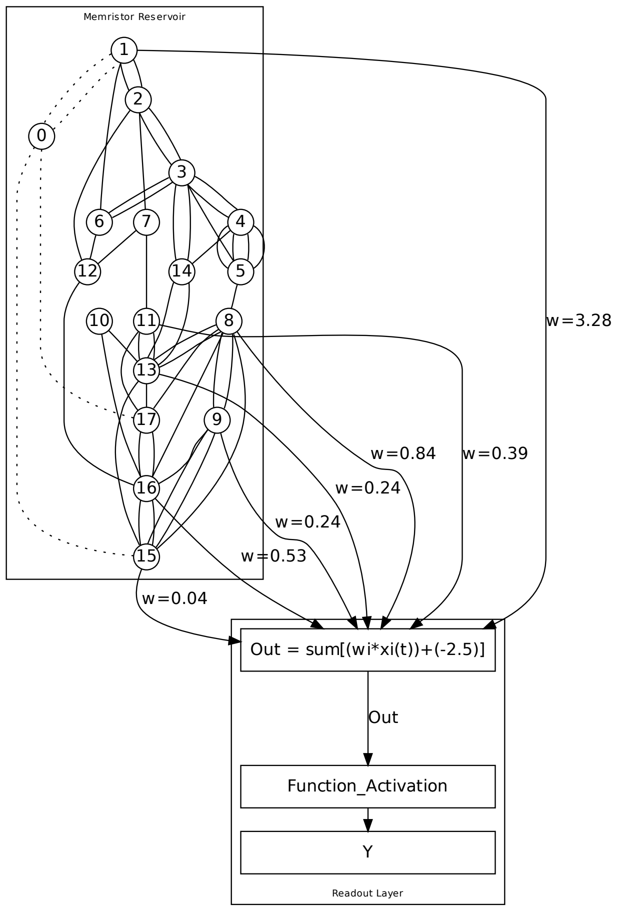


Figure 7.6: A representation of the readout *Out*, which maps the 15-node reservoir states x(t) for nodes $1, 9, 11, 13, 16$ and 15 with weights w represented for each node. A single bias of (-2.5) is used towards the final output Y. *sign* activation is used in this example. The raw readout response *Out* and the final readout response Y is shown in Figure 7.7.

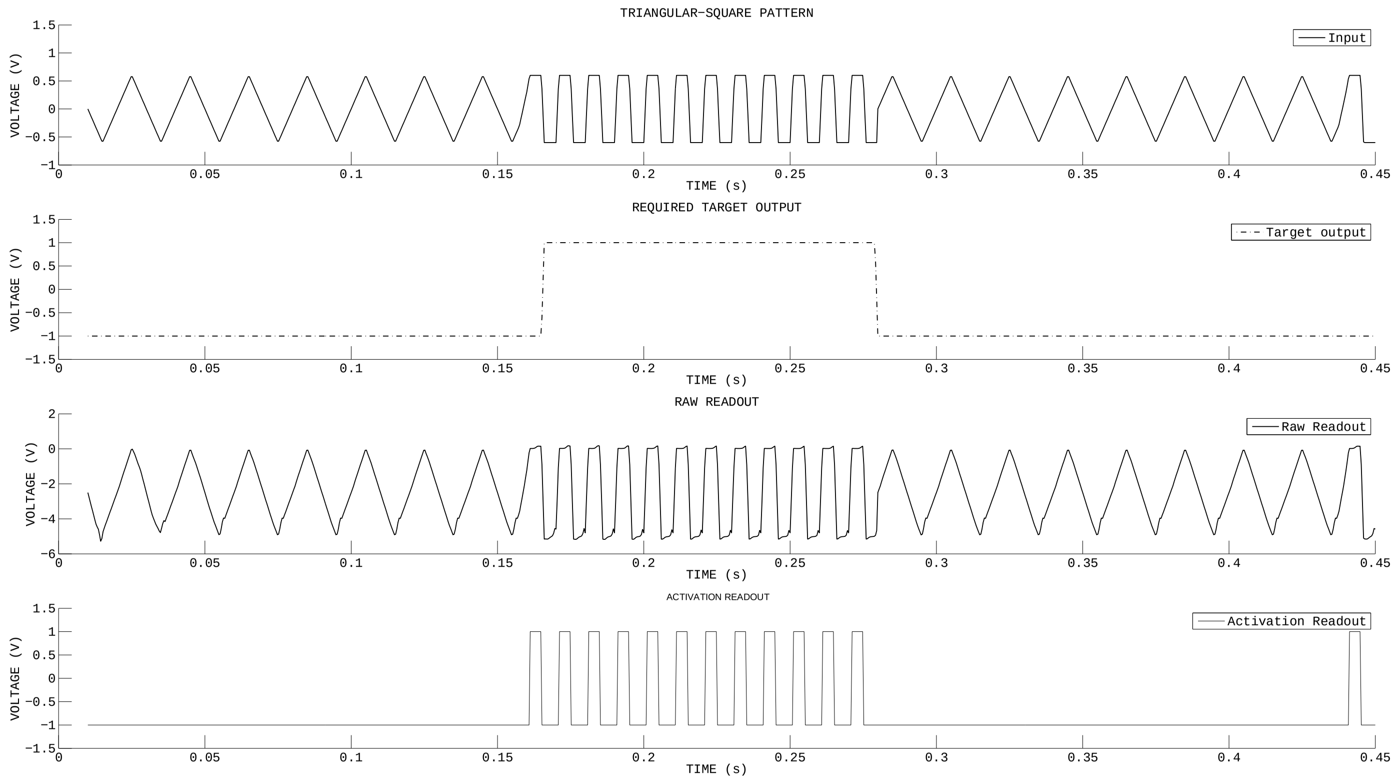


Figure 7.7: Readout *Out* response for the 15 node reservoir as shown in Figure 7.6. Describing the plot from (top to bottom). (*first*) Input to the reservoir; (*second*) The desired output response; (*third*) The raw readout response (before the activation function). We observed that for the triangular input the reservoir response converged towards a negative voltage and for the square signal the reservoir response shows positive spikes; (*fourth*) The final readout response after the *sign* activation function. The raw readout response is transformed by the *sign* function, whose response is -1 for a negative signal and $+1$ for a positive signal. Thus, for the triangular wave we observe a -1 logic and $+1$ logic spikes for the square input.
]

7.3 Experiment 2a: Input Variation for Triangular-Square Pattern

This section describes the experimental setup and the results obtained by introducing amplitude variations for the triangular-square input pattern recognition benchmark task described in Section 7.1. The input amplitude for the square-triangular pattern was varied in range of $0.2V$, $0.4V$, $0.5V$, $0.6V$, $0.7V$, $0.8V$ and $1V$ for the reservoirs used in Section 7.2. Table 7.3 shows the fitness comparison for the input amplitude variation for different reservoir sizes. For this experiment we trained the readout layer for all the reservoirs from Section 7.2 for an initial input of $\pm 0.6V$.

Amplitude (V)	**Fitness for Reservoir Size N**				
	40	30	15	10	6
0.2	1136.8	1136.8	1136.8	1136.8	1136.8
0.4	1020.8	936.8	1136.8	1136.8	1136.8
0.5	1064.8	1136.8	1136.8	1136.8	1136.8
0.6	**868.79**	**760.79**	**766.79**	**766.80**	**766.79**
0.7	1016.8	850.871	1028.8	1024.8	1024.8
0.8	1204.8	1212.80	1204.80	1144.8	1144.8
1	1268.8	1439.2	1419.2	1144.8	1311.2

Table 7.3: Simulation results obtained for the triangular-square input pattern as a function of amplitude variation and reservoir sizes.

Discussion: The fitness values for the initial setting of $\pm 0.6V$ is observed as the dip point in Figure 7.8. The lower signal amplitude range, i.e., $0.2V$, $0.4V$, and $0.5V$ do not cause significant change in the memristor's (i.e., reservoir's) internal state due to the memristor property (see Section 2.4.1). Hence, the readout response, which is a function of the internal reservoir states, do not show significant variation. This is observed across all the reservoirs. By varying the signal in a higher amplitude range, i.e., $0.7V$, $0.8V$, and $1V$ the internal reservoir states show a higher state

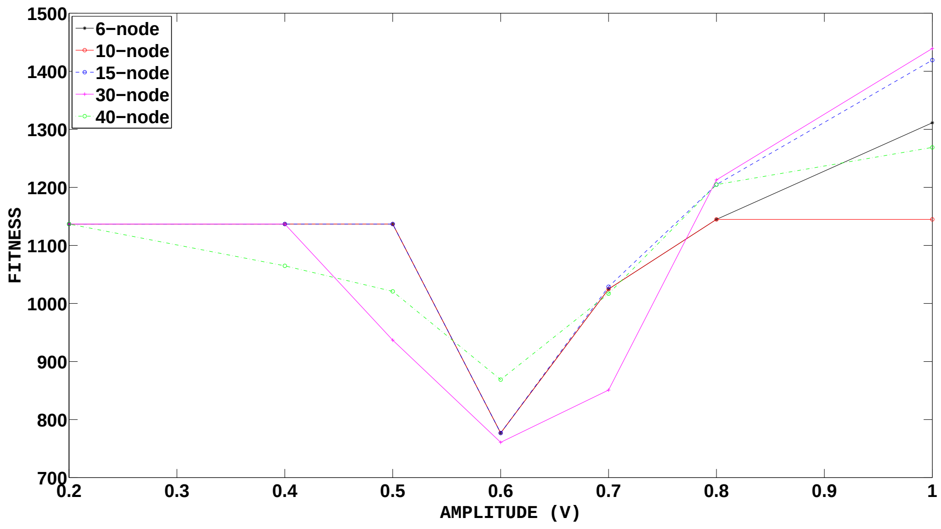


Figure 7.8: Fitness plot for triangular-square input pattern as a function of amplitude variation. The reservoir was trained for $0.6V$ amplitude.

change and hence the readout response varies significantly. This results in a fitness variation, which is seen in Figure 7.8. We conclude that signal variation causes changes in the reservoir states that are unique with respect to the change in the amplitude. This amplitude variation is reflected as a change in the fitness values.

7.4 Experiment 3: Pattern Recognition for Frequency Modulated Signal

To explore the memristor's dynamic characteristics that are dependent on variation in amplitude and frequency, we simulate memristor-reservoirs with more realistic input signals that are frequency modulated. We set the input amplitude to $0.5V$ with $300Hz$ carrier frequency, $100Hz$ low frequency signal with modulation index of $MI = 5$ and $MI = 4$. Modulation index relates to the variations in the carrier frequency. $MI = 5$ corresponds to the frequency variation in the input signal with constant signal amplitude and $MI = 4$ corresponds to the frequency variation in the input signal with the change in the signal amplitude. The input used is a *Single Frequency Frequency Modulated* (SFFM) source from the Ngspice voltage source library [17]. The expected readout response should converge to logic (0) for low frequency and logic (+1) for high frequency signal.

As discussed in Section 7.2 the higher node networks have higher simulation time and also have a higher memristor count associated with them. Hence, we choose to perform simulations on 6, 10, 15 and 30 node reservoirs. Table 7.4 summarizes the simulation results for input signals with varying modulation index.

Figure 7.9 and Figure 7.10, shows the response for input signal with $MI = 5$ and $MI = 4$ respectively. From Figure 7.9 it is observed that for $MI = 5$ the 30-node reservoir does not show convergence towards the expected target response with respect to the same node reservoir simulated for $MI = 4$ input.

This can be attributed to signal representation of the low frequency signal with respect to the high frequency signal and the reservoir topology. For the input with $MI = 5$, both the low and high frequency signal have the same amplitude range. For $MI = 5$, the modulating signal is varied from the range of 100Hz to 500Hz.

Frequency Modulated Input Pattern Recognition Experiment					
Modulation Index	Reservoir Size N	Memristor Count MC	Reservoir Figure Reference	Output Figure Reference	Fitness
$MI = 5$	30	76	Fig. 6.16	Fig. 7.9 (subplot A)	623.12
	15	35	Fig. 6.13	Fig. 7.9 (subplot B)	604.94
	10	40	Fig. 6.8	Fig. 7.9 (subplot C)	411.49
	6	19	Fig. 6.2	Fig. 7.9 (subplot D)	411.36
$MI = 4$	30	76	Fig. 6.16	Fig. 7.10 (subplot A)	94.31
	15	36	Fig. 6.11	Fig. 7.10 (subplot B)	94.47
	10	40	Fig. 6.8	Fig. 7.10 (subplot C)	94.36
	6	19	Fig. 6.2	Fig. 7.10 (subplot D)	94.26

Table 7.4: Fitness results obtained for SFFM input signal with $MI = 5$ and $MI = 4$.

For the 500Hz signal, the characteristics are more in the linear regime [5]. Also, as seen from the memristance property (see Section 2.4), for the high frequency and amplitude signals, the memristor show very less variation in its internal state. This is reflected in the signals not being distinguishable. In the case of reservoirs with lower memristor count i.e., 6, 10 and 15 node reservoirs, the signals are distinguished with sharp spikes for the high frequency signal and no spikes for low frequency signal, indicating a distinction between the the two signals. This is because the memristance effect is not completely canceled due to smaller topology and a lower memristor count. For $MI = 4$, the modulating signal varies from 100Hz to 400Hz. Memristors show change in the internal state for the 400Hz signal for the lower amplitude range. This is reflected in the converged results as seen in Figure 7.10.

Discussion: Comparing the reservoir performance for the input with $MI = 5$ with $MI = 4$, we observe that for signals with $MI = 5$, all the reservoirs except the 30-node reservoir are able to distinguish between the two signals. For the

$MI = 4$ input, all the reservoirs are able to robustly classify the two signals. Thus, we conclude that signal variation and topology plays an important role in differentiating the signals.

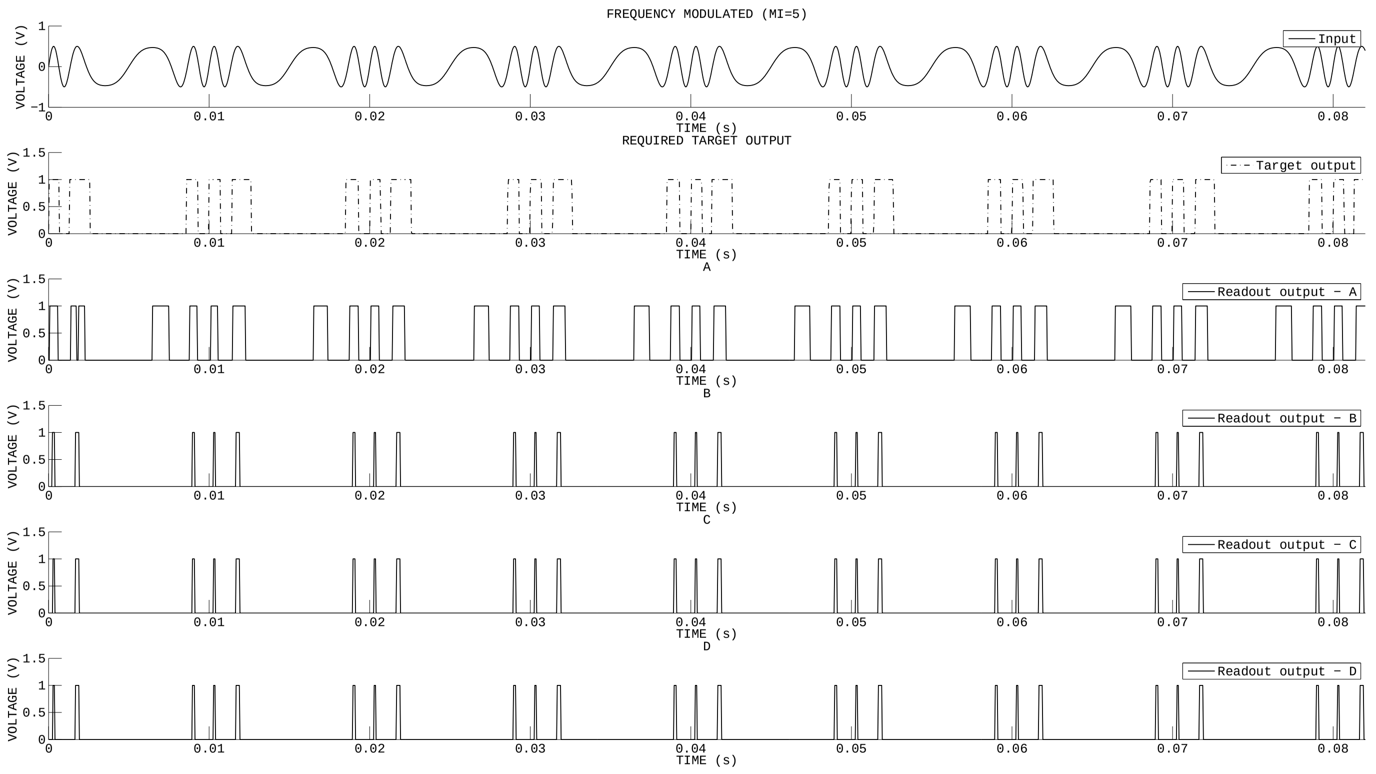


Figure 7.9: Simulation response for $MI = 5$. Describing the plot from (top to bottom). ($first$) Input to the reservoir; (*second*) the desired output response. Next, the readout responses for 30-node, 15-node, 10-node and 6-node reservoirs are shown from A to D respectively. For the high frequency input, the readout response for all the reservoirs converge to logic 1 and for the slow frequency input the response is logic 0 for all the reservoirs except for the 30-node. The logic 1 going pulses for 6-node, 10-node and 15-nodes are narrow pulses. Clearly, except for the 30-node reservoir, all other reservoirs are able to distinguish between the two signals.

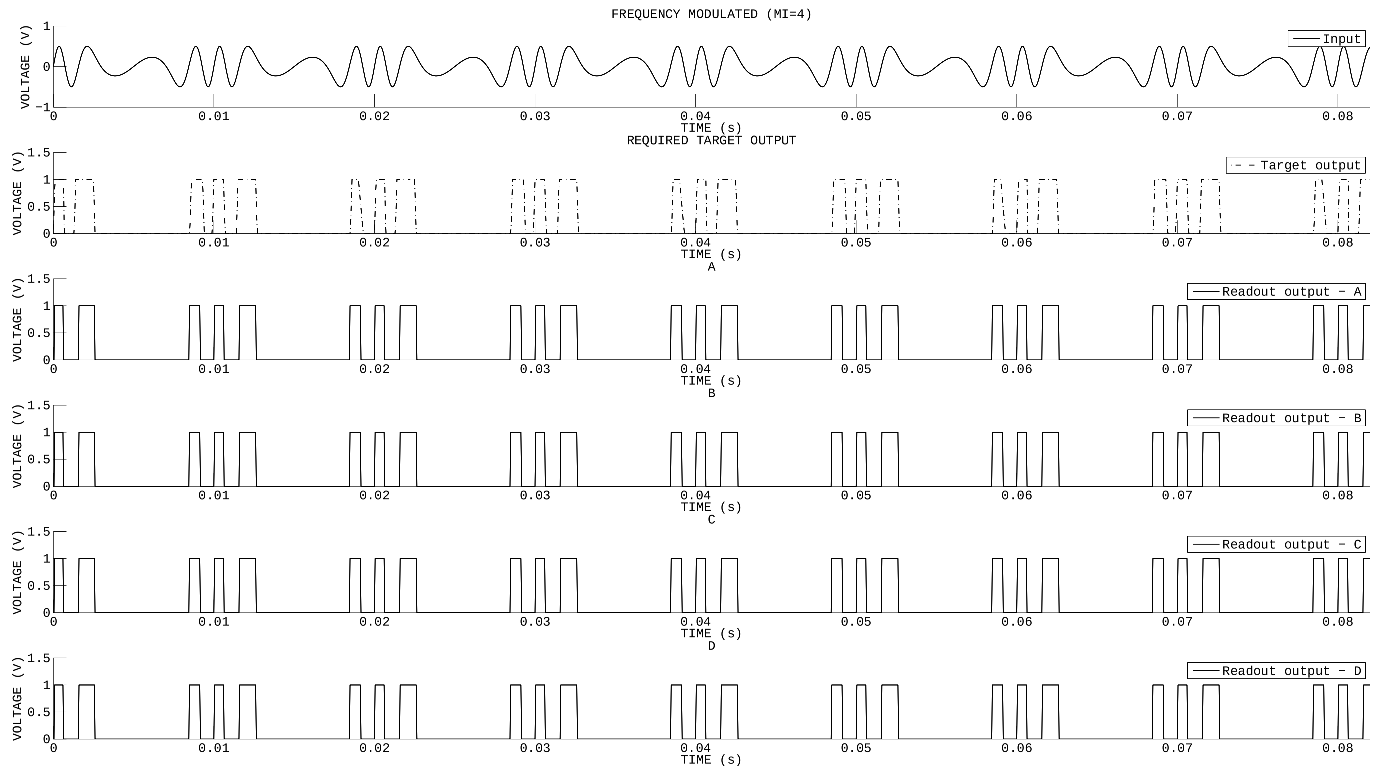


Figure 7.10: Simulation response for $MI = 4$. Describing the plot from (top to bottom). ($first$) Input to the reservoir; (*second*) the desired output response. Next, the readout responses for 30-node, 15-node, 10-node and 6-node reservoirs are shown from A to D respectively. For the high frequency input, the readout response for all the reservoirs converge to logic 1 and for the slow frequency input the response is logic 0. All the reservoirs are able to distinguish between the two signals.

7.5 Experiment 4: Pattern Recognition for Amplitude Modulated Signal

In one of the recent publication, Wey and Benderli [38] demonstrated amplitude modulation circuit architecture for a titanium dioxide TiO_2 memristor. Thus, we decided to explore the variation in memristance for an amplitude modulated signal using the reservoir computing architecture. In this experiment, the amplitude of the input signal is set to 0.5V with 100Hz carrier signal frequency and 500Hz modulating frequency. The target output is defined to represent logic (1) for high amplitude and logic (0) for low amplitude signal. The input used is a *Amplitude Modulated* (AM) source from the Ngspice voltage source library [17]. The expected readout response should converge to logic (0) for low amplitude and logic (+1) for high amplitude signal.

Reservoirs used in this experiment and their corresponding fitness values are summarized in Table 7.5. The simulated output response for all the reservoirs is shown in Figure 7.11.

Amplitude Modulated Pattern Recognition Experiment				
Reservoir Size N	Memristor Count MC	Reservoir Figure Reference	Output Figure Reference	Fitness
30	76	Fig. 6.16	Fig. 7.11 (subplot A)	512.58
15	36	Fig. 6.11	Fig. 7.11 (subplot B)	496.44
10	40	Fig. 6.8	Fig. 7.11 (subplot C)	496.56
6	19	Fig. 6.2	Fig. 7.11 (subplot D)	480.39

Table 7.5: Simulation results obtained for amplitude modulated signal.

Discussion: In this experiment, the readout uses the *step* activation function, which response is 0 for a negative signal and +1 for a positive signal. For example, for the low amplitude signal, the average reservoir activity is negative giving a zero response after the *step* activation as seen in Figure 7.11. It can be observed that all the reservoirs are able to distinguish the high amplitude signal as a positive spike and low signal as no spike. For amplitude modulated signals, the memristance change is a result of the variation in the signal amplitude. We have demonstrated that memristor-based reservoir computing can be used to distinguish amplitude variation in a signal. This experiment shows an important application for representing digital data (0 or 1) as variation in the signal amplitude.

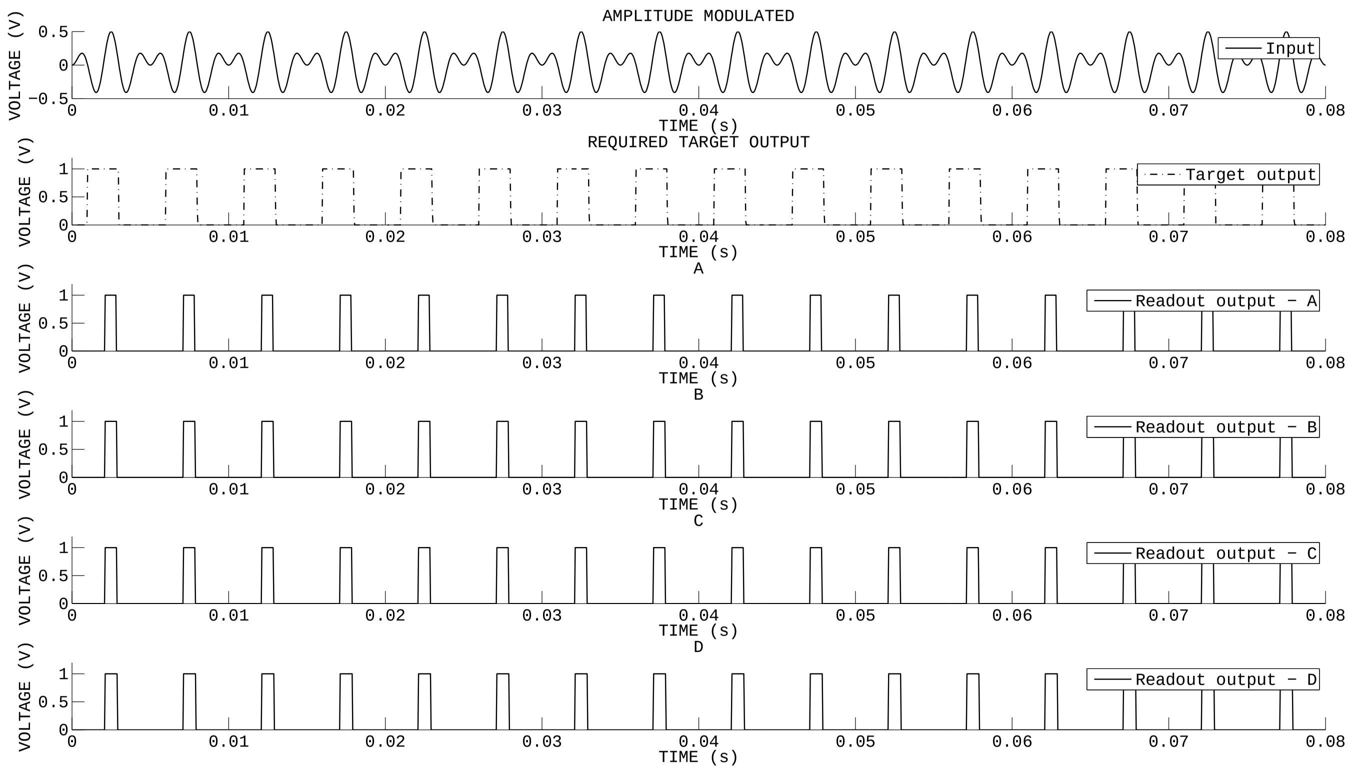


Figure 7.11: Simulation response for the amplitude modulated experiment. Describing the plot from (top to bottom). (*top*) Input to the reservoir; (*second*) The desired output response. Next, the readout responses for 30-node, 15-node, 10-node and 6-node reservoirs are shown from A to D respectively. For the high amplitude signal, the readout response should converge to logic 1 and for the low amplitude signal input the response is seen as pulses of logic 0. The readout uses the *step* function, whose response is 0 for a negative signal and +1 for a positive signal. Clearly, all the reservoirs are able to distinguish between the high and low amplitude for the input.

7.6 Experiment 5: Associative Memory

In this experiment we demonstrate ability of a memristor reservoir to solve the task of associating two different input signals. An associative memory is defined as the ability to associate different memories to specific events [7].

Pershin *et al.* in [7], experimentally demonstrated associative memory behavior based on Pavlov's famous example of associative memory behavior in dogs [39]. In this experiment Pavlov demonstrated that, the sight of food first (i.e., acting as first input) sets salivation of the dog's mouth. Then if, the sight of food is accompanied with a sound (i.e., second input) over a certain period of time, the dog learns to associate the sound with the food. Hence, when only presented with sound alone (i.e., second input), salivation can be triggered without the sight senses [7]. Pershin *et al.* [7] experimentally demonstrated this associative behavior between two different inputs using a simple neural network as shown in Figure 7.12. Here, N_1, N_2, and N_3 represent neurons and S_1 and S_2 represents synapse. In this experimental setup an analog to digital converter (ADC) emulates (N_1, N_2, and N_3) and the circuit emulating memristor property is configured as a synapse.

For this experiment, we define the inputs as V_A and V_B. Inputs are defined to be PULSE signal with amplitude of $+0.5V$ and period of 1ms. Here, V_A is the primary input and V_B is the secondary input that needs to be associated with V_A. The associative behavior output transient response is divided in four-phases:

1. **Phase A - Secondary Input**: When *only* input V_B is presented, output V_{TARGET} should not be activated.

2. **Phase B - Primary Input**: When *only* input V_A is presented, output V_{TARGET} should be activated.

3. **Phase C - Learning Phase**: When *both* input V_A and V_B are presented, output V_{TARGET} should be strongly activated.

4. **Phase D - Response Phase**: When *only* input V_B is presented, output V_{TARGET} should be activated.

In this experimental setup, we choose reservoirs with $6, 10$ and 15 nodes. Table 7.6 summarizes simulated reservoir sizes and corresponding fitness values. The output response shown in Figure 7.14 is clearly marked with four phases described above.

Associative Memory Experiment				
Reservoir Size N	Memristor Count MC	Reservoir Figure Reference	Output Figure Reference	Fitness
15	35	Fig. 6.13	Fig. 7.14 (subplot 4)	6.060
10	25	Fig. 6.7	Fig. 7.14 (subplot 5)	9.006
6	21	Fig. 6.1	Fig. 7.14 (subplot 6)	5.701

Table 7.6: Fitness values obtained for the associative memory experiment.

For example we consider 10-node and 15-node reservoir response shown in Figure 7.14. From the ideal output response it can be observed that during $phaseA$ there should not be any output activation. But, we observe some activations for both the reservoir's readout in $phaseA$. In the real world noise is introduced in the system due to device mechanics and component connectivity. Here, we define an acceptable threshold level as $0.2V$ to determine the validity of the reservoir response. Thus, we can observe that the $phaseA$ response is valid, which is below the set threshold level.

Discussion: Comparing the reservoir performance for the 6-node, 10-node and 15-node reservoirs, we conclude that only the 10-node and the 15-node reservoirs introduces noise in the $phaseA$ output. This noise is below the pre-defined threshold level and hence is acceptable. Also, no noise is introduced in the output phases from B to D. We also observe that the output response show amplitude variation across the phases from B to D; this is due to the non-linear memristance change, which is reflected as the non-linear voltage change across the memristors. Thus, in conclusion, the response obtained is because of the non-linear memristor characteristics, which cannot be obtained given a resistor topology. Noise is introduced due to the passive device behavior and their interconnectivity. Note that acceptable noise levels is ultimately a design choice.

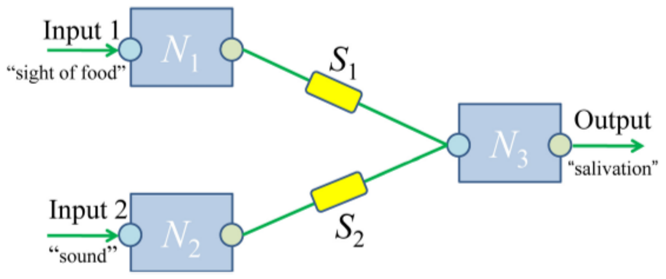


Figure 7.12: Simple neural network. Here, N_1, N_2, and N_3 represent neurons and S_1 and S_2 represent synapse. (Source: [7]).

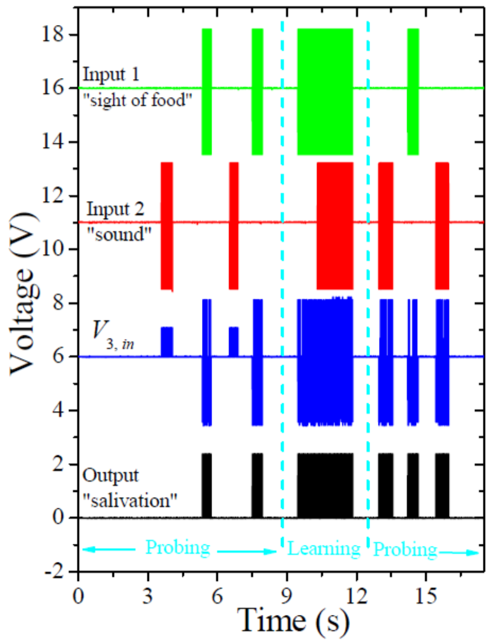


Figure 7.13: Output response from the electrical circuit emulating associative memory using a neural network [7]. *Input1* represents (sight of food), *Input2* represents (sound) and the probing phase of the output represents salivation which indicated the learning behavior when the only input present is the sound. (Source: [7]).

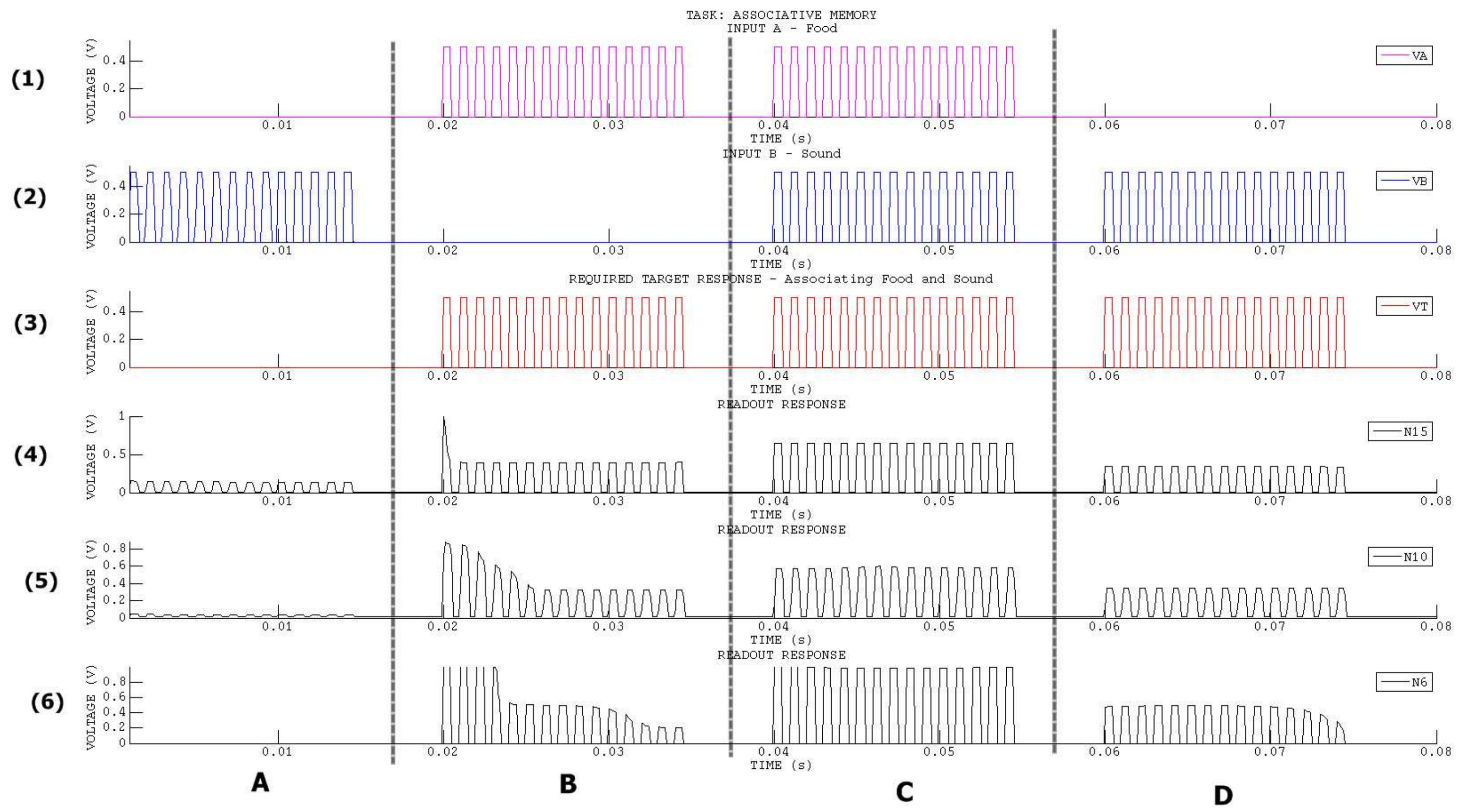


Figure 7.14: Describing the plots from *top* to the *bottom*. (1) shows the first input signal V_A representing the $food$ signal; (2) shows the second input signal V_B representing the *sound*; (3) is the required target response. The response for the associative memory is marked with 4 phases from $A - D$. In $PhaseA$, when only input V_B presented, the output should not show any response. In $phaseB$, when only input V_A is presented, the output should show a strong response. In $PhaseC$, which is the training phase when both inputs V_A and V_B are presented, the output should respond. In $phaseD$, when only input V_B is presented, the output should show response, indicating the system has learned to respond in absence of the main input V_A. $(4 - 6)$ is the readout response obtained for the 15-node, 10-node and 6-node reservoirs respectively. We observe that all the resvoirs respond to the required target response, except for some noise in $PhaseA$ which is within an acceptable range.

7.7 Experiment 6: Logical Computation

Memristor based logic circuit have been demonstrated using a structured crossbar architecture approach in [40]. In one of the recent publications by Daniel *et al.* in [41] logical AND operation has been demonstrated using only memristors.

We conducted experiments to show basic logical computation AND, OR and XOR using the reservoir computing approach. Table 7.7 summarizes the fitness results and the reservoirs used in this experiment. Logic implementation using memristors require less than four memristor count as shown by Raja *et al.* and Batas *et al.* [40,41]. Hence, we demonstrate the logic implementation using smaller reservoirs i.e., 6-node and 10-node reservoirs with a lower memristor count as compared to the 15-node or 30-node reservoirs.

Figure 7.15 shows the output response for the OR gate. We can clearly observe the internal state change for the memristor element. The output shows the voltage level of approximately $0.4V$ when only one input is present representing the (1 0) condition and when both the inputs are present, we observe the voltage changes from $0.4V$ to approximately $0.6V$ representing the (1 1) condition. Similarly, the response for logical AND computation is shown in Figure 7.16. Here, we observe that the output shows a reduced voltage level of $0.4V$ when an $0.5V$ input is presented. From Figure 7.17 representing logical XOR computation, we observe that the responses do not show convergence towards the desired output response.

Discussion: Comparing the logical computation response for the AND and OR and XOR operation for 6-node and 10-node reservoirs, we conclude that both the reservoirs performed equally well for AND and OR operations, while none of the reservoirs were able to converge towards the expected XOR response. This maybe

Logical Computation Experiment					
Logical Operation	Reservoir Size N	Memristor Count MC	Reservoir Figure Reference	Output Figure Reference	Fitness
OR	10	30	Fig. 6.6	Fig. 7.15	5.476
	6	14	Fig. 6.3	Fig. 7.15	5.537
AND	10	33	Fig. 6.10	Fig. 7.16	0.00823
	6	13	Fig. 6.4	Fig. 7.16	0.0104
XOR	10	33	Fig. 6.10	Fig. 7.17	17.87
	6	14	Fig. 6.3	Fig. 7.17	17.714

Table 7.7: Table summarizes the simulation results obtained for logical OR operation.

because XOR is not a linearly separable function. In Figures 7.15 and 7.16, the characteristic voltage drop seen for the logical OR and AND operation is due to the non-linear state change of memristors and is not possible by using only resistors. We conclude that memristor-based reservoir computing can be used to perform simple logical operations.

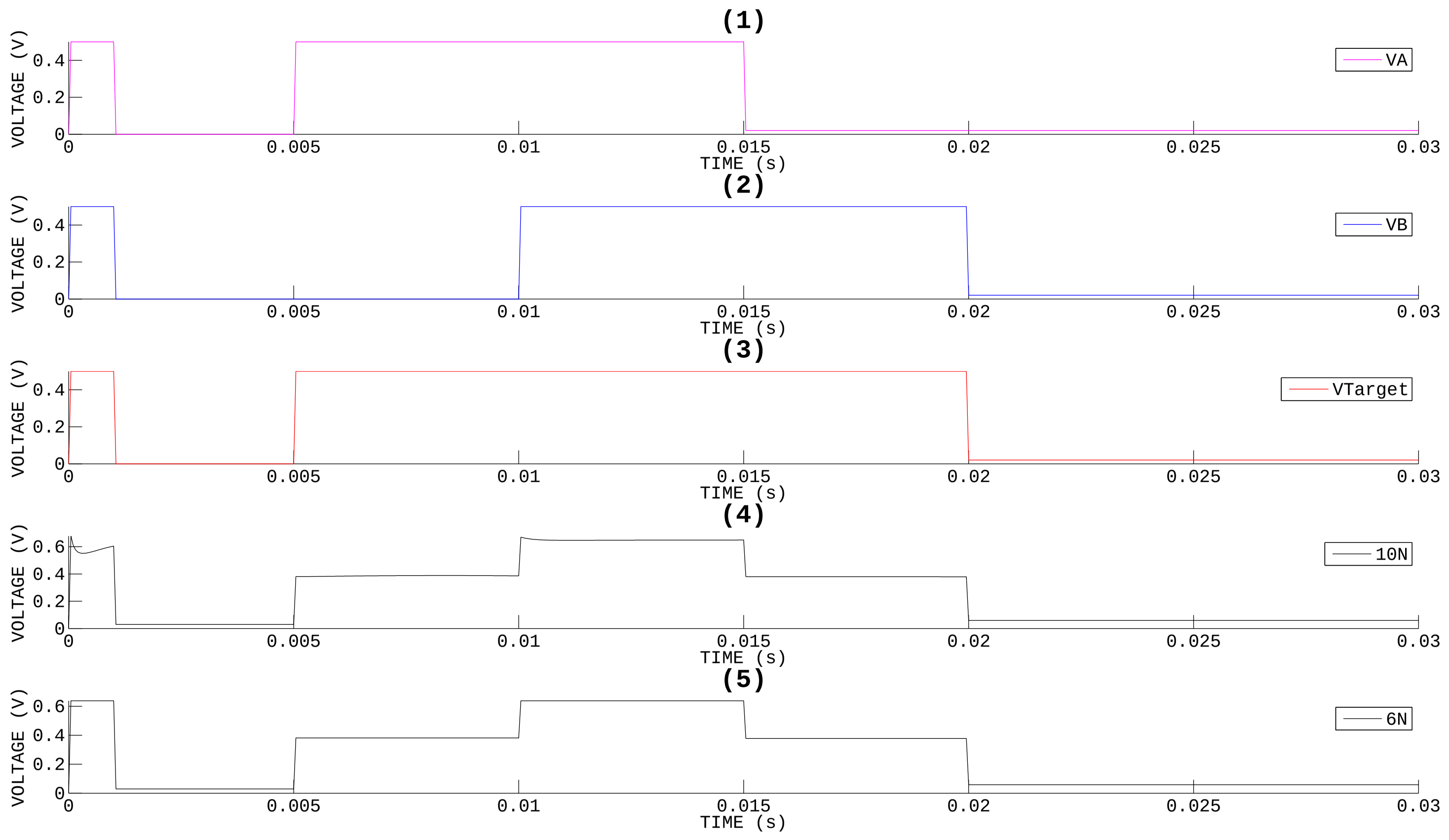


Figure 7.15: Simulation response for logical OR operation. Here, (1) and (2) are inputs A and B respectively with amplitude of 0.5V. The desired output response is shown in (3) and the readout output response for 10-node and 6-node is shown in (4) and (5) respectively.

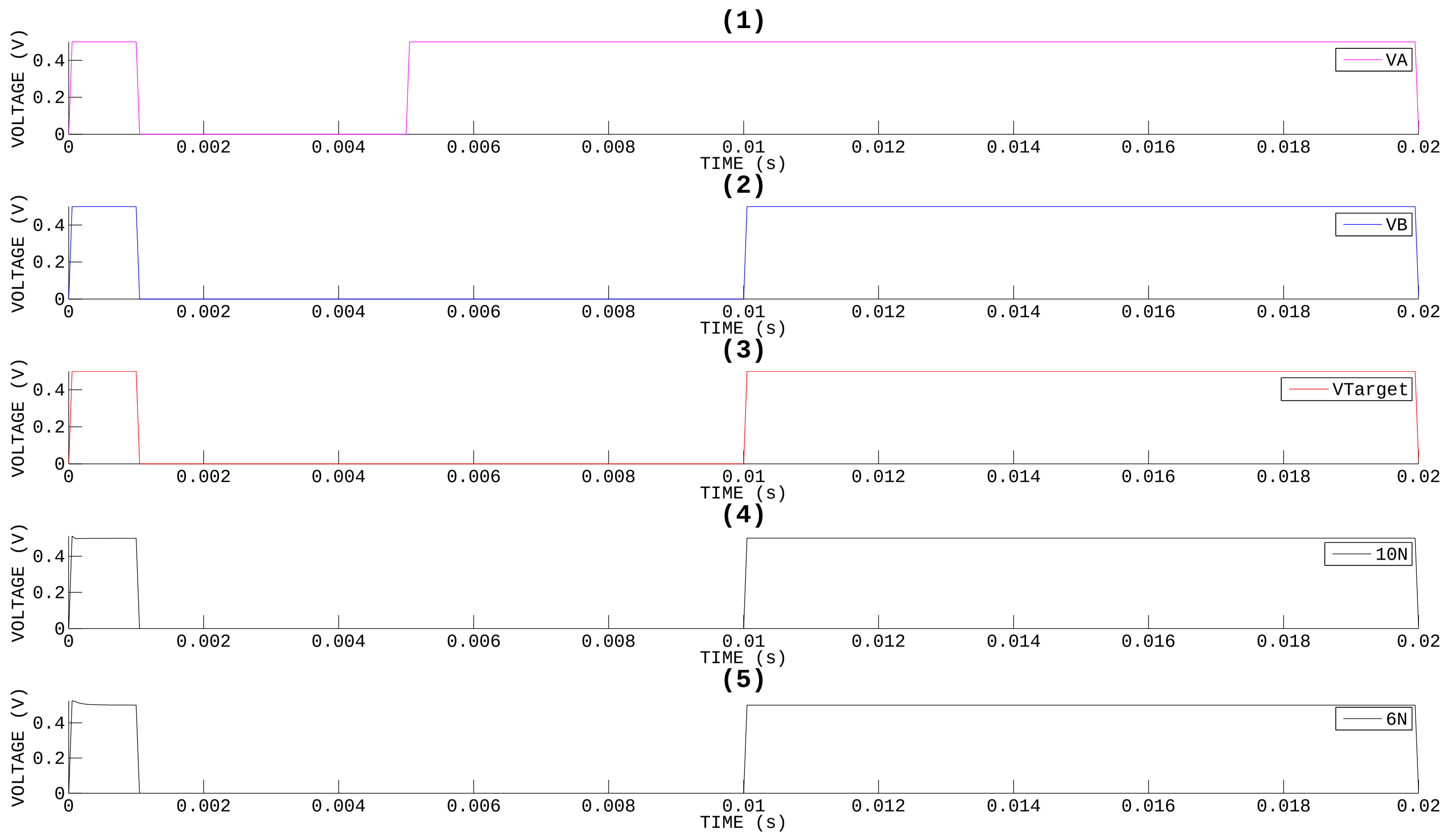


Figure 7.16: Simulation response for logical AND operation. Here, (1) and (2) are inputs A and B respectively with amplitude of 0.5V. The desired output response is shown in (3) and the readout output response for 10-node and 6-node is shown in (4) and (5) respectively.

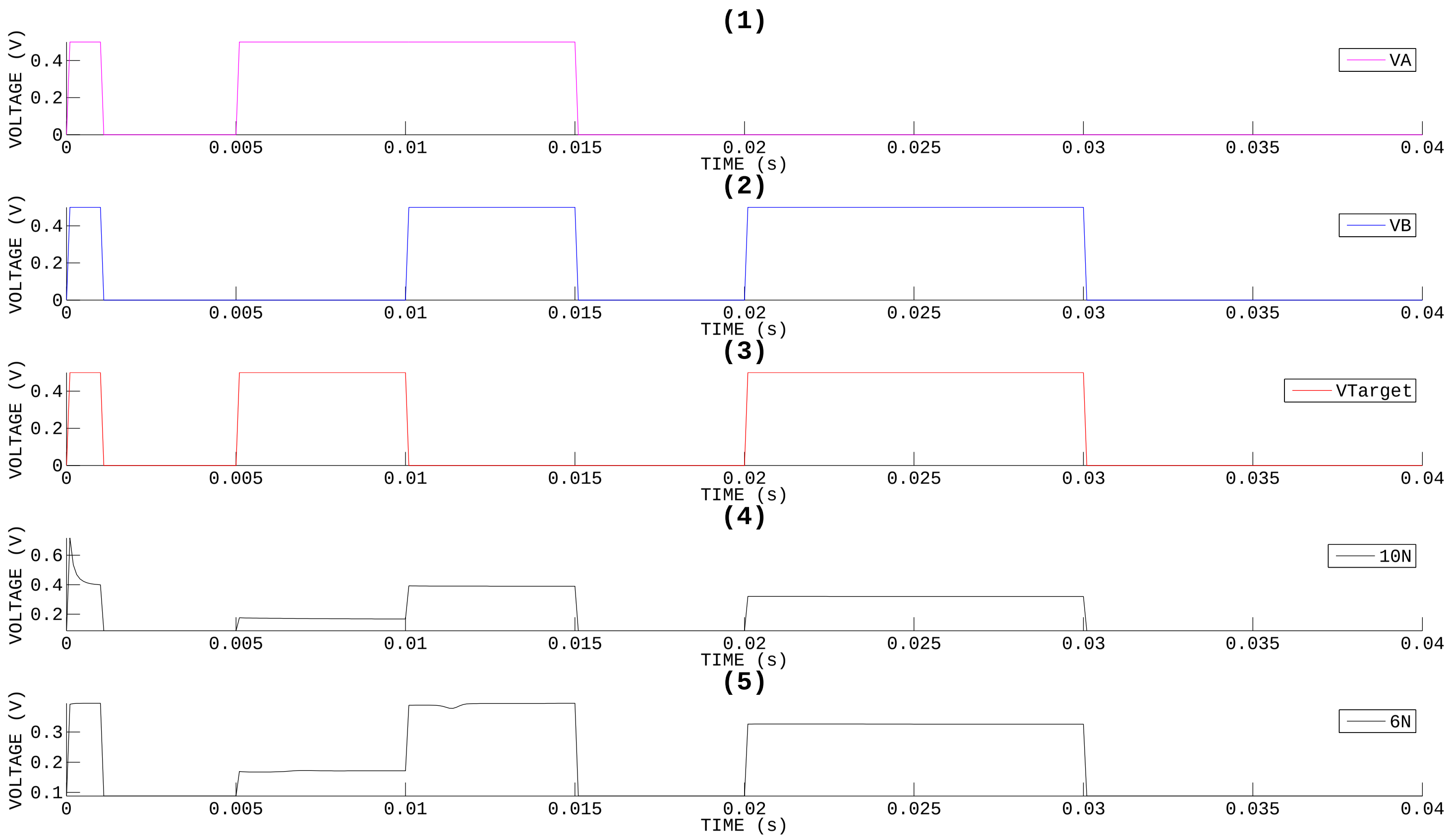


Figure 7.17: Simulation response for logical XOR operation. Here, (1) and (2) are inputs A and B respectively with amplitude of 0.5V. The desired output response is shown in (3) and the readout output response for 10-node and 6-node is shown in (4) and (5) respectively.

8

Conclusion and Future Work

In this thesis, we have presented a memristor-based reservoir computing architecture. To the best of our knowledge, memristors have never been used as reservoir components. Our approach to explore the inherent properties of nanoscale devices using this novel reservoir computing architecture is an important step that provides a unique perspective on the computational power of a random dynamical networks to perform particular tasks. Our framework can also be expanded to explore new nanoscale devices, such as the *memcapacitor* and the *meminductor*. For this thesis we have developed a Matlab-based framework called the *Reservoir Computing Simulation Program with Integrated Circuit Emphasis* (RCspice) for this purpose.

Using this software simulator, we demonstrated the generation of a N-node random memristor reservoir using a graph-based approach. The performance of the reservoir, i.e., the ability to solve a task is evaluated using a genetic algorithm.

In our first experiment, we evaluated the reservoir size and the GA parameters for the triangular-square pattern recognition benchmark task. The reservoir size is a measure of the memristor's dynamic state space and the GA parameters evaluate the quality of the solution. We chose to explore 6-node, 10-node, 15-node, 30-node and 40-node reservoirs. The optimum GA parameter set was found to be a mutation rate of 0.05, a population size of 50 and reinsertion rate of 0.08. Our results showed that for the 30-node and 40-node reservoirs, the GA required more than $7,500$ generations to reach the target solution as compared to the 6-node, 10-node and 15-node reservoirs which required less than $3,000$ generations. This is a because of the larger search space the GA has to explore. We conclude that the 6-node, 10-node and 15-node reservoirs showed the best performance with respect to the final solution obtained and the computational time.

The optimum GA parameter set found in the first experiment was used for all the subsequent experiments. Next, we demonstrated the capability of the memristor-reservoirs for pattern recognition benchmark tasks. This included the triangular-square, the frequency and the amplitude modulated input patterns. For the first task, i.e., triangular-square pattern recognition was performed on different reservoir sizes and connectivity. On an average, all the reservoirs converged towards the required target response. But the evaluation for the 30-node and 40-node reservoirs showed higher fitness values and computation time in comparison with the smaller reservoirs. To study the reservoir characteristics for signal variation, we performed a variation experiment by sweeping the signal amplitude from $0.2V$ to $1V$ for the triangular-square input pattern. We observed that for amplitude variation, the internal reservoir state space is unique, which is reflected as a change in the fitness response with respect to the applied signal variation.

Next, to study the frequency-dependent characteristics for memristors in a network, we simulated reservoirs with frequency modulated signals with variation in the modulation index MI, i.e., $MI = 5$ and $MI = 4$. Our results showed that for $MI = 5$ signal, the 30-node reservoir did not convergence towards the desired response and smaller size reservoirs, i.e., 6, 10 and 15 nodes showed signs of convergence. While for $MI = 4$, all the reservoir sizes robustly converged towards the desired response. Next, to study the response of variation in the amplitude, we applied an amplitude modulated signal. We observed that all the memristor-reservoirs can robustly distinguish between the signals with different amplitude. In summary, we learned that the reservoir connectivity, the size, and the type of the applied input are key parameters.

To explore the memristor's memory property, we performed a benchmark associative memory experiment on 6-node, 10-node and 15-node reservoirs. Our results show that the memristor reservoirs are able to demonstrate the associative behavior properly. This experiment is an important step that demonstrated the memory behavior using reservoir computing, which can be further expanded for more complex tasks. Although memristors are been explored for different applications, the basic logic gates are important building blocks from a circuit perspective. Hence, we tested our memristor reservoirs for the basic logical computation i.e., OR, AND and XOR gates. Since these tasks do not demand large number of memristors, we performed experiments on smaller reservoirs only, i.e., 6-node and 10-node reservoirs. Our results show that the reservoirs were able to solve both the OR and AND task except the XOR task.

Some of the limitations for evaluating the reservoir performance are due to the memristor SPICE model. The model used for our framework is a reasonable model that is based on the original TiO_2 physical device. Using a more accurate SPICE model that allows for fast simulation time and higher number of memristor elements would help us to gain a more detail insight into the reservoir state space.

We conclude that the reservoir topology and size are important parameters for a given task. Large reservoir topologies have an disadvantage with respect to the average power consumed. This is because in a large reservoir, some of the memristor elements are non-varying and simply act as a resistor. These non-varying elements contribute towards the static average power.

For future work, we would like to extent our framework to study structured reservoir topologies and validate our hypothesis by implementing our approach on a physical memristor network hardware.

References

[1] "The genetic algorithm toolbox for MATLAB." `http://www.shef.ac.uk/acse/research/ecrg/gat`.

[2] C. Fernando and S. Sojakka, "Pattern recognition in a bucket," *Advances in Artificial Life*, pp. 588–597, 2003.

[3] "Water drop." `www.theroadtothehorizon.org`.

[4] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, May 2008.

[5] A. Rak and G. Cserey, "Macromodeling of the memristor in SPICE," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 632–636, Apr. 2010.

[6] Z. Gan, Z. Yang, T. Shang, T. Yu, and M. Jiang, "Automated synthesis of passive analog filters using graph representation," *Expert Systems with Applications*, vol. 37, no. 3, pp. 1887–1898, 2010.

[7] Y. Pershin and M. Di Ventra, "Experimental demonstration of associative memory with memristive neural networks," *Nature Precedings*, vol. 23, no. 7, pp. 881–886, 2010.

[8] "International technology roadmap for semiconductors ITRS, semiconductor industry association." `http://www.itrs.net/Links/2010ITRS/Home2010.htm`, 2010.

[9] V. Zhirnov and D. Herr, “New frontiers: Self-assembly and nanoelectronics,” *Computer*, vol. 34, no. 1, pp. 34–43, 2001.

[10] R. I. Bahar, D. Hammerstrom, J. Harlow, W. H. J. Jr., C. Lau, D. Marculescu, A. Orailoglu, and M. Pedram, “Architectures for silicon nanoelectronics and beyond,” *Computer*, vol. 40, pp. 25–33, Jan. 2007.

[11] C. Teuscher, “On irregular interconnect fabrics for self-assembled nanoscale electronics,” *2nd IEEE Int. Workshop on Default and Fault Tolerant Nanoscale Architectures, NANOARCH’06*, pp. 60–67, 2006.

[12] J. W. Lawson. and D. H. Wolpert, “Adaptive programming of unconventional nano-architectures,” *Journal of Computational and Theoretical Nanoscience*, vol. 3, pp. 272–279, 2006.

[13] B. Cooper and A. Sorbi, *Computability in Context: Computation and Logic in the Real World*, ch. Motivation, theory, and applications of liquid state machines, pp. 275–296. World Scientific, 2011.

[14] H. Jaeger, “The echo state approach to analysing and training recurrent neural networks with an erratum note 1,” *Technical Report GMD Report 148 German National Research Center for Information Technology*, pp. 1–47, 2010.

[15] R. Williams, “How we found the missing memristor,” *Spectrum, IEEE*, vol. 45, no. 12, pp. 28–35, 2008.

[16] A. Sinha, M. S. Kulkarni, and C. Teuscher, "Evolving nanoscale associative memories with memristors," *IEEE Nano In Proceedings of the 11th International Conference on Nanotechnology Conference on Nanotechnology*, pp. 860–864, 2011.

[17] "Ngspice." `http://ngspice.sourceforge.net/`.

[18] S. Haykin, *Neural Networks: A Comprehensive Foundation 3rd Edition*. Prentice-Hall, Inc., 2007.

[19] J. A. Anderson, *An Introduction to Neural Networks*. Cambridge, MA: MIT Press, 1995.

[20] M. Lukosevicius and H. Jaeger, "Reservoir computing approaches to recurrent neural network training," *Computer Science Review*, vol. 3, no. 3, pp. 127–149, 2009.

[21] J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, p. 2554, 1982.

[22] R. Legenstein and W. Maass, "What makes a dynamical system computationally powerful," *New directions in statistical signal processing: From systems to brain*, pp. 127–154, 2007.

[23] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: A new framework for neural computation based on perturbations," *Neural Computation*, vol. 14, no. 11, pp. 2531–2560, 2002.

[24] H. Jaeger, W. Maass, and J. Principe, "Special issue on echo state networks and liquid state machines," *Neural Networks*, vol. 20, no. 3, pp. 287–289, 2007.

[25] W. Maass, T. Natschläger, and H. Markram, *Computational models for generic cortical microcircuits*, pp. 575–605. No. ISBN 1-58488-362-6, CRC-Press, 2004.

[26] B. Schrauwen, M. D'Haene, D. Verstraeten, and J. V. Campenhout, "Compact hardware liquid state machines on fpga for real-time speech recognition.," *Neural Networks*, vol. 21, no. 2-3, pp. 511–523, 2008.

[27] B. Jones, D. Stekel, J. Rowe, and C. Fernando, "Is there a liquid state machine in the bacterium escherichia coli?," in *Artificial Life, 2007. ALIFE '07. IEEE Symposium on*, pp. 187–191, April 2007.

[28] K. Vandoorne, W. Dierckx, B. Schrauwen, D. Verstraeten, R. Baets, P. Bienstman, and J. Van Campenhout, "Toward optical signal processing using photonic reservoir computing," *Optics Express*, vol. 16, no. 15, pp. 11182–11192, 2008.

[29] Y. Paquot, F. Duport, A. Smerieri, J. Dambre, B. Schrauwen, M. Haelterman, and S. Massar, "Optoelectronic reservoir computing.," *Sci Rep*, vol. 2, p. 287, 2012.

[30] W. Maass, R. Legenstein, and H. Markram, "A New Approach towards Vision Suggested by Biologically Realistic Neural Microcircuit Models," pp. 1–6, 2009.

[31] L. Chua, "Memristor-the missing circuit element," *Circuit Theory, IEEE Transactions on*, vol. 18, no. 5, pp. 507–519, 1971.

[32] D. Strukov and R. Williams, “Four-dimensional address topology for circuits with stacked multilayer crossbar arrays,” *Proceedings of the National Academy of Sciences*, vol. 106, no. 48, p. 20155, 2009.

[33] Z. Biolek, D. Biolek, and V. Biolková, “SPICE model of memristor with nonlinear dopant drift,” *Radioengineering*, vol. 18, no. 2, pp. 210–214, 2009.

[34] MATLAB, “version 7.13.0.564 (r2011b).” `http://www.mathworks.com`, 2011.

[35] “Open source graph visualization software Graphviz.” `http://www.graphviz.org/`.

[36] F. Boesch and J. McHugh, “Synthesis of biconnected graphs,” *Circuits and Systems, IEEE Transactions on*, vol. 21, no. 3, pp. 330–334, 1974.

[37] Q. Xia, W. Robinett, M. W. Cumbie, N. Banerjee, T. J. Cardinali, J. J. Yang, W. Wu, X. Li, W. M. Tong, D. B. Strukov, G. S. Snider, G. Medeiros-Ribeiro, and R. S. Williams, “Memristor-CMOS hybrid integrated circuits for reconfigurable logic,” *Nano Letters*, vol. 9, pp. 3640–3645, Oct. 2009.

[38] T. A. Wey and S. Benderli, “Amplitude modulator circuit featuring tio2 memristor with linear dopant drift,” *Electronics Letters*, vol. 45, no. 22, pp. 0–1, 2009.

[39] I. P. Pavlov, “Conditioned reflexes: An investigation of the physiological activity of the cerebral cortex (translated by G. V. Anrep),” *London: Oxford University Press*, 1927.

[40] T. Raja and S. Mourad, “Digital logic implementation in Memristor-Based crossbars - a tutorial,” pp. 303–309, IEEE, 2010.

[41] D. Batas and H. Fiedler, "A memristor spice implementation and a new approach for magnetic flux-controlled memristor modeling," *Nanotechnology, IEEE Transactions on*, vol. 10, pp. 250–255, march 2011.