Portland State University

PDXScholar

1-1-2011

Evolving Nano-scale Associative Memories with Memristors

Arpita Sinha
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Let us know how access to this document benefits you.

Evolving Nano-scale Associative Memories with Memristors

by

Arpita Sinha

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
Xiaoyu Song

Portland State University
2011

Abstract

Associative Memories (AMs) are essential building blocks for brain-like intelligent computing with applications in artificial vision, speech recognition, artificial intelligence, and robotics. Computations for such applications typically rely on spatial and temporal associations in the input patterns and need to be robust against noise and incomplete patterns. The conventional method for implementing AMs is through Artificial Neural Networks (ANNs).

Improving the density of ANN based on conventional circuit elements poses a challenge as devices reach their physical scalability limits. Furthermore, stored information in AMs is vulnerable to destructive input signals. Novel nano-scale components, such as memristors, represent one solution to the density problem. Memristors are non-linear time-dependent circuit elements with an inherently small form factor. However, novel neuromorphic circuits typically use memristors to replace synapses in conventional ANN circuits. This sub-optimal use is primarily because there is no established design methodology to exploit the memristor's non-linear properties in a more encompassing way.

The objective of this thesis is to explore denser and more robust AM designs using memristor networks. We hypothesize that such network AMs will be more area-efficient than the traditional ANN designs if we can use the memristor's non-linear property for spatial and time-dependent temporal association. We have built a comprehensive simulation framework that employs Genetic Programming (GP) to evolve AM circuits with memristors. The framework is based on the ParadisEO metaheuristics API and uses *ngspice* for the circuit evaluation.

Our results show that we can evolve efficient memristor-based networks that have the potential to replace conventional ANNs used for AMs. We obtained

AMs that a) can learn spatial and temporal correlation in the input patterns; b) optimize the trade-off between the size and the accuracy of the circuits; and c) are robust against destructive noise in the inputs. This robustness was achieved at the expense of additional components in the network.

We have shown that automated circuit discovery is a promising tool for memristor-based circuits. Future work will focus on evolving circuits that can be used as a building block for more complicated intelligent computing architectures.

Acknowledgements

I would like to express my heart-felt gratitude to a number of people without whose support this thesis would not have been possible. My advisor Dr. Christof Teuscher for his continuous guidance and encouragement throughout the course of my graduate studies. Alireza and Haera, with whom I had many interesting and stimulating discussions. Prof. Dan Hammerstrom and Prof. Xiaoyu Song for agreeing to serve as members of my thesis committee. Fellow graduate students (both past and present) of our lab for their help, inspiration, exchange of ideas, and sincere feedback. My friends at PSU, for making the time spent here a memorable one. And most of all, my husband for his unending support and love.

Contents

List of Tables

List of Figures

x

1

Introduction

The real-world computational problems today have to deal with massive quantities of low precision and ambiguous data. An example for such problems can be seen in artificial intelligence that deals with unifying computer vision, speech recognition, content and context recognition etc. [53]. Conventional computational models used in numerical simulations, model fitting, data analysis etc. are inherently designed to solve problems that are precise and well-defined. Additionally, they lack the ability to learn from the complex relationships that exist in space and time. Hence, conventional models are inefficient in solving this class of problems. There is an increasing interest in using neuro-cortical models as inspiration for intelligent computing. Intelligent computing here refers to inference-based models that act upon real world data and learn/adapt while computing results. The motivation behind developing intelligent computing models is the human brain. In support, the *BrainScaleS* program in Europe [31] and *DARPA SyNAPSE* program in the US [28] intend to realize novel computing paradigms that exploit the observations in biological nervous systems.

Associative Memories (AMs) are regarded as essential building blocks for the human brain [2]. Hence, drawing from biological inspiration, AMs in intelligent computing are candidate building blocks capable of memorization as well as learning and adaptation. An AM block based on a robust circuit design would allow the realization of hierarchical models.

We hypothesized memristor networks can be useful in robust nano-scale AM

designs. In this thesis, we propose a novel methodology for implementing AMs using memristors as circuits elements.

1.1 High-Level Associative Memories: Biological Implication

An Associative Memory (AM) has the ability to associate different memories to specific events. Such memories form an integral part of cognition in life forms, including humans [2]. This ability allows the brain to react or adapt to external stimuli based on past experiences. The famous Pavlov experiments [47] are a good example of associative memory: Pavlov observed that if a particular stimulus in the dog's surroundings was present when the dog was presented with meat powder, this stimulus would become associated with food and cause salivation on its own. The desired characteristics of associative memories have been summarized by Pao [45] as follows:

- It should store many associated pattern pairs through a self-organizing process in a distributed manner.

- It should generate the appropriate response output response despite distorted or incompletely received inputs.

- It should dynamically append new associations to the existing stored memory.

Research in high-level AM models inherits these important properties along with biological plausibility and fast training times. Central to these ideas is the learning matrix [60], the Hopfield network [27], the bidirectional associative memory (BAM) [33], and the hierarchical temporal memory (HTM) [20]. The learning matrix adapts the connection weights using a Hebbian learning rule [24]. In case of

the Hopfield network, feedback creates a system which uses input and output patterns to represent its states. The BAM is similar to the Hopfield network, but has two layers of neurons, and additionally uses connection matrix to calculate both a) outputs given some inputs and b) inputs given a set of outputs. The HTM concept involves having a hierarchy of spatial and temporal operators with multiple nodes in each layer of the hierarchy. All nodes perform identical computations except for the top layer node, which has additional features for performing classification.

AMs also form a specific area of research within self-organizing systems, commonly referred as *Kohonen networks or self-organized maps* [32]. Kohonen networks are rigorous mathematical models for two dimensional array of neurons, where the weight of each element corresponds to its coordinates in an ordered map. There are numerous application areas for high-level AMs within the intelligent computing paradigm, such as: pattern recognition, language learning, fact retrieval, inference and decision making, robotic controls etc. [2].

High-level AM models as above rely on low-level building blocks that can associate pairs of inputs or detect sequence/context within an encoded input stream. Low-level AMs have been traditionally implemented using *Artificial Neural Networks* (ANNs) [58]. The next section describes traditional designs and associated challenges.

1.2 Associative Memory Implementation with Artificial Neural Networks: Design and Challenges

We chronologically review here some traditional designs for electronic ANN implementations of AMs, with an emphasis on the challenges in their large-scale integration.

The first description of ANN integrated circuit [58] implements a continuous-time analog circuit for AM. The design used a 22 x 22 matrix with 20,000 transistors, averaging 40 transistors per node to implement a Hopfield AM network. The design faced a scalability challenge at higher levels of integration. The paper advocates handling larger problems by a collection of smaller networks or hierarchical solutions, while predicting, "significantly different connection technologies" as essential for success in larger systems.

The next seminal work by Sage and Withers [54] built AMs using discrete-time analog technology for high-speed computation in combination with analog nonvolatile storage for synaptic weights. The network demonstrated was a 9x9 Hopfield associative memory network. The issue with the design was that although the synaptic weights could dynamically adapt, there were only three possible states to the weights (1, 0, -1). Thus, the network could demonstrate learning for a very few specific computations only. The message from this study was that a continuous range weights would be a desirable feature for the synapses. In an attempt to achieve high resolution synaptic weights, Schwartz and Howard [57] proposed representing each weight as a difference in voltage between two capacitors. With the additional circuitry for sense-amplifiers, a 32 x 32 matrix with 75,000 transistors averaged 70 transistors per neural node. The high-level integration required scaling of the components to nano-scale levels and further simplification of the node design.

Other efforts from Holler *et al.* [26] use floating gate technology for the representation of synaptic weights to achieve higher synapse density, but the design has electrically programmable static weights, and the dynamics of input presentation

4

has no bearing on the real-time network associations. A mix of 8 x 8 matrix of digitally stored weights gate the inhibitory/excitatory pulse stream from 4 x 4 input layer. The pulse stream generation, integration and modulation results in much lower densities (140 transistors per neural node) than the aforementioned designs.

Among biological applications, Lyon *et al.* [40] implemented an electronic analog equivalent for the human cochlea (inner-ear). The design uses CMOS transconductance amplifiers circuits, follower-integrator circuits and second-order filter circuits to emulate perceptron machines. The authors see inherent deficiencies with digital threshold logic and emphasize the need for high-density analog learning-based implementations for more precise biological equivalence.

Hammerstrom *et al.* demonstrated one of the first custom digital ANN processor *CNAPS* [21]. The CNAPS architecture, customized for ANN simulations, had significant performance *vs.* cost improvements over arrays of commercial microprocessors. The authors proposed that further speed-ups could be achieved by exploiting the high-speed memory structure and the inherent parallelism of field-programmable-gate-arrays (FPGAs). Along the lines of exploiting the FPGA advantage, Changian *et al.* [8] have demonstrated a best-match association using distributed representations on FPGA hardware. Similarly, Deshpande [12] implemented a Bayesian-memory (BM) module on a FPGA. The term BM is used by the author to describe a building block in the hierarchical design of an equivalent HTM model [20]. Both these studies show that the performance of the FPGA based designs for associative memory models is dominated by the available chip area and the logic resources. Hammerstrom and Zaveri [22] analyzed the optimum use of such resources, and compared the performance vs. price trade-off for different architectures. They concluded that the mixed-signal CMOL design had the

best performance-to-price ratio. The authors also suggested that [22], "if in the future, nano devices/materials research provides robust solutions for implementing various analog functions using nanotechnology, this performance over price advantage is going to increase even further."

Other fully digital implementations of ANN AM integrated circuits can be seen in [14,29,41], which present various trade-offs between silicon area and computation time. 16 x 16 pattern storage and recognition networks are implemented using multi-chip modules. All authors were convinced that their proposed architectures would benefit in terms of more complex computations, if digital devices scaled down another 1,000 fold from then existing 3–μm CMOS fabrication technology. Additionally, the proposed designs have only one stable state. More than one stable state exponentially increases the component count within each building block.

A fair insight into the algorithmic implementation of high-level AM algorithm including learning is achieved from the work of [1]. The design employed analog amplifiers to act as 'neurons', five-bit registers as synapses, and noise amplifiers for the simulated annealing. The research highlighted several challenges:

- lack of an effective algorithm for learning in modular, hierarchical networks;

- necessity of modularity to manage connectivity;

- simplification of node design in addition to synaptic density;

- constraints, such as power dissipation and capacitive loading across the chips;

- at least 100 x 100 neurons interconnected by 1000 x 1000 synapse for the simplest of meaningful computation.

As Pao *et al.* [45] set the rules for high-level AM models, the above experiments stressed the need for optimizing low-level AM designs that could be hierarchically

integrated in densities not achievable even with the current 22nm nand-flash technologies. Additionally, Bailey *et al.* [4] assert the need for multiplexed interconnects for large scale ANN based AM system integration.

The implementation of building-blocks for high-level AMs memories is a challenging problem in artificial vision, image recognition, and other intelligent and adaptive computing areas. This challenge has previously been addressed in many different ways, for example by modeling artificial neural networks using traditional components such as resistors, capacitors, operational amplifiers, including voltage and current sources, as summarized above. However, the traditional approaches lack scalability. The other problem is that an AM building block unlearns as well if destructive input patterns are introduced.

1.3   Exploiting Memristors: Towards Nano-scale Designs

This thesis explores denser designs with novel nano-scale components that bypass the scalability hurdle with their inherently small form factors and with new properties. The device of choice for our investigation is a memristor.

Memristors were theoretically introduced by Leon O. Chua [9] as passive two terminal devices in which the resistance is a function of the magnitude, polarity, and the duration of the applied voltage, and hence, a passive element with memory. In 1976 Chua and Sung generalized memristors to a class of nonlinear dynamical systems or the memristive systems [10]. Memristive systems were theoretically shown to be perfectly non-linear resistors showing hysteresis at lower frequencies and reducing to linear resistors at higher frequencies. The non-linear characteristic of a memristor is exemplified in Figure 1.1.

Memristance was observed in nano-scale solid state devices by Strukov *et al.* [61]
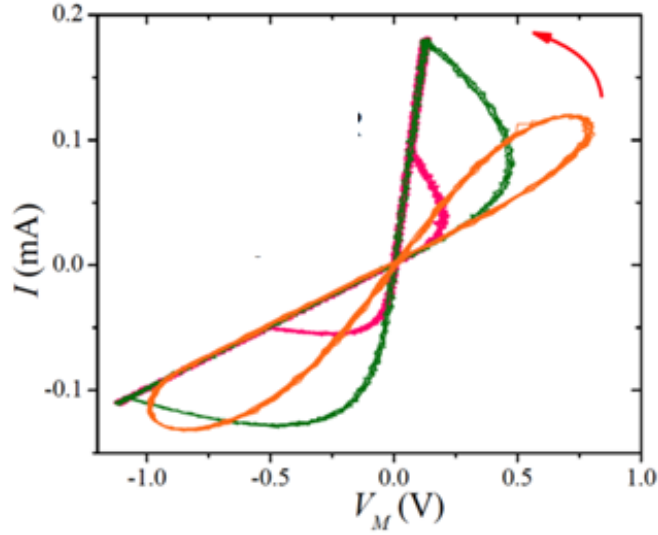
Figure 1.1: An example of the non-linear characteristic in the I–V plot of a memristor. Figure re-drawn from [11].

and it was experimentally confirmed that resistance in memristive systems spans a continuous range of value. Memristance was quickly observed in several nano-scale systems, for example, oxide-vacancy-based titanium dioxide electrodes reported by Williams [66], electrochemically controlled polymeric memristors reported by Erokhin and Fontana [17], and magnetic memristors as reported by Wang *et al.* [65].

Soon after their discovery, memristor circuits were shown to perform universal logic operations, like material implication [7] and useful arithmetic operation [38] quite accurately. Memristor crossbar latch memory or RRAM have been demonstrated by Pascal *et al.* [46] to achieve densities higher than DRAMs. However, a significant hurdle to realizing the potential of RRAM is the sneak path problem which occurs in larger passive arrays. Memristor-based circuits have been hand-designed to emulate biological responses like environment awareness in amoeba [48] and may find application in pattern recognition [55].

In AM and ANN paradigms, memristors have been explored solely as synapses

[59, 69], effectively working as switches between 'on' and 'off' resistance values. We hypothesize that memristor network AMs will be more area-efficient than the traditional AMs if we exploit their:

- non-linear property for spatial association, and their

- time-dependent property for temporal association.

Memristors, with their nano-scale form factors, continuous resistance range, non-linearity and time-dependency promise desirable circuits that can emulate low-level AM model. Discovered only in 2008, these novel devices have no established design methodology yet as in the case of CMOS technology, which has developed over the past six decades. In the absence of design experience, we chose automated circuit discovery, employing stochastic search and evolutionary optimization as a tool for exploiting memristors design space.

1.4   Genetic Programming: Exploring the Memristor Design Space

For this thesis, we hypothesize that associative memories exploiting the non-linear and time-dependent properties of memristors will be more area-efficient than associative memories built traditionally, e.g., by using neural networks. The results presented in the thesis confirm this hypothesis. In Chapter 2 we will present the detailed methodology for stochastic search for memristor-based AM designs using Genetic Programming (GP). We rely on GP as it is a proven technique for circuit evolution [34] and is simple to implement. And, since our framework is built upon ParadisEO API [39] in C++, the classes can easily be extended or ported from one system to the other. The optimum node size is decided within the framework. The only downside is the parameterized choices related to the algorithm (for example,

population/offspring size, mutation rates, replacement and reinsertion policies, fitness functions etc.,) is design-specific and can only be stochastically determined. This translates into a number of exploratory simulations for fine tuning each specific experiment. We use our framework to explore denser and robust associative memory designs using memristors.

1.5   Thesis Statement

The goal of this thesis is to develop a methodology to implement the functionality of AMs using memristor networks. We have achieved this by developing an evolutionary optimization framework that works in tandem with a circuit simulator to evolve closed circuits matching the desired target functionality.

The challenge of equivalent circuit designs emulating AMs has previously been addressed by using ANN models driven by biological inspiration. ANN models try to add learning at different hierarchical levels, which would ultimately also lead to intelligent and cognitive computing [45]. However, the traditional designs have many problems, such as a lack of scalability and susceptibility to noise.

Our approach is novel in that it will rely on novel circuit elements as opposed to classical neural networks. We present that associative memories using memristors with their non-linear time-dependent properties are more area-efficient than traditional AM designs. So far, memristors have been explored/exploited solely as a synapse in the neural nets, effectively working as on/off switches. Our approach is transformative because we evolve rather than hand-design memristor networks to enable the modular abstraction of associative memories. Our results show adding memristor to the set of active and passive components for neuromorphic designs leads to compact and simpler AM circuits.

Our research will be of interest to groups:

- modeling memristors: as we rely on our evolutionary framework rather than hand-designing circuits, the modeling teams can focus on closer-to-reality models for automatic discovery rather than simplified versions for a designer.

- working on intelligent computing models: as we can tune our framework to provide equivalent circuits for the desired functional blocks optimized for accuracy, size or noise tolerance.

Our thesis is a demonstration that memristor networks can implement AMs at nano-scales, which ANNs could achieve at micro-levels. However there are certain risks associated with the interpretation of results. For example, although the size of the circuit may be estimated using the 3nm x 3nm memristor fabrication benchmarks [67], the size advantage is derived from custom designing for different block level functionalities and relies on available spice memristor models for accuracy. Besides, at this stage, there is no simple way to describe the speed of the circuits. Also, at the moment, we do not have an efficient algorithm that can describe the block level functionality of AMs especially temporal AMs. Therefore, the evolved designs are customized and optimized for the specific functionalities chosen as representative of spatial and temporal AMs respectively.

To validate our framework, we performed initial experiments and successfully evolved the equivalent circuits for:

- low-pass filters that match [37] in evolution, convergence and final evolved circuit.

- Hodgkin-Huxley model for a potassium channel in a neuron with [11] as reference and improving upon their results.

Our key result is that we evolved efficient memristor-based networks that have the potential to replace conventional artificial neural networks used for associative memories.

1.6 Thesis Contribution

We now summarize the main contribution of this thesis:

1. We have developed a generic framework for designing analog memristor-based circuits. GP approach ordinarily has a tree type data structure and hence the mapping is generally presented [37] for single input and single output port circuits. This mapping was extended in order to apply GP for evolving multiple input/output ports AM circuits.

2. Since our framework is in C++, the classes can easily be extended or ported from one system to the other. The optimum node size is decided within the framework and has no hard bound limits.

3. The following 5 mutations were added by us to improve the quality of the circuits:

   (a) Branch mutation: Picks a random branch and replaces it by a newly generated branch.

   (b) Point mutation: Picks a non-terminal random node and mutates it into a new randomly chosen node.

   (c) Hoist mutation: Picks a random non-terminal node and makes the node and its sub-trees the main tree.

(d) Collapse mutation: Picks up a random non-terminal node and collapses its sub-trees.

(e) Expansion mutation: generates a new tree and replaces a randomly chosen terminal node in the original tree as a sub-tree.

4. In the absence of a solid design methodology, we show that automated circuit discovery is a promising tool for memristor-based circuits. Our results show that we can efficiently implement complex functions with few components.

5. Our key result is that we evolved efficient memristor-based networks that have the potential to replace the conventional artificial neural networks used for associative memories.

(a) We could evolve associative memories that can learn

- spatial correlation between inputs and

- temporal correlations within the inputs stream.

(b) We have explored the trade-off between the size and the accuracy of the circuits.

(c) We could evolve circuits that were robust against noise and variation on the input. This robustness was achieved with some cost of additional nodes in the network.

Our results show that we can efficiently implement complex functions with memristors leading to compact complex inferring machines. See chapter 6 for the interpretation of our results.

1.7 Thesis Organization

The rest of the dissertation is organized in the following manner. In Chapter 2, we present our evolutionary framework based on genetic programming to automate the design of analog circuits. In this chapter we focus on the methodology and the design choices of the framework itself. Additionally, section 2.3.5 serve as a user guide to running the program and designing new experiments. The frame work was validated by evolving two different circuits, namely 1) a low-pass filter, and 2) an analog circuit model for ion channels. The design for these experiments and results are presented in Chapter 3.

We extend the design exploration towards spatial associative memories in Chapter 4. In section 4.1, formalizing the spatial association problem in terms of our target function. We present evolutionary data and evolved designs for basic associative memory blocks in section 4.2. We have furthermore explored the trade-off between the size and the accuracy of the circuits by weighting both the aspects into the evaluated fitness. In section 4.3 We show that by varying the cost associated with the circuit size, the framework can be tuned to give more accurate results or lower noise levels vs. smaller designs. Finally in section 4.5, we introduce destructive noise on inputs and evolve designs that retain learning despite noise on input.

We delve into the temporal AM designs in Chapter 5. The representative task undertaken is that of context-recognition in a data stream and is described in section 5.1. The evolutionary statistics and design results for basic context-learning temporal designs are presented in section 5.2. We perform tests related to the frequency and voltage limits for the evolved design in section 5.3 and evolve variation tolerant designs in section 5.4.

The concluding remarks and directions of future research are presented in Chapter 6.

2

Methodology

2.1 Genetic Programming: History and Context

In the 1950's, terms like "Machine Intelligence" [64], "Artificial Intelligence" or AI [42], "Machine Learning" [56], all seeking human-like intelligence from behaviors exhibited by machines, were introduced. The vast domain of AI research in the next fifty years can broadly be categorized into a) Conventional AI [23] and b) *Computational Intelligence* (CI) [16, 49]. Conventional AI uses logical and formal knowledge to replicate human intelligence, while CI involves learning by successive trials and errors. In a modern context, machine learning is defined as "the study of computer algorithms that improve automatically through experience" [43], which is closer to CI. *Evolutionary computation* (EC) is a narrower field of research within CI that draws inspiration from nature. It has led to the development of *Evolutionary Algorithms* (EAs) that selectively breed a population to reach a desired end-point. The general term EA groups a set of four methods that actualize Darwinian principles of natural selection [3]. See Figure 2.1 for this overview.

Research and development of *Genetic Programming* (GP) was preceded by research in *Genetic Algorithms* (GA), *Evolutionary Strategies* [6] and *Evolutionary Programming* [15]. Since the method's first successful demonstration in 1992 [37], the GP approach has been shown to have applications in the design of both analog and digital circuits, control systems, robotics [34] and many more. The differentiating factor between GA and GP is the genome representation. GAs use strings of a fixed length to characterize a solution, whereas GPs can conform to multiple
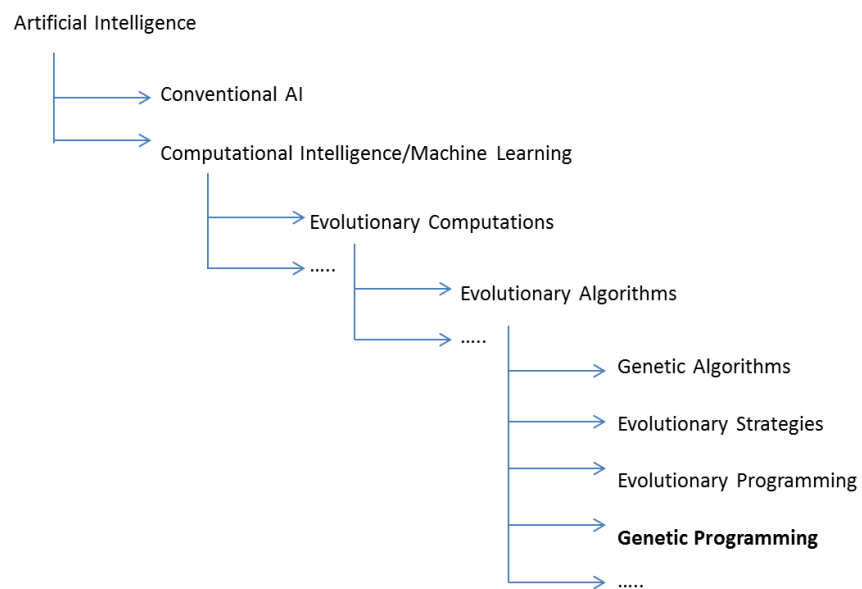
Figure 2.1: Placing Genetic Programming in the context of related research. Figure redrawn from [13].

representations of variable sizes. This is one of the strengths of GP, it can adapt to the problem complexity without any maximum node size restrictions. Effectively, GAs are efficient in parameter optimization given a fixed structure, whereas GP can explore both structure and parameter spaces. Koza presents GP as a superset of all EA methods and claims it to be theoretically possible "for a properly designed GP system to evolve any solution produced by other evolutionary algorithms" [34].

We chose GP over other methods for its versatility in terms of structure, scalability to solutions of any conceivable size, and availability of APIs to build upon well-tested and documented frameworks like the *ParadisEO framework* [39]. The next section will introduce the basic concepts of GP.

2.2 The Basics of GP

Genetic programming evolves a random population of competing solutions in order to match the desired functionality. More specifically, the population is transformed in each generation with the help of genetic operators. The objective is to minimize a specific criterion, also called the *fitness function*. The genetic operators include crossover, mutation and reproduction. GP typically runs on a tree-based data representation with ordered nodes. Other representations would include: linear structures [68] and graph structures [30]. We adopt the most common tree-based GP approach and will now introduce its primary elements.

2.2.1 Node assignment

A solution structure is built upon nodes of two types: functions and terminals. The function nodes act upon the inputs and map them to the outputs. The terminals

are useful in terminating a growing tree. For example, in circuit evolution, a typical set of function nodes may have *circuit-constructing components*, like resistors, capacitors etc. along with *circuit-processing functions*, like series and parallel. Creating a comprehensive pool of functions and terminals is an important preparatory step. Without a suitable set of nodes, the GP algorithm may not reach a solution or take a long time in doing so.

2.2.2 Genetic representation

In tree-based GP, all candidate solutions have functions and terminals randomly interconnected in an ordered tree structure. A GP tree comprises of a root node positioned at the top of the tree, while the leaves appear at the fringes and are called terminal nodes. Any subgroup of nodes is defined as a subtree. A tree with a terminal at its root node is defined as a *null-tree*. Executing the example tree from Figure 2.2 in depth-first fashion would result in terminal 4 added to multiplication of terminals u and 2, giving $y = 4 + 2u$ as an output. A GP tree is formally termed as an *individual*.

2.2.3 Population initialization

A GP run is initiated by creating a population. A population is comprised of a group of individuals (the randomly generated GP trees). Generating individuals by random interconnecting nodes involves two processes called *seeding* and *expansion*. *Seeding* can be done at random or hand crafted seeds from the results of previous runs may be used to create new individuals. There are three strategies for *expansion*: *grow*, *full*, and *ramped-half-and-half* [39]. The *grow* strategy randomly chooses between functions and terminals and allows each branch to grow
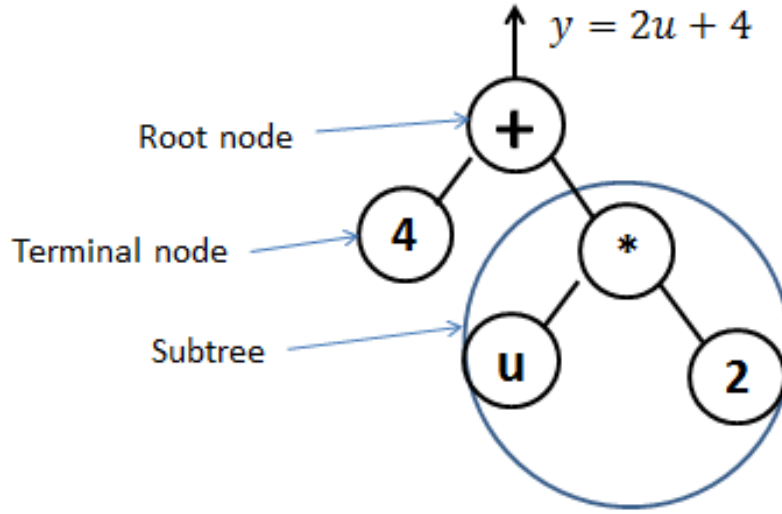
Figure 2.2: An example of a program tree illustrating how the root, terminals and subtrees are defined in tree-based GP. Depth-first execution of this tree would yield the output $y = 4 + 2u$.

until they find terminals. The *full* strategy chooses from the function nodes alone until the specified tree depth is attained and then uses terminal functions to restrict the growth. The third strategy, *ramped-half-and-half*, switches between the *full* and *grow* approach during initialization of a population. The most effective *seeding* technique and the best *expansion* strategy must both be experimentally determined.

2.2.4 Genetic operations

Once the population is initialized, the algorithm proceeds by varying the population with genetic operations. Three primary operations are: crossover, mutation and reproduction. For every new generation, a new population is created by applying these genetic operations to the parent population.
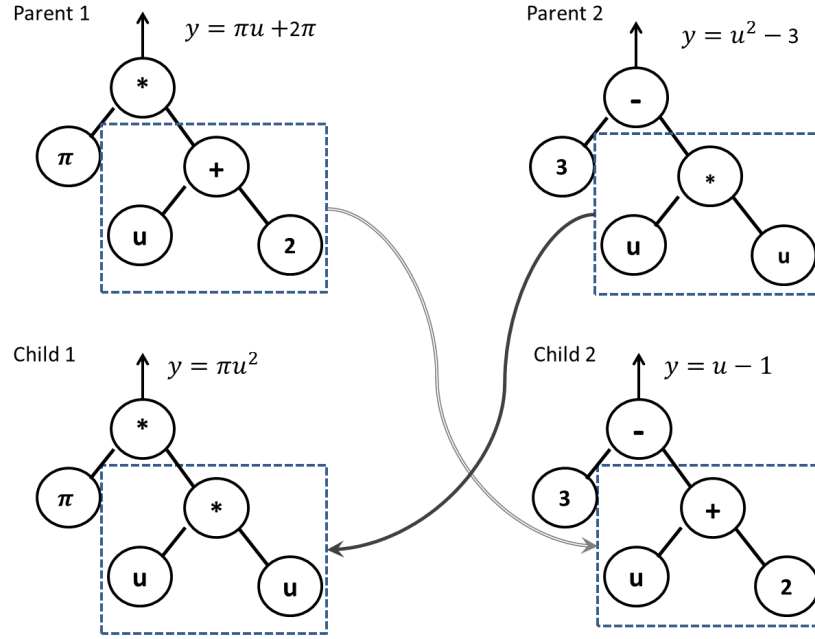
Figure 2.3: An example of crossover operation carried out on parents 1 and 2 to yield children 1 and 2. The crossover operator picks randomly chosen subtrees from parent 1 (at node +) and parent 2 (at node ∗) and switch them in child 1 and child 2.

- Crossover: The crossover operation swaps randomly chosen subtrees of two parents. The idea behind crossover is that useful building blocks exist within the population and crossover permits recombination of these blocks for a better solution. Figure 2.3 depicts a crossover example.

- Mutation: The mutation operator acts upon a single individual. There are a number of conceivable mutations in a tree acting upon either a node, a branch, or a subtree. For example a functional node could be mutated to a different node in the child changing the interpreted output as can be seen in Figure 2.4. Mutation is aimed at diversifying the population so the algorithm can avoid any local minima within the solution space.
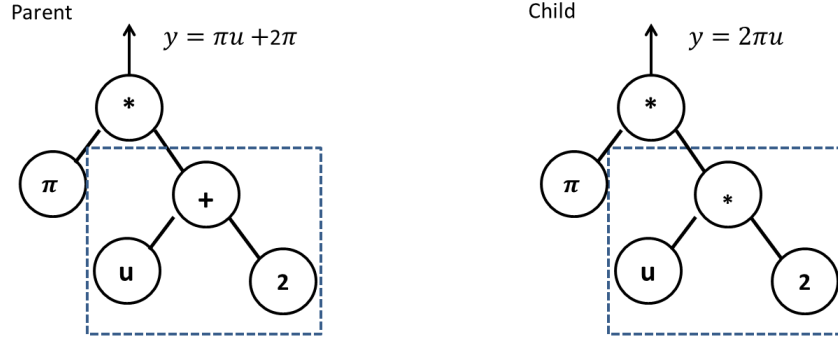
Figure 2.4: An example of mutation operation working on functional node '+' of the parent and transforming it to functional node '$*$' in the child thus changing the interpreted output from $y = \pi u + 2\pi$ to $y = 2\pi u$.

- Reproduction: The selected individual is cloned without any modification into the next generation, allowing good solutions to survive.

2.2.5 Fitness function

The fitness function determines how well an individual is able to solve the target problem. It varies greatly from one problem to another. For example, if the problem was to fit some data points on a curve, the fitness function could simply calculate the squared-error across all data points. The calculated squared-error will be assigned as *fitness value* for that individual. The assigned *fitness value* governs the selection of that individual for genetic operations and hence its survival in the next population. Within a GP cycle, the fitness function is responsible for execution and evaluation of each individual in a population for every generation. Fitness functions are problem-specific and could have multiple evaluation criteria.

2.2.6 Replacement strategy

Replacement is the step where GP creates a new generation by choosing individuals from both the parent population and the new child population created by using genetic operations. Replacement aims at allowing only the fittest individuals to survive while maintaining population diversity and keeping the population size constant. The two main approaches are: a) *generational* and b) *merge-and-reduce* [39]. With *generational* GP, the child population created by the genetic operators replaces the parent population completely. With *merge-and-reduce* GP, we first merge both parents and offspring and then reduce this combined population to the correct size. To choose the right replacement strategy, one should consider the trade-off between faster evolution and the risk of getting trapped in a local optima.

2.2.7 The GP algorithm

To summarize, the following preparatory steps have to be performed before starting a GP run:

- Define the functions and the terminal set.

- Define the fitness criterion.

- Define the parameters, such as the population size, the maximum tree depth, the probability for each genetic operation, tournament size, stopping criterion, etc.

Once equipped with the main GP elements, the basic learning algorithm, as shown in Figure 2.5, comprises of the following steps:

1. **Initialization:** Create an initial population.

2. **Evaluation:** Evaluate the entire population and assign each individual a fitness value.

3. **Replacement:** Add offsprings to the new generation.

4. **Stop Criteria:** Check the termination criterion (e.g. a minimal fitness value or a maximum number of generations).

5. **Selection:** Select individual(s) for the chosen genetic operation.

6. **Variation:** Apply genetic operators until the new generation is fully populated.

7. **Repeat steps 2-6** while the termination criterion is not satisfied.

2.3 Adapting GP for Automated Circuit Design

The flexibility of GP with regard to exploration of analog circuit designs is well acknowledged [37]. However, fine-tuning of the algorithm for performance and efficiency remains a problem-specific challenge. A large number of GP applications in circuit design are only a proof-of-concept that the GP approach is useful. Their use of GP is limited to rediscovering simple tasks with traditional components for which the optimal solutions are well-known [5, 35, 36]. We have outlined in Chapter 1 that this thesis focuses on exploring non-linear memristor networks that can solve complex tasks, like associative memories. In order to apply GP to this task, we established a mapping from tree structure to closed circuits. Koza *et al.* [34] discuss this mapping along with the preparatory steps for the circuit evolution. This section describes how we have implemented GP for non-linear analog circuit design.

Figure 2.5: The GP learning algorithm.

2.3.1   Circuit initialization

The initialization step transforms all the possible nodes into a vector. There are three types of possible nodes: the component nodes, the function nodes, and the terminal nodes. The component nodes are used to construct a circuit and have the following options: `R` for resistors, `L` for inductors, `C` for capacitors, `D` for diodes, `NMOS` and `PMOS` for MSOFETs, and `X` for memristors. The function nodes are used to modify the topology of the developing circuit. The possible function nodes are `SERIES`, `PARALLEL`, `TO_GND`, `FLIP` and `RANDOM`. Each function node has a well-defined objective. For example, the `SERIES` and `PARALLEL` functions result in series or parallel addition of components, respectively. The `FLIP` function reverses the polarity of a component. `END` is the only available terminal node and is used for terminating a growing tree.

2.3.2   Circuit representation

The *ParadisEO* framework [39] initiates as many *null-trees* with zero nodes as the specified population size. Nodes are randomly chosen from the initialization vector to fill the top node. Each node has a defined arity. Here the term arity refers to the number of child nodes. An arity-2 node (`SERIES` and `PARALLEL` in our case) have two ordered subtrees and arity-1 nodes have only one subtree. The growth of each individual tree continues until it hits a terminal node (`END` in our case). Each node is given the structure of an operation with an ID, a value, a type, a function, and its connections. The connections are filled during the evaluation phase. A *ngspice* executable netlist is generated by mapping an individual into a sub-circuit. This sub-circuit is added to an *embryo circuit*. The purpose of *embryo circuit* is to provide input signals, loads, and directives for *ngspice* [44] execution of

the individual. Figure 2.6 is an example of how we develop an *ngspice* executable circuit from a randomly generated tree.
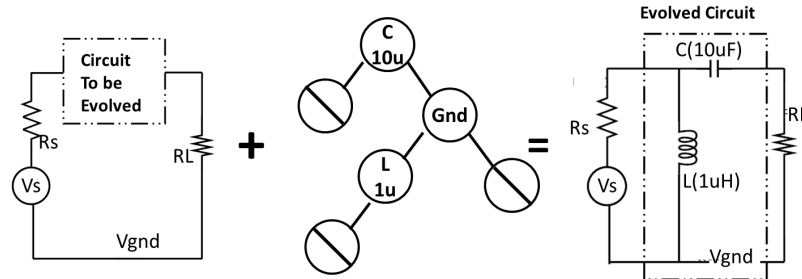
Figure 2.6: An example of an embryo circuit and a randomly generated tree, mapped together into a fully developed circuit.

In the example from Figure 2.6, the ports from the embryo circuit, say 1 as input port and 2 as output port, are passed to the root node of the tree, which is component node C-10μF. This component node adds the following line in the netlist:

C1 1 2 10u

Next, node C passes its connections (1 and 2) to function node TO_GND. TO_GND is defined as a ground-setting function node which receives ports (1, 2) from node C and passes, (1, 0) to its child node L-1μ. Similar to C-10μ, L-1μ is a component node and it adds the following line in the netlist:

L1 1 0 1u

L then passes (1,0) to its child node; the terminal in this case. The terminal node no longer extends the netlist. The tree from the example has now been mapped into a sub-circuit.

27

In order to map any randomly generated tree to a sub-circuit, the component nodes add a component and the function nodes, manipulate the incoming connections. All possible nodes introduced in section 2.3.1, have a well-defined functionality. The node functionalities are the basis for our GP implementation and a few of original function nodes from Koza's mapping [37] were further generalized using a more comprehensive set of function nodes defined by Bennett *et al.* in [5]. For example, Koza originally talks of `TO_GND` function node, which grounds the output port. Since there are always two connected nodes - parent node and child node, Bennett splits `TO_GND` into `TO_GND1` and TO_GND2 that operate on its parent and child node respectively and have an equal probability of either the input connection point or the output connection point being set to ground. Similar extensions were been implemented for `FLIP` and `RANDOM` functions.

2.3.3 Circuit evaluation

The purpose of the *fitness function* is three-fold. First, it translates an individual into a circuit netlist. Second, it runs the netlist through *ngspice*. And third, based on the optimization criteria, it assigns a *fitness value* to the individual.

During the netlist generation stage, the input connections are initialized and an empty netlist is created. This netlist is then populated by component nodes from the candidate solution tree. The populated netlist is then written into an *ngspice* executable format. The desired netlist format corresponding to a component node in the tree can be seen from the examples below:

```
Case R/L/C: R/L/C-ID connection-1 connection-2 value
e.g. 1: R1 1 2 1000
e.g. 2: L3 3 6 35000 or
```

```
e.g. 3: C3 2 4 1e-5 etc.

Case NMOS/PMOS: M-ID connection-1 to connection-4 mosfet-model Width Length
e.g. 1: M1 1 2 3 0 nm Width=5e-6 Length=10e-6
e.g. 2: M1 1 2 3 0 pm Width=10e-6 Length=10e-6

Case Diode: D-ID connection1 connection 2 Diode-model #
e.g.: D2 2 5 Diode-3

Case Memristor:  X-ID connection1 connection2 memristor-model
e.g.: X2 3 4 memristor
```

Once the sub-circuit netlist is generated, it is appended to an embryo circuit
as seen in Figure 2.6 and then passed to *ngspice* for circuit analysis. The analysis
from *ngspice* is read back by the *fitness function* and matched against a target
response for the squared-error calculation.

Finally, the squared-error and the number of components are weighted to give
the final *fitness value*. We normalize both squared-error and number of components
before computing their weighted average. The error component is normalized with
the highest possible error, which is determined experimentally from initial runs.
The size component of the fitness is normalized against the maximum number of
components allowed, i.e., 200 in our case. The final *fitness value* is then calculated
as:

$$fitness = (1 - w) \times squaredError + w \times netlistSize \qquad (2.1)$$

Here, w is a weight that determines the importance between the cost and the
squared error. A typical value of the weight used in our experiments is 0.5 or
50%. The objective of our GP implementation is set to minimize the fitness. This
implies that the closer an individual evaluates to zero, the closer it gets to reducing
both error and size, thus giving us the optimum solution.

2.3.4 Mutation operation

In *ParadisEO*, the evaluated population is ranked by the fitness. If the stopping criterion (i.e., the number of generations in our case) is not met, each individual undergoes a mutation with a probability defined in the parameters to evolve into an offspring. There are 13 possible mutations built into the framework. The following 8 mutation functions were based on [11, 34].

1. **Parameter change:** A components value is assigned as a new randomly chosen value.

2. **Series addition:** A new component is added in series configuration to the component. The type and value of the new component is randomly chosen.

3. **Parallel addition:** A new component is added in parallel configuration to the component. The type and value of the new component is randomly chosen.

4. **Component deletion:** A component is removed from the circuit.

5. **Type change:** A component's type is swapped to a different one randomly.

6. **Ground setting:** A component is connected to the ground.

7. **Adding a component randomly:** A new component bridges between two randomly chosen wires (not identical wire).

8. **Replacement mutation:** A component is replaced with a new component (possibly of the same type).

We added the following five mutations. The value of adding these mutations will be presented in section 2.5.2.

1. **Branch mutation:** Picks a random branch and replaces it by a newly generated branch.

2. **Point mutation:** Picks a non-terminal random node and mutates it into a new randomly chosen node.

3. **Hoist mutation:** Picks a random non-terminal node and makes that node and its sub-trees the main tree.

4. **Collapse mutation:** Picks up a random non-terminal node and collapses its sub-trees.

5. **Expansion mutation:** Generates a new tree and replaces a randomly chosen terminal node in the original tree as a sub-tree.

These, additional mutations were inspired by the mutation function in the genetic algorithm community, see e.g. [19]. Each of the five added mutations can be visualized from Figure 2.7.

2.3.5 GP implementation in C3EA

We now describe how GP elements described above are implemented in our application *Compact Complex Circuit Evolution Algorithm* or for short C3EA.

We maintain the traditional tree-based approach in C3EA for several reasons. First, with the tree-based GP, we can conceive any circuit design with a sufficient pool of component nodes and function nodes. Furthermore, the tree-based representation is supported widely by open-source GP frameworks. A different representation (e.g., linear structures [68]) would require a new GP framework built from scratch.

Mutation Operator	Parent Tree	Child Tree
Expansion Mutation: replace a terminal with a randomly created subtree		
Branch Mutation: replace a strongly typed subtree with a randomly created strongly typed subtree		
Point Mutation: replace a Node with a Node of the same arity and type		
Hoist Mutation: replace the individual with one of its strongly typed subtrees		
Collapse Subtree: replace a subtree with a randomly chosen terminal		

Figure 2.7: Examples of how the additional mutation operators transform the parent tree. Nodes in blue are inherited from the parent and nodes in red are transformed because of the respective mutation function.

C3EA allows multiple input and output ports in evolving circuits by leaving the ports empty during the tree generation and instead filling them during the tree's evaluation. During evaluation, the ports are identified and additional checks are performed that ensure all input and output ports are connected. C3EA can also evolve circuits with feedback loops by making use of the function node `RANDOM`.

In the following are some features of the GP elements as implemented in *C3EA*:

- Introduction of function nodes as possible nodes within a tree. This enables topological diversity starting from the very first initial population. See section 2.3.1

- An executable representation of analog circuits based on the traditional program tree. See section 2.3.2

- Three stage *ngspice* execution: a) the program tree is converted into a sub-circuit netlist, b) the sub-circuit is inserted into an embryo circuit of test signals and load components, and c) the component spice model along with combined netlist is then executed in *ngspice*. See section 2.3.3

- Definition of two types of fitness criterion: First, an accuracy-based fitness for guiding the evolution towards the target response. Second, a size-dependent cost for optimizing the number of evolved components for the fewest possible components. See section 2.3.3.

- Parameterization of replacement strategy: The choice of replacement strategy between a) generational and b) four different types of merge-and-reduce, was left as a run-time choice. See section 2.2.6

- Simultaneous search for circuit structure and parameter tuning. This is

achieved by having 13 different types of mutation functions, selected with a uniform probability. See section 2.3.4

- The actual circuit evaluation is implemented in the *ngspice* environment and GP itself is implemented as a C++ application, based on an open-source evolutionary computation framework, *ParadisEO* [39].

2.4 Architecture of C3EA

C3EA is built upon the meta-heuristics framework *ParadisEO* [39]. The structure of C3EA is depicted in Figure 2.8. Each part of this structure has been implemented as a C++ class and hence is portable and can be specialized independently. The main blocks that make C3EA are: the *evolution system* and the *evolver*.

An *evolution system* is a set of generic modules provided by *ParadisEO* for managing all functions and data structures needed by C3EA during the evolution. The *templates* allow customizing the system to a specific problem. The *EO algorithm* sets up the framework for executing a specific run. The *population* is structured into three layers: generation, population, and individual. *ParadisEO* enables execution and ranking of each individual of every population in every generation. The *continuation* classes permit the evolutionary run to continue until the stopping criteria is met. The *selection*, *transformation*, and *replacement* classes introduce the variation within a population across generations as per user-defined configuration.

The *evolver* is specific to C3EA and is a collection of operators that customize GP for analog circuit design. The following operators were implemented in C3EA: the *initialization* operator creates trees and individuals and the *mutation* operators modify the structure of individuals. The *evaluation* operator interfaces with

34

ngspice for circuit evaluation. The operator also provides each individual with a fitness value via a fitness function. The *statistical* operator plots the overall performance of C3EA for each successful run and the *result-plotting* operator plots the response of the best-evolved circuit.

Finally, since C3EA is designed as a generic analog circuit evolver, the configuration file permits run time customization of the experiments. This file allows users to choose:

- the experiment to run,

- the maximum number of generations,

- the population size,

- the initial tree depth,

- the maximum possible nodes,

- mutation rates,

- whether to allow or disallow different component nodes and finally,

- to choose between all possible replacement strategies.

2.4.1 The evolver algorithm

The C3EA *evolver*, as mentioned above, is a structured package of operators. The ordering and type of operators determine the working of the GP algorithm. The basic loop from Figure 2.5 is extended to represent the changes needed for analog circuit design and the result is shown in Figure 2.9. The genetic operators are chosen with roulette-wheel selection. The mutation operator gets a part of the

C3EA control

Evolution System (*)
- ParadisEO

ParadisEO Enablers (*):
• Templates
• EO algorithms
• Population
    • generation
    • evaluation
• Run continuation
• Selection
• Transformation
• Replacement

C3EA Evolver (†)
- My Classes

C3EA Operators (†):
• Initialization
• Circuit representation
• Fitness evaluation
• Mutation
• Statistical
• Result plotting

Ngspice

Gnuplot

Run-time Configuration (†)
- Which experiment?
- Maximum generation
- Population size
- Initial and maximum tree depth
- Mutation and elitism rates
- Allow which components
- Replacement strategy

Figure 2.8: The architecture of C3EA, a mix of elements inherited from the *ParadisEO* framework (*) and elements added or redefined by C3EA (†).

wheel (proportional to a user-specified mutation probability), the wheel spins, and depending on the spin outcome, either mutation or reproduction is selected. If mutation is selected, only one type from 13 possible mutations is applied to create the offspring. Again, each mutation is given a uniform distribution on a roulette wheel.
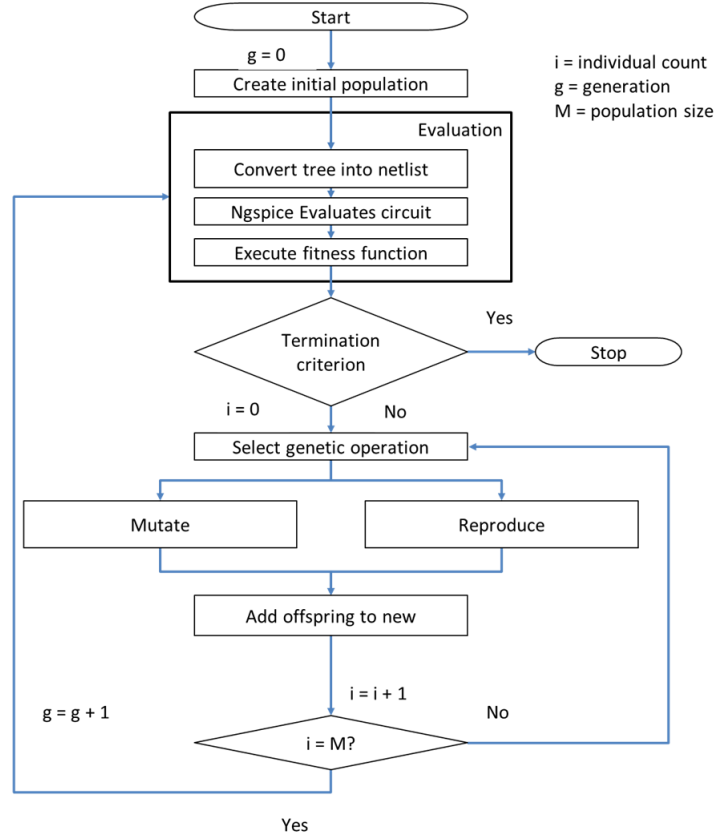
Figure 2.9: A flowchart of the GP algorithm as used by C3EA.

2.4.2 Features of C3EA

The following are some special features built into C3EA framework:

- *Configuration file:* The configuration file read at run-time can be used to choose the experiment, and set all the parameters. The file also defines the

components to use in circuit evolution. An example configuration file looks
like the following:

```
configuration.txt
----------------------------
which experiment:       2
nGenerations:           2000
population size:        100
offspring size:         2
MaxSize:                200
InitMaxDepth:           4
mutation rate:          0.8
ElitismRate:            0.1
EugenismRate:           0.0
sOffspringElitismRate:  0.0
sOffspringEugenismRate: 0.0
use R:                  1
use C:                  1
use L:                  0
use Diode:              0
use MOSFET:             0
use Memristor:          1
reduced mutation:       1
which replacement:      4
----------------------------
```

- *Statistical milestones:* At a rate specified by the user, a statistical operator
  writes the complete population together with the algorithm configuration
  and parameters to a single file, called a milestone. The advantage is that
  the run can be aborted at any moment, without any loss of data or the
  results achieved so far. The milestone (with optional changes) can be used to
  restart the GP run. A typical example is the modification of the termination
  criterion (e.g., the maximum number of generations) before the GP run is
  continued.

- *Population seed:* It is possible to provide GP with initial designs (or seed
  structures). During the initialization of a population, the seeded embryonic
  structures are copied and when the size of the seed is less than the population
  size, the remainder is created randomly with the initialization operators.

- *Post-processing:* The post-processing allows averaged plotting of fitness profiles from various GP runs for gathering statistics. It is possible to plot the *ngspice* outputs for the best evolved designs from each run.

2.5 Parameter Exploration

C3EA has been implemented in C++, hence the classes can easily be extended or ported from one system to the other. The optimum node size is decided within the framework and has no fixed limits. There are several parameterized choices related to the GP algorithm (for example, population/offspring size, mutation rates, replacement strategy etc.). These choices are problem-specific and evolver-specific, and can only be experimentally determined. In the following are the parameter determining experiments we performed in order to fine tune our C3EA framework.

2.5.1 Population size

Choosing the parameter *population size* is a trade-off between more variation vs. simulation time. The larger the population size, the more the variation and the sooner the algorithm may find a solution. But a large population also increases the simulation time per generation. We determined the optimum population size of 100 by running simulations and looking for an average number of generations taken to converge. The experiments were done for basic associative memory designs, details of which will be presented in Chapter 4. Figure 2.10 compares fitness vs. generations averaged over 15 experiments, each for population sizes 30, 100 and 200. It was observed that population sizes of 100 and 200 converged to the solution around generation $2,000$ while population size 30 converged around generation $3,200$. With the fastest convergence rate and reasonable simulation times,

a population size of 100 was chosen as a default.
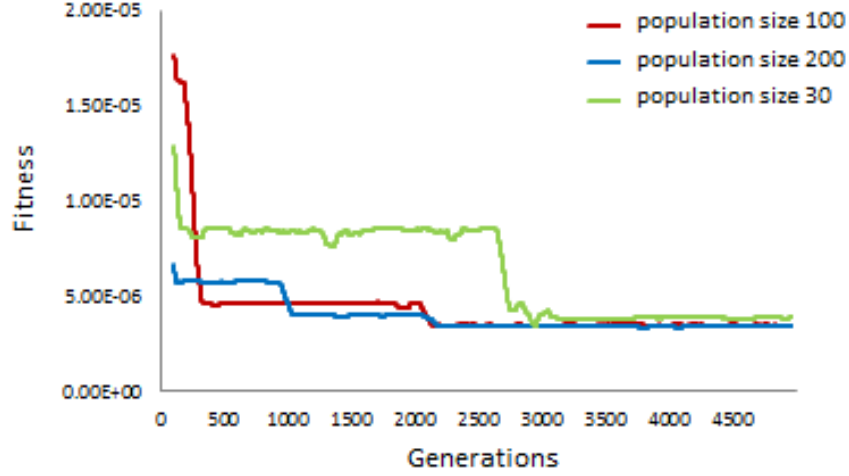
Figure 2.10: Comparison of different population sizes and their effect on evolution of basic associative memory. We observe that population sizes of 100 and 200 converge around generation 2,000, while a population size of 30 takes 1,200 more generations to converge to the best fitness value.

2.5.2 Mutation functions

While performing initial experiments, it was observed that the evolutionary run would often get stuck in a local optima and would not converge to the actual solution. The problem was narrowed down to the mutation functions. The set of 8 mutations suggested by Koza in [34] would add, delete or transform the component nodes only, and not the function nodes or the terminals, thus limiting the search space. The genetic algorithm community has several topology modifying functions (see [19]). We added 5 additional mutation functions that could act upon the tree itself. For details on the added mutation functions see section 2.3.4. In order to justify the value of these added mutations, we ran some basic associative memory design experiments comparing performance of all 13 mutations against that of

40

original 8. Figure 2.11 presents a comparison for the evolutionary data averaged over 10 experiments for: a) runs with original 8 mutations and b) runs with all 13 mutations. We observe that within the first $5,000$ generations, while the runs with original 8 mutation runs converges at a local optima, the runs with all 13 mutations settle at a lower-fitness global optima.
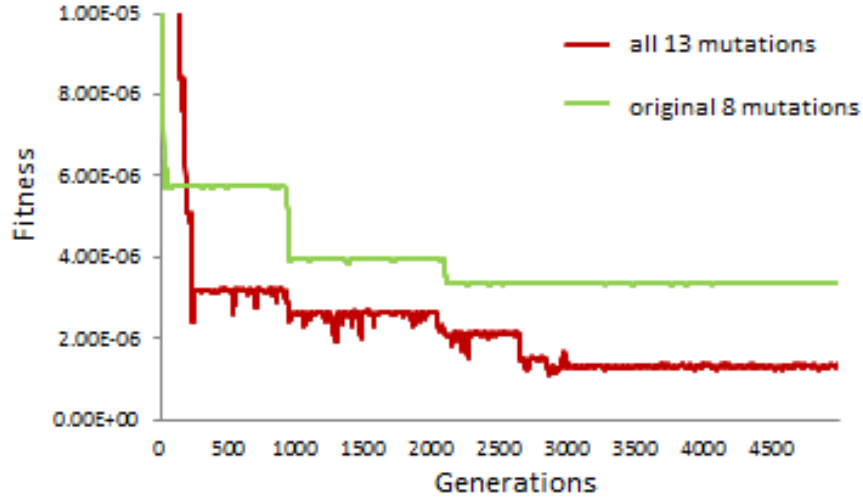
Figure 2.11: Justification for additional mutation functions. We compare evolution of basic associative memories using original 8 original mutations vs. using all 13 mutations. While runs using original 8 mutations converge at a local optima, the runs using all 13 mutations converge at the actual solution.

2.5.3 Mutation rates

Unlike population size, mutation rate does not affect the simulation time. Here, the parameter affects the search for the best solution itself. With higher rates, a higher percentage of the population gets mutated from one generation to the next. Since the best individuals are first selected for mutation, there is a probability that the evolution loses its good solution(s) and then gets stuck in some local minimum. The choice of a high-enough rate to ensure optimal solution is very

much determined by the severity of change any mutation has on the corresponding circuit. The reduced mutation sets resulted in minor variations, thus the higher mutation rates of 0.8 used by Cornforth *et al.* in [11] resulted in faster convergence. With the added five mutations, the suitable mutation rate had to be experimentally determined. We ran three different experiments of 10 runs with mutation rates of 0.3, 0.6, and 0.9. The fitness data presented in Figure 2.12 confirms that the lower mutation rate of 0.3 ensures smoother evolution and converges faster than the higher mutation rates of 0.6 and 0.9. Thus, a mutation rate of 0.3 is set as a default for all subsequent experiments.
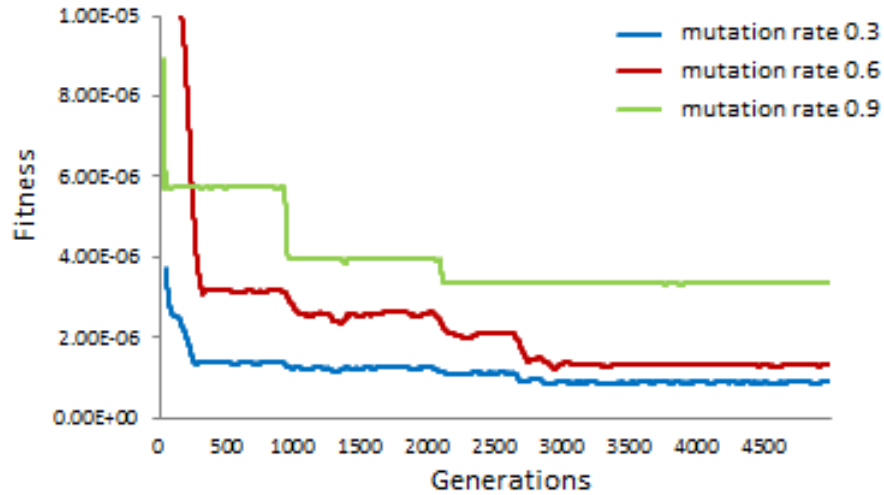
Figure 2.12: Comparison of different mutation rates for basic AM evolution. We observe that a lower mutation rate of 0.3 ensures smoother evolution and converges faster than higher mutation rates of 0.6 and 0.9.

2.5.4 Replacement strategy

Replacement aims at allowing only the fittest individuals to survive while maintaining population diversity and keeping the population size constant. The following

instances of replacement are available as *ParadisEO* classes:

1. **Generational:** All offspring replace all parents.

2. **Merge and Reduce:** It merges both populations of parents and offspring, and then reduces this big population to the right size. The other replacements are a subset of merge and reduce:

   (a) **Comma:** Selects the best offspring.

   (b) **Plus:** The best from offspring and parents become the next generation.

   (c) **Reduce Merge:** The parents are first reduced, and then merged with the offspring.

   (d) **Worse:** The worse parents are killed and replaced by all offsprings.

   (e) **Deterministic Tournament:** Parents to be killed are chosen by a (reverse) deterministic tournament.

   (f) **Survive and Die:** Allows strong elitism and eugenism in both the parent population and the offspring population.

Incidentally, no particular strategy is optimal for all experimented fitness functions. Hence, the strategy was left as a parameter in the configuration file. All experiments were run in parallel with all the strategies. The best solution from all the runs are presented as results in this thesis.

2.6   Discussion

C3EA *evolver* allows automated design of circuits with a high degree of flexibility, i.e., the search space for GP is quite large. But, because the computational power is limited, not every point in the search space can be explored. Thus, some degree

of creativity is required to let C3EA create novel non-linear designs. One method to simplify the design task is to apply a well-chosen subset of components. This is equivalent to giving GP a direction. For example, in the associative memory exploration with memristor networks, we disallowed some computationally intensive components like inductors, transistors, and diodes. This somewhat constrains GP, but results in more appropriate designs in terms of components, structure, and parameters.

Search of memristor designs with multiple parameters (e.g., on and off resistances and threshold voltages) led to circuits that could not be evaluated in *ngspice*. The design load is lessened by leaving the parameters constant as in the original memristor *ngspice* model and not proceeding with value mutation (see section 2.3.4) if a memristor is selected for mutation.

C3EA implements Koza's tree mapping, but this mapping allows for single input and single output ports. Here, Koza's mapping is extended to process multiple ports at both inputs and outputs. This is done by first, pre-defining the desired ports within the fitness function and then ensuring that all these ports are connected in any candidate circuit. If any of the desired ports is found to be unconnected, the evolved circuit is immediately given the worst fitness, thus preventing *ngspice* from slowing down the evolutionary run.

3

Validation

This section demonstrates the ability of C3EA to evolve analog circuits with the desired target functions. The evolved circuits used for validation include two circuits: a) a low-pass filter, and b) a Hodgkin-Huxley model for a potassium channel in a neuron.

3.1 Low-Pass Filter

This problem involves designing a low-pass filter having a one-input one-output circuit using capacitors and inductors that passes all frequencies below $1kHz$ and suppresses all frequencies above $2kHz$. Section 2.2.7 introduced the preparatory steps for evolving analog circuits using GP. We now present the preparatory steps and results from the low-pass filter experiments.

3.1.1 Embryo circuit

In the automated process for low-pass filter circuit design, an electrical circuit is created by combining a fixed embryo circuit with a randomly generated tree (section 2.3.2). The tree contains various component nodes and function nodes (described in section 3.1.2) that are mapped into a sub-circuit. Each tree in the population creates one candidate circuit. The evaluation process uses the program tree to convert an embryo circuit into a *ngspice* executable candidate circuit, and the specific embryo used depends on the number of inputs and outputs.

Figure 3.1 shows a one-input one-output embryo circuit in which VSOURCE

is the alternating current input signal that drives the circuit. There is a fixed $1k\Omega$ load resistor RLOAD and a fixed $1k\Omega$ source resistor RSOURCE in the embryo. In addition to the fixed components, we have defined two fixed nodes: 1 and 2. All the randomly generated trees are mapped into a sub-circuit and attached to these fixed nodes. Node 2 is also the probe point for the output signal.
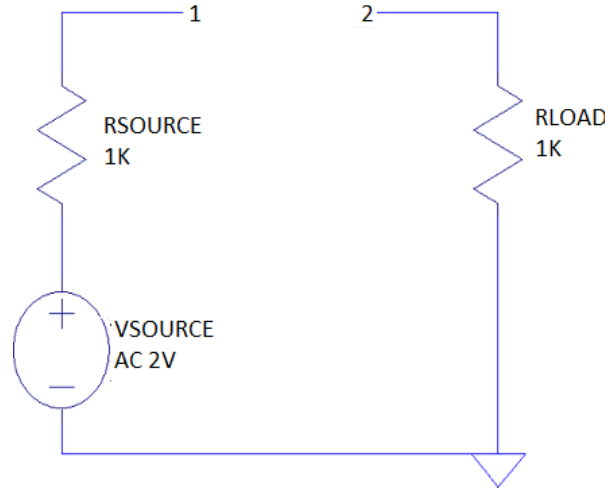
Figure 3.1: A one-input one-output embryo circuit used for the low-pass filter circuit. The input appears at VSOURCE, and the output is probed at node 2. The mapped sub-circuit from a tree is added between connection points 1 and 2. RSOURCE and RLOAD are $1k\Omega$ resistors.

3.1.2 Component, function, and terminal nodes

The set of component nodes for a target problem dictates the type of electrical components that may be used to construct the circuit. Resistors (R), capacitors (C), and inductors (L) were used for evolving this low-pass filter task. The component nodes insert components into a developing sub-circuit and establishes its numerical value randomly from a predefined range of values presented in Table 3.1.

The function nodes manipulate the connections of the developing sub-circuit and hence modifies the circuit topology. The function nodes used in the evolution of

Table 3.1: Components and parameter ranges used in low-pass filter evolution.

Component	Parameter range
Resistor	$\texttt{R} = 1 - 1 \times 10^9 k\Omega$
Capacitor	$\texttt{C} = 1 - 1 \times 10^9 fF$
Inductor	$\texttt{L} = 1 - 1 \times 10^9 kH$

low-pass filter circuits are: `SERIES`, `PARALLEL`, `FLIP`, `TO_GND`, `RANDOM`. The function nodes `SERIES` and `PARALLEL` allow serial and parallel addition of their two sub-trees in the candidate circuit. The function node `FLIP` performs polarity-reversal and attaches the positive end of its child node to its negative end. The function node `TO_GND` creates a connection to ground for its child node. The function node `RANDOM` connects distant parts of a circuit by randomly selecting one connection for its child node. The zero-argument terminal node `END` terminates the growing branch of a tree, thereby ending that particular circuit development path.

3.1.3 Fitness measure

The target low-pass filter has a pass-band below $1kHz$ and a stop-band above $2kHz$. The circuit is driven by an incoming AC voltage with a $2V$ amplitude. With $1k\Omega$ RSOURCE and RLOAD in the embryo circuit, the incoming $2V$ signal is divided in half. Thus, a voltage in the pass-band of exactly $1V$ and a voltage in the stop-band of exactly $0V$ are regarded as ideal. A voltage in the pass-band of between $970mV$ and $1V$ and a voltage in the stop-band of between $0V$ and $30mV$ are regarded as acceptable. The voltage at connection point 2 is measured in the frequency domain. The software *ngspice* performs an AC small signal analysis and reports the circuit behavior for frequencies chosen over two decades (between $100Hz$ and $100kHz$). Each decade is divided into 10 parts (using a logarithmic scale), so there are 20 probe frequencies for this problem. Figure 3.2 presents

the ideal low-pass filter frequency-domain response along with the 20 probe points used in the fitness evaluation.
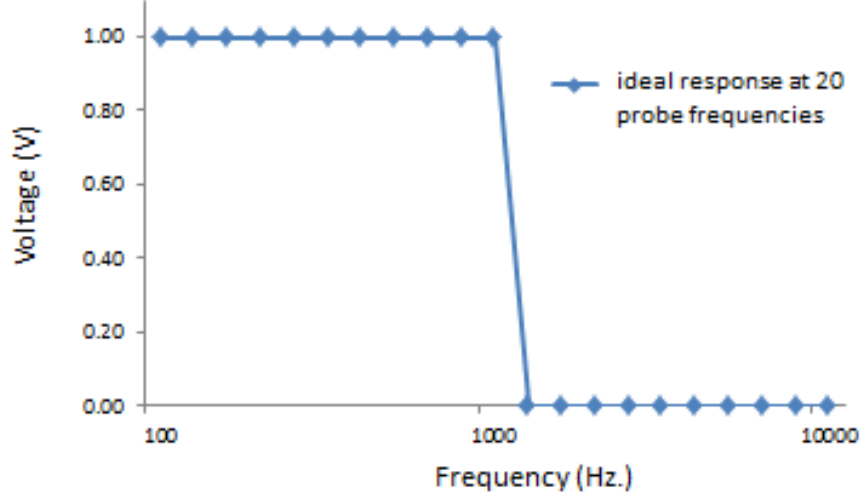
Figure 3.2: The ideal $1kHz$ low-pass filter frequency-domain response and 20 probe frequencies ranging from $100Hz$ to $10kHz$ used in fitness evaluation for the candidate solution circuits.

The fitness for each candidate circuit is measured in terms of the sum of squared-error between the actual value of the voltage at connection point 2 (see Figure 3.1) and the target value for voltage. A fitness of zero represents an ideal low-pass filter. Since the maximum error at each probe frequency can be 1 V. Hence, the maximum sum of squared-error is determined as 20. This value is used to normalize the fitness and to assign a *fitness value* as:

$$fitness_{normalized} = \frac{\sum_{i=1}^{20} (Error_i)^2}{20} \qquad (3.1)$$

Here, i is the data point number, and $Error_i$ is the difference between the actual voltage at the probe point and the target voltage for the frequency of $100 \times i$ Hz. We divide the sum of the squared-error by 20 in order to normalize the fitness

48

against maximum error.

3.1.4 Control parameters

We chose a population size of 100 for all tasks. For this problem, the 90% probability of the mutation operation on each generation were the same as those used by Koza *et al.* [36]. This being the first experiment with C3EA, we used Koza's original eight mutations as described in section 2.3.4 and the replacement strategy was left to the default (*generational replacement*), where all off-springs replace all the parents. We did not try to optimize the control parameters for the low-pass filter evolution. Each GP run was terminated at $2,000$ generations, and the best individual over five runs was presented as a result.

3.1.5 Results for low-pass filter

The best-evolved circuit (Figure 3.3) has two capacitors and one inductor. This circuit has a sum of squared-error of $0.1V$ and a normalized fitness value of 5×10^{-3} across the 20 probe frequency points. This circuit has a recognizable "bridged Π" arrangement. The "Π" consists of capacitors C1 and C2 and the inductor L1. The frequency domain behavior of this best-evolved design is 99.5% similar to the ideal response.

Figure 3.4 shows the frequency-response of this best-evolved circuit against the ideal response. All five GP runs converged to the same circuit within the first 700 generations.

In Figure 3.5, we present the fitness vs. generation averaged over the five GP runs. The error bar gives the standard deviation on best, average, and the worst fitness every 100 generations. In all five GP runs, the best fitness was observed
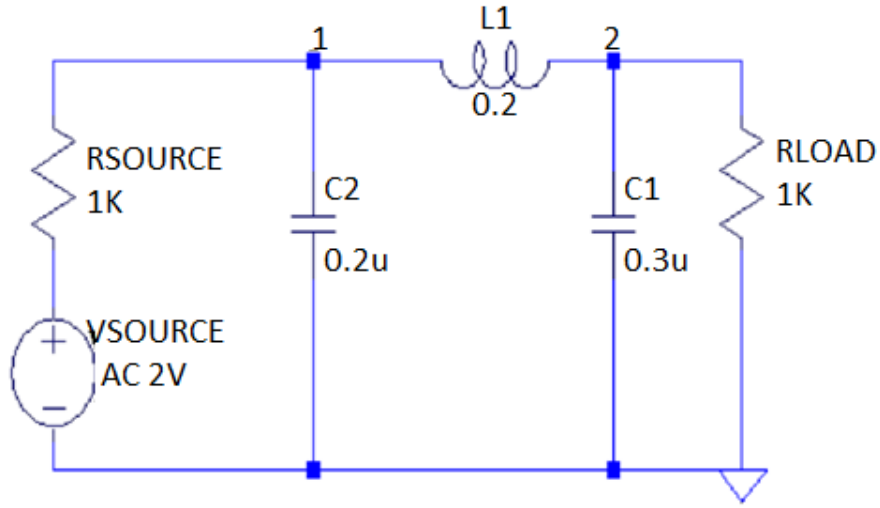
Figure 3.3: The best-evolved low-pass filter circuit with two capacitors and one inductor. All five GP runs converged to the same circuit as the optimum solution.
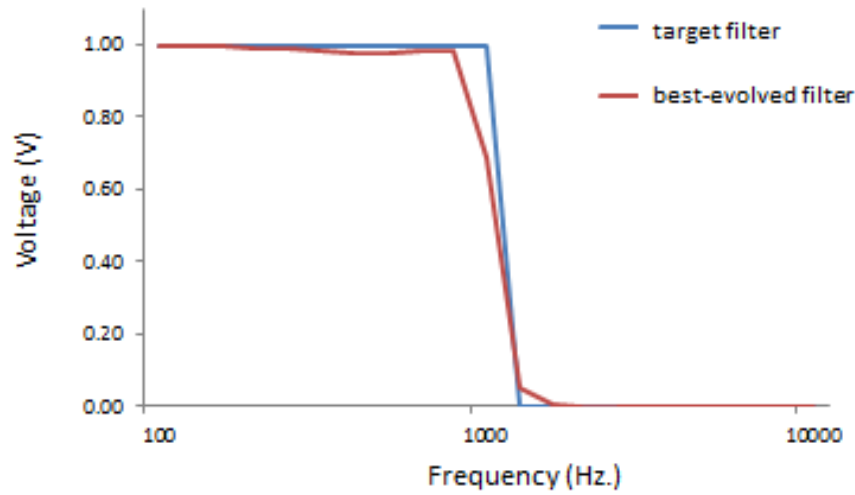
Figure 3.4: Frequency response of the best-evolved circuit compared with the ideal low-pass filter response. Here the average error for each of the 20 frequency probe points is $5mV$, which implies the circuit is 99.5% similar to the ideal behavior.

to improve over successive generations and converge to a solution. For each run, satisfactory results were generated between generation 500 and 700. Many of the random initial circuits and many that are created by the mutation operation in subsequent generations cannot be simulated by *ngspice*. These circuits receive a high penalty and a *normalized fitness value* of 1 and become the worst-of-generation program for that generation. In the fitness plot we observe that the initial worst fitness value remains constantly at 1. This implies that some initial circuits were not able to be simulated by *ngspice*. But, from the quick drop in the average fitness value, we can conclude that most circuits were simulatable after only a few generations. This demonstrates that the evolutionary selection process creates offsprings from fitter parents that are *ngspice*-simulatable.
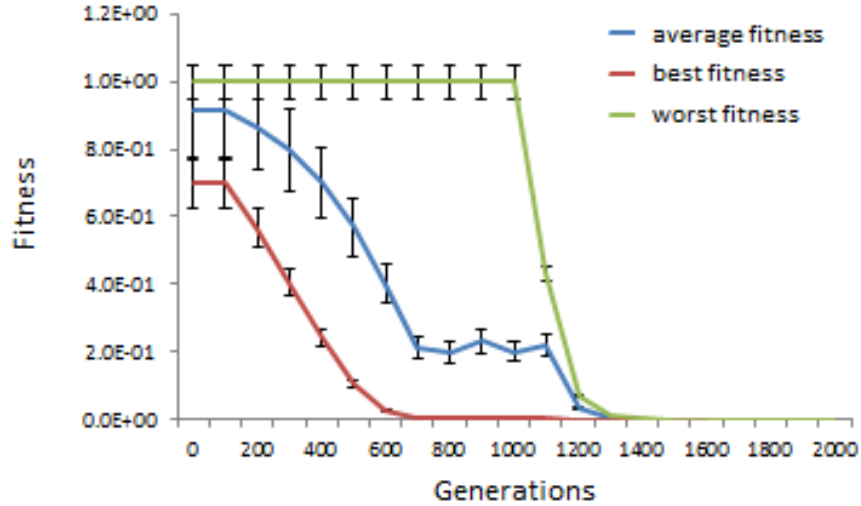
Figure 3.5: Fitness averaged over five GP runs for evolving low-pass filter. Here the error-bars denote the standard deviation over the five runs. Both the standard deviation and the fitness value decrease as the evolution progresses toward the solution.

3.2 Hodgkin-Huxley Neuron Model

The Hodgkin-Huxley model [25] in computational neuroscience is a mathematical description of the action potential across the membrane layer in a neuron. Cornforth *et al.* [11] applied a GP technique of analog circuit evolution to the task of creating an equivalent circuit for the Hodgkin-Huxley ion-channel models. To validate our framework, we used C3EA to evolve the Hodgkin-Huxley potassium-ion-channel model and compared our results with those presented in [11].

3.2.1 Embryo circuit

We used the same embryo circuit as Cornforth *et al.* [11]. The embryo circuit as shown in Figure 3.6 is a model for the cell membrane without any ion-channels. The stimulus current of $1nA$ is presented for $1ms$ to drive the potassium-ion-channel sub-circuit. The voltage source V_{el} ($70mV$) and load resistor R_{load} ($1G\Omega$) combine to represent the intrinsic driving force on ions. V_{el} and R_{load} model the leak potential and the resistance to ion-flow across the membrane. A capacitor C_{mem} ($1pF$) forms the membrane capacitance. A voltage source V_{mem} ($70mV$) emulates the membrane potential and the connection points labeled 1 and 2 are used as connection points for the evolved sub-circuits. The GP task is to find a sub-circuit such that the entire circuit at the probe point 2 reproduces the behavior of the Hodgkin-Huxley potassium-ion-channel. We will describe the behavior of the Hodgkin-Huxley model in section 3.2.3.

3.2.2 Component, function, and terminal nodes

In addition to the component nodes presented in section 3.1.2, we added diode nodes, and p-type and n-type MOSFET nodes. These components were added
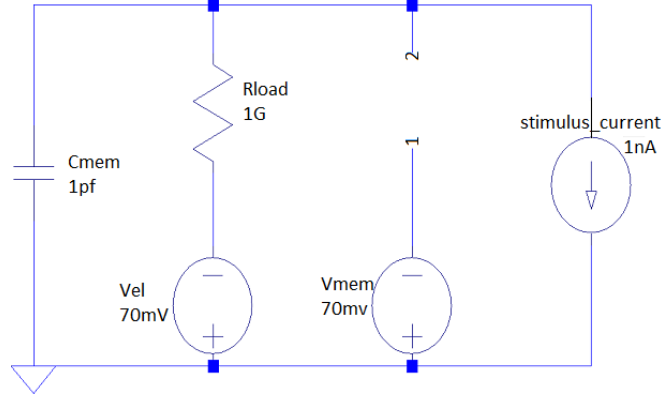
Figure 3.6: A one-input one-output embryo circuit used for the Hodgkin-Huxley potassium-ion-channel circuit. The stimulus current of $1nA$ destabilizes the system. The V_{el} and R_{load} combination, drive the ions through the membrane sub-circuit to be evolved between connection points 1 and 2. The voltage source V_{mem} ($70mV$) represents the membrane potential and capacitor C_{mem} represents the membrane capacitance. The output is probed at connection point 2.

to match the component pool used by Cornforth *et al.* [11]. The diode-creating component node `D` inserts diode into a developing sub-circuit. There are also six MOSFET-creating component nodes: `NMOS1, NMOS2, NMOS3, PMOS1, PMOS2,` and `PMOS3`. These MOSFET-creating component nodes insert a transistor into the developing sub-circuit. Depending on whether the inserted transistor is p-type or n-type, the source and the body terminals are connected to the positive or the negative power supply. As for the gate and the drain terminals, one is randomly chosen and connected to the parent node and the other to the child node. There are only six such functions in this family because the other possibilities are not electronically reasonable. See Table 3.2 for the parameter range of these added component nodes.

We used an extended set of function nodes for the Hodgkin-Huxley potassium-ion-channel model. This circuit set was adopted from Bennett *et al.* [5], and includes the following function nodes: `SERIES, PARALLEL, FLIP1, FLIP2, TO_GND1,`

Table 3.2: Additional components and their parameter ranges used in Hodgkin-Huxley model evolution.

Component	Parameter range
Diode	D, $Model = Diode1 - Diode25$
P-type mosfet	M, $length = 10\mu m, width = [5, 10, 20]\mu m$
N-type mosfet	M, $length = 10\mu m, width = [5, 10, 20]\mu m$

TO_GND2, RANDOM1, and RANDOM2. The function nodes SERIES and PARALLEL are the same as in the low-pass filter experiments, i.e., they allow serial and parallel addition of components. The original function node FLIP was split into: a) FLIP1 that performs polarity-reversal on its parent node, and b) FLIP2 that performs polarity reversal on its child node. Similarly, the original function TO_GND was split into: a) TO_GND1 that sets its parent node to ground, and b) TO_GND2 sets a connection to ground for its child node. The function node RANDOM that connects distant parts of the circuit was again split into: a) RANDOM1 acting on the parent node, and b) RANDOM2 acting on the child node. END is used for terminating the growing branch of a tree.

3.2.3 Fitness measure

We obtained the target response for the Hodgkin-Huxley potassium-ion-channel using the *HHsim* simulation environment [63]. The ideal membrane voltage response shown in Figure 3.7 is obtained using a step stimulus current of $1nA$ applied for $1ms$.

To evaluate the fitness of a candidate equivalent circuit, we stimulate it with a step current of $1nA$ in *ngspice* and compare the resulting membrane voltage at connection point 2 with the target voltage obtained from *HHsim* simulated membrane voltage. The *ngspice* simulation data is recorded at $0.1ms$ resolution for
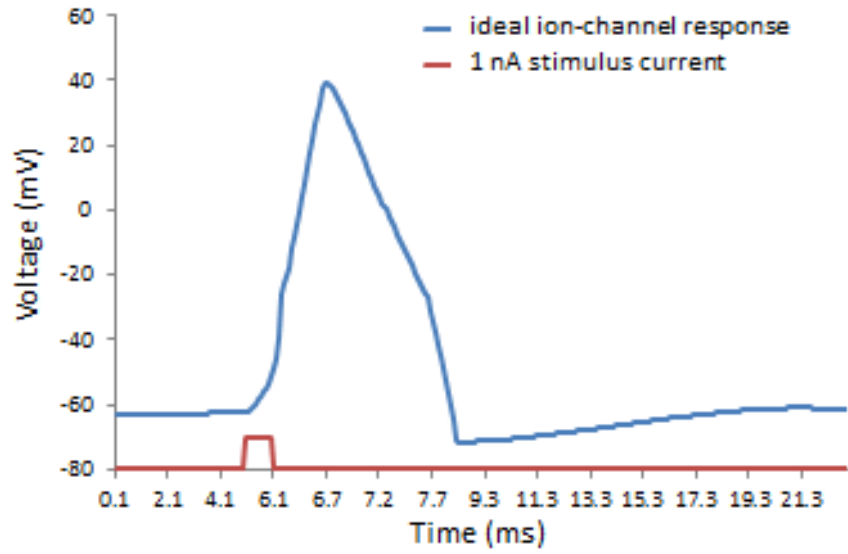
Figure 3.7: The ideal Hodgkin-Huxley potassium-ion-channel response to $1nA$ stimulus current presented between the 5 ms and 6 ms time-steps. This ideal response has been extracted from the *HHsim* simulation environment. There are 201 data points for the fitness evaluation of the candidate circuits. The data points are sampled every $0.1ms$ between $0ms$ and $23ms$.

$23ms$ and each of those 231 data points are compared with the corresponding data points from the target *HHsim* data. The *fitness value* is then computed from the sum of the squared-error at each data point weighted against the size of the evolved sub-circuit (see section 2.3.3). The squared-error component is normalized with the highest possible error, which was determined experimentally from initial runs to be 10^8. The size component of the fitness is normalized against the maximum number of components allowed, i.e., 200 in our case. A reasonable value of the weight w determined for this experiment is 0.05%. This value was determined by first running the experiments for accuracy alone and determining the acceptable range of the normalized squared-error. We then adjusted the weight such that the circuit size component dominates the evolution only when the squared-error was minimized to an acceptable level of 2×10^{-5}. The final *fitness value* is then calculated as:

$$fitness_{normalized} = 99.5\% \times \frac{\sum_{i=1}^{231} (Error_i)^2}{10^8} + 0.05\% \times \frac{size_{netlist}}{200} \qquad (3.2)$$

Here, i is the data point iteration number, $Error_i$ is the difference between the actual voltage at the probe point and the target voltage for i^{th} sample point every $0.1ms$. We divide the sum of the squared-error by 10^8 in order to normalize the squared-error against the maximum error. The term $\frac{size_{netlist}}{200}$ is the normalized cost associated with the size of the circuit. Then the final $fitness_{normalized}$ is the weighted average of the normalized squared-error and the normalized size cost.

3.2.4 Control parameters

We chose the population size to be 100. For this problem, the 80% probability of the mutation operation on each generation were the same as those used by Corn-forth *et al.* [11]. During the initial few runs with Koza's original eight mutation functions, the GP runs would get stuck in some local minima. We solved this problem by adding five new mutation functions as described in section 2.3.4. We tried all five replacement strategies from section 2.5.4 in different GP runs. Each GP run was terminated at 5,000 generations and the best individual from five runs, each with a different replacement strategy, is presented as a result.

3.2.5 Results for the Hodgkin-Huxley model

C3EA evolved compact circuits mimicking the behavior of an idealized Hodgkin-Huxley model for the potassium-ion-channel in a neuron. Figure 3.8 shows a circuit that evolved with four components that include: a p-type MOSFET, an inductor, a capacitor, and a diode. With four components and a weight of 0.05%, the cost of the evolved circuit size evaluates to 2×10^{-5}.

Figure 3.9 shows the step response of the best-evolved equivalent circuit from Figure 3.8 comparing it against the idealized model response from the *HHsim* simulator. For this circuit, the normalized squared-error over 231 data points was 7×10^{-5}. The final assigned normalized *fitness value*, calculated from equation 3.2, was 4.5×10^{-5}

Figure 3.10 shows the best, average and the worst fitness averaged over five GP runs. The fitness data was noisy on the worst and the average fitness plots because each of the five runs had a different replacement strategy and hence quite different evolutionary dynamics. The best fitness of each run converged to final
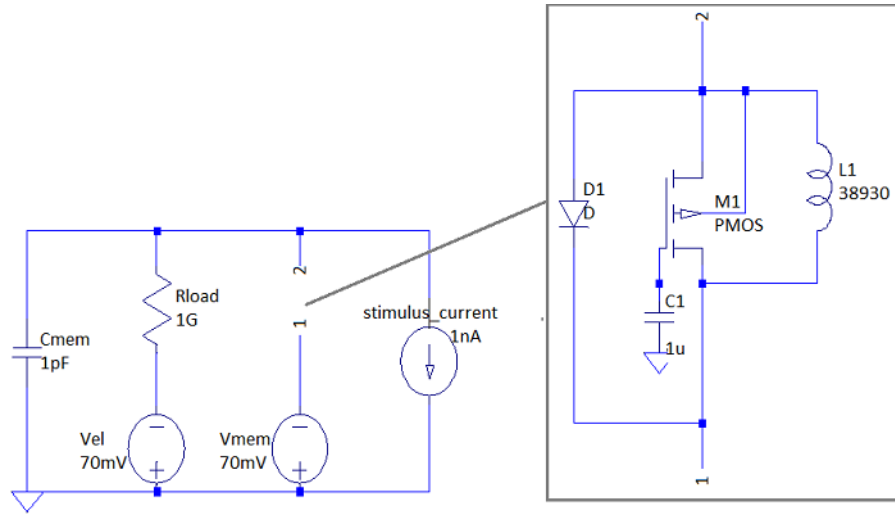
Figure 3.8: The best-evolved equivalent circuit for the Hodgkin-Huxley potassium-ion-channel model. It comprised of four components: a capacitor, an inductor, a p-type MOSFET, and a diode. Three of the five GP runs converged to the same circuit as the optimum solution. The other two had additional redundant component.
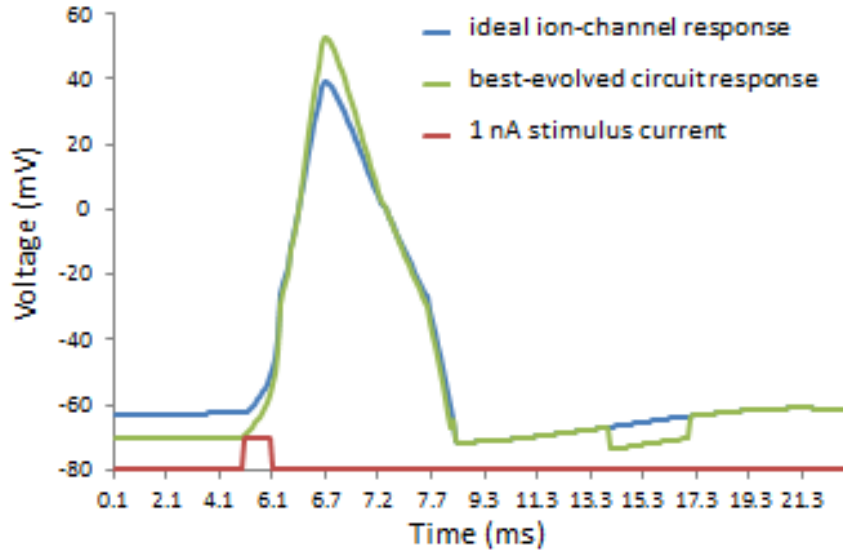
Figure 3.9: The transient response of the best-evolved circuit compared with the ideal Hodgkin-Huxley potassium-ion-channel response. Here the average error for each of the 231 data points is $0.5mV$. The sudden drop in the voltage between $14ms$ and $17ms$ is due to the discharging of the capacitor when the p-type MOSFET turns temporarily on during the same time-frame.

58

fitness values within the first $1,000$ evolutionary generations. In Figure 3.11, we plot the best fitness evolution alone along with error-bars every 200 generations. The error-bars represent the standard deviation among the five GP runs. The Hodgkin-Huxley task was more complex than the low-pass filter problem. We added the four-terminal transistors and diodes to the set of component nodes. We also experimented with adding cost to the size of the evolved-circuit in fitness assignment. Furthermore, we added new mutation functions in order to avoid being stuck in local minima. Yet, with some fine-tuning, C3EA could reproducibly evolve an acceptable solution for this problem.
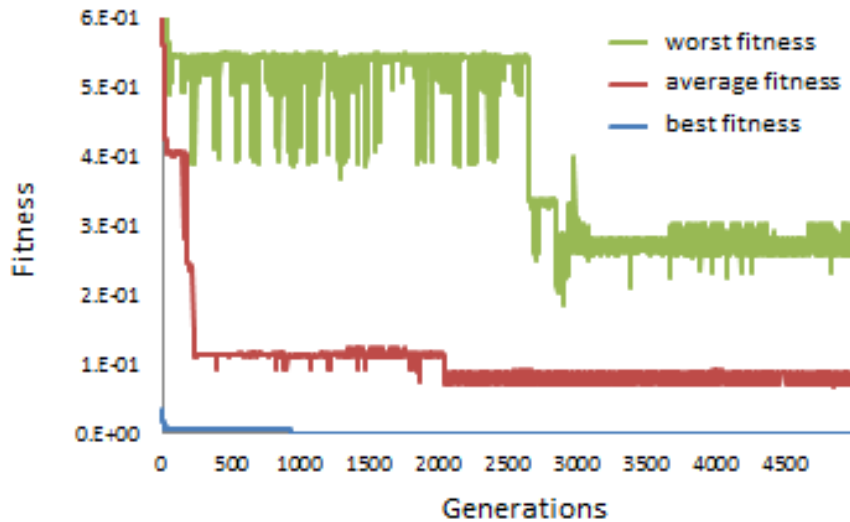
Figure 3.10: The fitness averaged over five GP runs for evolving the Hodgkin-Huxley model. Here, the worst and the average fitness plots are noisy because the plots presented are averaged over five runs each having a different replacement strategy, and hence the evolution would vary a lot from one run to the other.
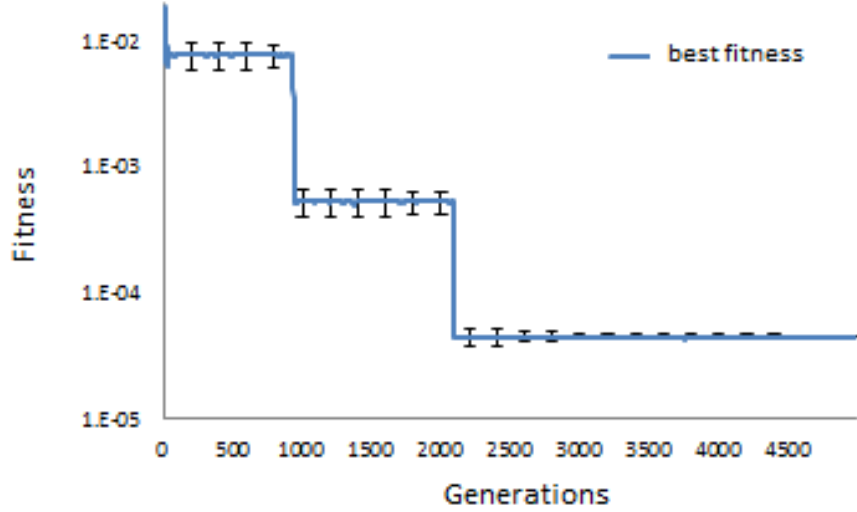
Figure 3.11: The best-evolved fitness averaged over five GP runs for evolving the Hodgkin-Huxley model. Here the error-bars represent the standard deviation between the runs. Both the fitness value and the error decrease as the evolution progresses and finally converge to a solution in less than a $1,000$ generations.

3.3 Discussion

We confirmed that C3EA can be used to automatically construct equivalent analog circuits for a given problem. We tested this idea with two experiments: a) low-pass filters, and b) an equivalent circuit for the Hodgkin-Huxley potassium-ion-channels. We confirmed that the evolved circuits agree with the target system's response to a large degree. Since the next experiments were related to exploring the memristor networks, a new component node for memristors (X) was added to the existing component node pool. However, memristors were not found useful in the low-pass filter and the Hodgkin-Huxley experiments. While using memristors, we observed that the memristors were either rejected by the selection process or they only replaced some redundant resistors.

60

4

Results: Evolving Spatial Associative Memories

4.1 The Spatial Association Problem

With our C3EA evolutionary framework validated (see chapter 3), we next deal with evolving spatial associative memories (AM). The spatial AM evolution is aimed at emulating Pavlov's classical conditioning experiments [47]. The evolved designs are compared against the results in [69] since they have solved the same classical conditioning problem with artificial neural networks (ANNs) with memristors as synapses.

4.2 Experiment 1: Basic spatial AM with sinusoidal input signals

An AM has the ability to associate different memories to specific events. Such memories form an integral part of cognition in most life forms, including humans [2]. This ability allows the brain to react or adapt to external stimuli based on past experiences. The famous Pavlov experiments [47] are a good example of AM: Pavlov observed that if a particular stimulus in the dog's surroundings was present when the dog was presented with meat powder, this stimulus would become associated with food and cause salivation on its own. Application areas for AMs are numerous: artificial vision, speech recognition, artificial intelligence [32], and other intelligent and adaptive computing areas.

Implementing dense and robust AM design is a challenging problem. This challenge has previously been addressed by modeling ANNs using two layers of neurons interconnected by synapses as shown in Figure 4.1. Both neurons and

synapses are traditionally implemented with components such as resistors, capacitors, operational amplifiers, including voltage and current sources. However, this traditional approach lacks density and scalability (see section 1.2). We are interested in exploring denser designs by using memristors, the novel nano-scale component that bypasses the density and scalability hurdle with its inherently small form factor. HP has demonstrated practical memristors working at 3nm x 3nm sizes [67]. Previously, memristors have been explored solely as a synapse in neural networks [69], effectively working as on/off switches. We use our C3EA framework to evolve memristor-based AM designs that are more area-efficient than the ANN AM blocks, and hence, have the potential to replace them.
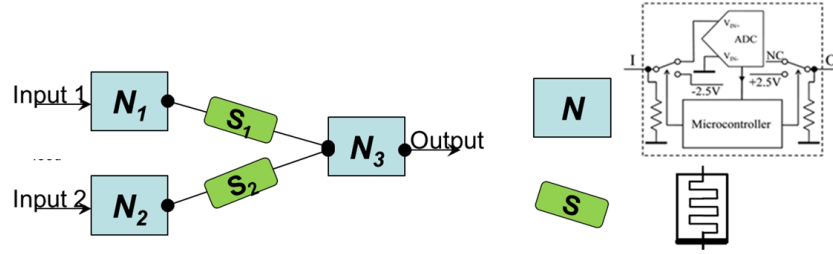
Figure 4.1: An example of the ANN-based AM design. N1, N2 are neurons that lie on the input layer and N3 is a neuron on the output layer. S2 and S2 are the synapses interconnecting the input and the output neuron layers. Both neurons and synapses are traditionally implemented using resistors, op-amps, etc. More recently, Yuriy and Massimiliano [69] have implemented synapses with memristors. Figure re-drawn from [69].

The basic functionality for the target AM has been adopted from Yuriy and Massimiliano [69]. There are four phases in the evaluation of the transient response of such memories (Figure 4.2):

- **Phase I:** *Input B* does not stimulate *Output C*.

- **Phase II:** *Input A* strongly stimulates *Output C*.

- **Phase III:** Training phase, where the inputs become "associated."

- **Phase IV:** *Input B* starts strongly stimulating *Output C.*
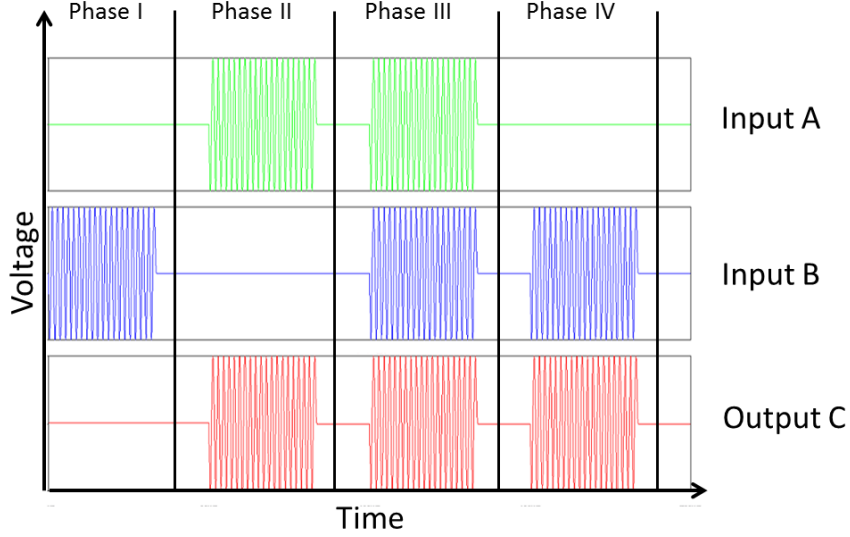
Figure 4.2: The four phases in the evaluation of an AM block. Phase I, where the second *Input B* does not stimulate the *Output C.* Phase II, where the first *Input A* strongly stimulates *Output C.* Phase III is the training phase, where the two inputs become *associated.* Phase IV, where the *Input B* starts strongly stimulating the *Output C.* The ideal basic AM response to inputs presented during the four phases as sinusoidal signal trains, each of amplitude $0.2V$, frequency of $600Hz$ and of duration $33.3ms$. There are $2,001$ data points for fitness evaluation of candidate circuits. The data points are sampled every $0.1ms$ between $0ms$ and $200ms$.

4.2.1 Embryo circuit

The purpose of the embryo circuit is to provide the input signal and loads for the *ngspice* evaluation of each randomly generated candidate AM sub-circuit. We used the embryo shown in Figure 4.3 for this experiment. There are five voltage sources, each excites a particular input during a designated phase. For example, the voltage source $V\_Phase\_I\_Input\_B$ is the source that excites the node `Input_B`

during the *Phase I* of the evaluation. R_{Load} is a $1k\Omega$ load resistor isolating the probe terminal Output_C from the ground.
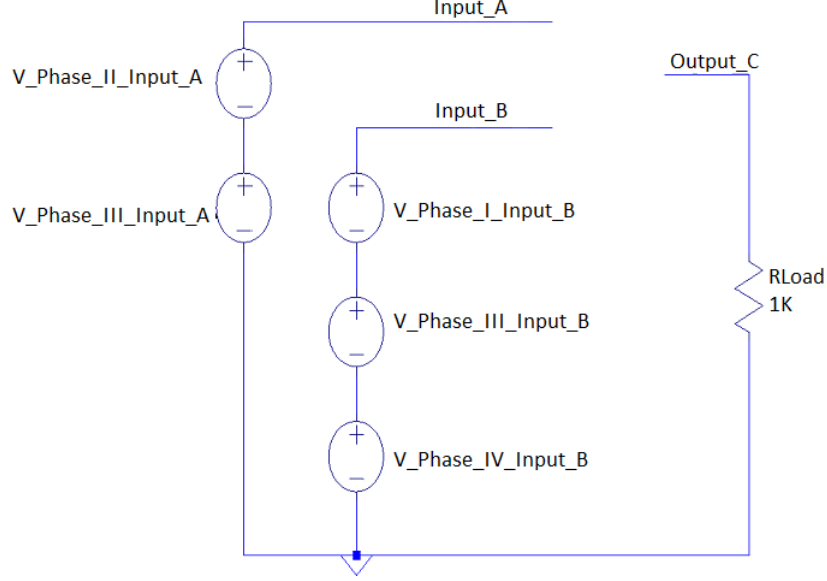
Figure 4.3: The embryo circuit for basic AM experiments. There are five voltage sources. $V\_Phase\_I\_Input\_B$ creates excitatory inputs at connection point Input_B during *Phase I*, $V\_Phase\_II\_Input\_A$ creates excitatory inputs at connection point Input_A during *Phase II*, $V\_Phase\_III\_Input\_A$ and $V\_Phase\_III\_Input\_A$ create excitatory inputs at connection point Input_A and Input_B respectively during *Phase III* and $V\_Phase\_IV\_Input\_B$ excites Input_B during *Phase IV*. R_{Load} is a $1k\Omega$ load resistor and Output_C is the probe point. The candidate AM block design evolves as a sub-circuit between the connection points Input_A, Input_B, and Output_C.

4.2.2   Component, function, and terminal nodes

In addition to the component nodes with the parameter ranges presented in Tables 3.1 and 3.2, we added a memristor component node. These memristor-creating component nodes insert either a two-terminal or a three-terminal memristor into the developing sub-circuit. For the memristor component node, the following *ngspice* memristor model from [50] was used.

Table 4.1: Memristors as component nodes used in AM evolution.

Component	Model
two-terminal memristor	X, *memristor* with connection 6 set to ground
three-terminal memristor	Xmem, *memristor* with connection 6 randomly chosen

The ngspice memristor model adapted from [50]:

```
.SUBCKT memristor 1 2 6
Eres 1 9 POLY(2) (8, 0) (11, 0) 0 0 0 0 1
Vsense 9 4 DC 0V
Fcopy 0 8 Vsense 1
Rstep 8 0 1K
Rser 2 4 10
Fmem 6 0 POLY(2) Vsense Ecopy -0.5E-10 0 1E-10 0 -1 0 0 0 1
Cmem 6 0 90nF
Rsp 6 0 1000Meg
Ecopy 7 0 0 6 1
Rc 7 0 1
Ecpy2 10 0 6 0 1
Vref ref 0 DC 1V
R1 10 11 100K
Ssat1 11 0 0 11 SWX
Ssat2 11 ref 11 ref SWX
.MODEL SWX SW(Ron=0.001, Roff=1000Meg, Vt=0.00001V, Vh=0.00001V)
.ENDS
```

This three-terminal memristor model has two passive device terminals, terminal '1' for the input and terminal '2' for the output connection. The third terminal '6' in the model functions as a control for external bias voltages if required. For the two-terminal memristor component nodes, the control terminal '6' was set at a default bias of $0V$. See Table 4.1 for the parameter range of these added memristor component nodes.

We used the extended set of function nodes presented in section 3.2.2. The function nodes included were: SERIES, PARALLEL, FLIP1, FLIP2, TO_GND1, TO_GND2, RANDOM1, and RANDOM2. The zero-argument terminal node END is maintained for

65

terminating the growing branch of a tree.

4.2.3 Fitness measure

This first set of basic AM experiments were conducted with sinusoidal signals with amplitude $0.5V$, frequency $600Hz$ and signal duration of $33ms$, i.e., 20 oscillatory pulses per signal train. We used oscillatory inputs in order to compare our results with those of Yuriy and Massimiliano [69]. The target response for basic AM with sinusoidal inputs with the four phases of evaluation was presented in Figure 4.2.

To evaluate the fitness of a candidate equivalent circuit, we stimulate it with the sinusoidal signals as shown in Figure 4.2. The figure also shows the corresponding target response at the probe point `Output_C`. We compare the candidate sub-circuits' resulting voltage at probe point `Output_C` with the target response. The *ngspice* simulation data is recorded at $0.1ms$ resolution for $200ms$ and each of those $2,001$ data points are compared with the corresponding data points from the target response. The *fitness value* is then computed from the sum of the squared-error at each data point weighted against the size of the evolved sub-circuit (see section 2.3.3). The squared-error component is normalized with the highest possible error, which was determined experimentally from initial runs to be 2.67×10^7. The size component of fitness is normalized against the maximum number of components allowed, i.e., 200 in our case. A reasonable value of the weight w determined for this experiment is 50%. This value was determined by first running the experiments for accuracy alone and determining the acceptable range of the normalized squared-error. We then adjusted the weight such that the circuit size component dominates the evolution only when the squared-error was minimized to an acceptable level of 1.5×10^{-2}. The final *fitness value* is then

calculated as:

$$fitness_{normalized} = \frac{\frac{\sum_{i=1}^{2001}(Error_i)^2}{2.67\times10^7} + \frac{size_{netlist}}{200}}{2} \qquad (4.1)$$

Here, $Error_i$ is the difference between the actual voltage at the probe point and the target voltage for i^{th} sample point every $0.1ms$. We divide the sum of the squared-error by 2.67×10^7 in order to normalize the squared-error against the maximum error. The term $\frac{size_{netlist}}{200}$ is the normalized cost associated with the size of the circuit. The final $fitness_{normalized}$ is then calculated as the average of the normalized squared-error and the normalized size cost ($w = 50\%$).

4.2.4 Control parameters

We used the default population size of 100. For this problem, the probability of the mutation was set to 30% as determined from the parameter exploration experiments detailed in section 2.5.1. We used the full 13 mutation functions (see section 2.5.2) and tried all five replacement strategies from section 2.5.4 in different GP runs. Each GP run was terminated at $5,000$ generations and the best individual from ten runs, each with a different replacement strategy, is presented as a result.

4.2.5 Results for basic spatial AM

C3EA evolved simple memristor networks with the AM functionality. Figure 4.4 shows an evolved circuit with four two-terminal memristors. With four components and a size normalization factor of 200, the cost of the evolved circuit size evaluates to 2×10^{-2}.

Figure 4.5 shows the transient response of the best-evolved equivalent circuit
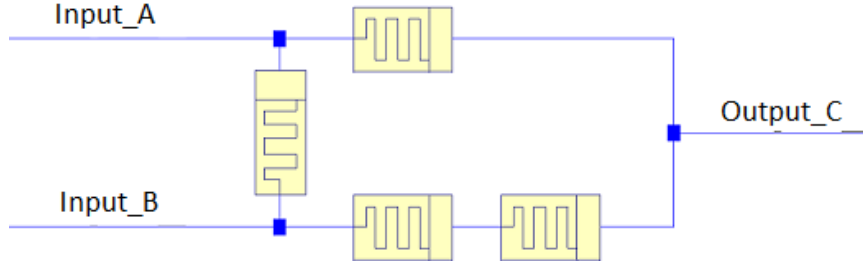
Figure 4.4: The best-evolved equivalent circuit for the basic spatial AM functionality. It comprised of four two-terminal memristors. Seven of the ten GP runs with weight w set to 50% converged to the same circuit as the optimum solution. The other three runs evolved circuits with more components.

from Figure 4.4 comparing it against the ideal response at Output_C from Figure 4.2. For this circuit, the normalized squared-error over $2,001$ data points was 3.2×10^{-3}. The final assigned normalized *fitness value*, calculated from equation 4.1, was 1.16×10^{-2}. The maximum amplitude of noise was observed during the *Phase I* of the evaluation with an amplitude of $20mV$ giving a signal-to-noise ratio of 10. In *Phase IV*, we observe that the spatial learning results in full rail $200mV$ stimulation at the output.

Figure 4.6 shows the best, the average, and the worst fitness averaged over ten GP runs. Due to the random nature of creating and mutating individuals, some mapped circuits cannot be simulated in *ngspice*. Within C3EA, these individuals are automatically assigned the maximum normalized squared-error of 1. With the weight w set to 50%, the final fitness value for such individuals would be the average of squared-error and the cost of the netlist size, i.e., $0.5 + \frac{size_{normalized}}{2}$. This explains why the worst fitness stays in the range of 0.5-0.6 throughout the run, implying that in every generation there were some individuals that were not executable by *ngspice*. The average fitness data was noisy with a high standard deviation among the different runs. This may be because each of the ten runs had
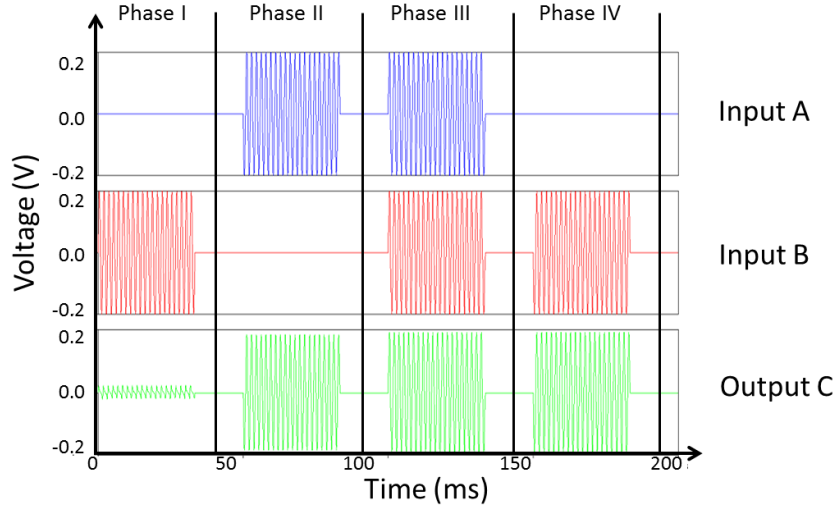
Figure 4.5: The transient response of the best-evolved basic AM circuit. Here the error was observed mostly in the *Phase I* of the basic AM evaluation. The maximum noise amplitude is observed as $20mV$ giving a signal-to-noise ratio of 10.

a different replacement strategy and hence different evolutionary dynamics. The best fitness of each run converged to a final fitness values within the first $4,000$ generations. In Figure 4.7, we plot the best fitness evolution along with error-bars every 200 generations. The error-bars represent the standard deviation among the ten GP runs. Seven out of the ten runs converged to the same fitness value. For the other three, the error-bars indicates the final fitness values were within 1% of the best fitness of the seven well-converged runs. This indicates all best-evolved individuals are close in performance.
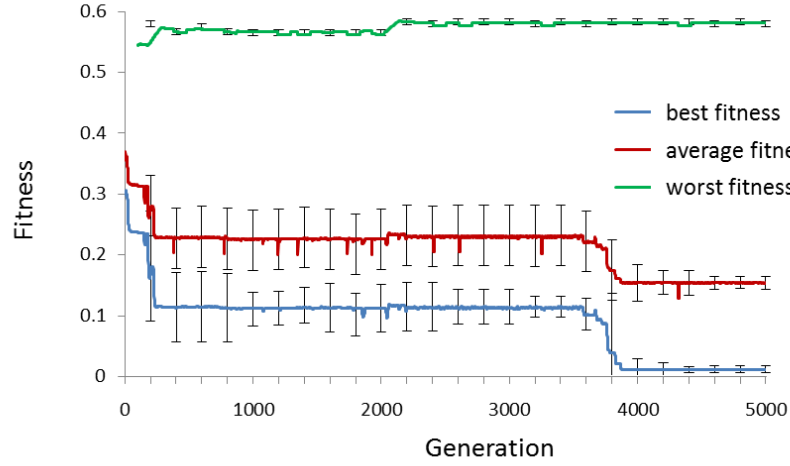
Figure 4.6: The fitness averaged over ten GP runs for evolving the basic spatial AM design. Here, the average fitness plots are noisy because the plots presented are averaged over ten runs each having a different replacement strategy and hence the evolution would vary a lot from one run to the other.
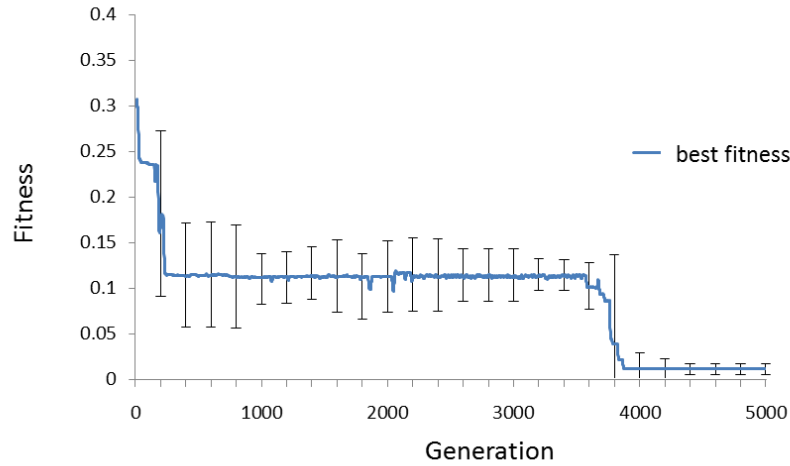
Figure 4.7: The best-evolved fitness averaged over ten GP runs for evolving the basic spatial AM design. Here, the error-bars represent the standard deviation between the runs. Both the fitness value and the error decrease as the evolution progresses and finally converge to a solution around $4,000^{th}$ generation.

4.2.6 Discussion

Memristors evidently simplified the design of AMs significantly from the micro-controller-based neural model presented in [69]. However, in the case of the best-evolved circuits response presented in Figure 4.5, for *Phase I*, one can see that the Output_C contains some noise. Some noise is both anticipated and tolerated in analog designs as both inputs and outputs are coupled through passive devices. The definition of acceptable noise level is a design choice. Our second set of experiments explored the circuit size vs. accuracy trade-off. Meanwhile, we believe that the evolution resulted in such compact circuits because memristors, like resistors, when in series, can result in a voltage drop and when in parallel can act as a voltage divider. Thus, bigger networks generated during the initial generations would, because of more memristors in series and parallel, have lower rail-to-rail voltages in *Phase II–IV*. A lower rail-to-rail voltage and a larger network size would imply higher squared-error and higher size-cost respectively. Thus, such bigger networks were eventually eliminated during the process of evolution, yielding compact 3-7 memristor circuits as the solution.

4.3 Experiment 2: The size vs. accuracy trade-off

We anticipated that by varying the weight w associated with the circuit size, our framework can be tuned to give more accurate results or lower noise levels vs. smaller designs. This set of experiments was meant to explore this size vs. accuracy trade-off. The only change in the experimental set-up was in the fitness evaluation. The embryo circuit, component and function nodes, and the control parameters were inherited from the previous experiment.

4.3.1 Fitness measure

In order to explore the trade-off in fitness evaluation, we parameterized the weight w in our fitness function. The value of w was received at run-time from the C3EA configuration file. The final *fitness value* is then calculated as:

$$fitness_{normalized} = (1-w) \times \frac{\sum_{i=1}^{2001} (Error_i)^2}{2.67 \times 10^7} + w \times \frac{size_{netlist}}{200} \qquad (4.2)$$

Here, i is the incremental data point step, and $Error_i$ is the difference between the actual voltage at the probe point and the target voltage for i^{th} sample point every $0.1ms$. We divide the sum of the squared-error by 2.67×10^7 in order to normalize the squared-error against the maximum error. The term $\frac{size_{netlist}}{200}$ is the normalized cost associated with the size of the circuit. The final $fitness_{normalized}$ is the weighted average of the normalized squared-error and the normalized size cost, where the weight parameter w is left as a run-time choice for this set of experiments.

4.3.2 Results for size vs. accuracy trade-off

Table 4.2 presents fitness values for various experiments with different size-accuracy trade-offs, highlighting the importance of the weight parameter. As we increase the constraint on size by increasing w, the framework starts reducing the component count (see size and memristor count columns) with some loss of accuracy (see the squared-error column). The associated result for each experiment is shown in Figure 4.8. We observe that the noise amplitude in *Phase I* increases with w but memristor count decreases. We also observe that the best-evolved circuits retain rail-to-rail voltage swing of $200mV$, thus almost zero noise in *Phases II-IV*.

Table 4.2: Summarizing the size vs. accuracy trade-off.

Experiment	$squared - error$ normalized	$size$ normalized	$Fitness$ normalized	Memristor count
$w = 1\%$	1.25×10^{-3}	2.50×10^{-2}	2.44×10^{-3}	5
$w = 50\%$	3.20×10^{-3}	2.00×10^{-2}	1.16×10^{-2}	4
$w = 75\%$	5.00×10^{-3}	1.50×10^{-2}	1.25×10^{-2}	3
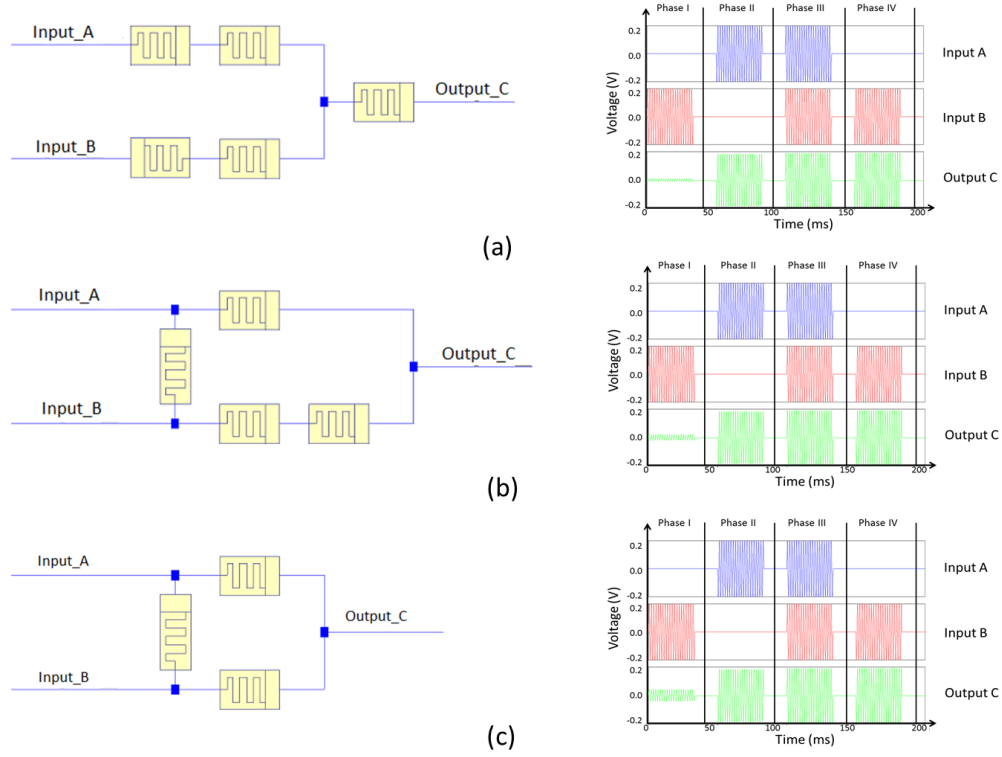
Figure 4.8: The best-evolved basic AM designs and their transient responses with (a) weight $w = 1\%$, (b) weight $w = 50\%$, and (c) $w = 75\%$. We observe that the noise amplitude in *Phase I* drops considerably as the weight on size is lowered.

4.3.3 Discussion

With this experiment, we demonstrated that one of C3EA's parameters, w, allows designers to explore the trade-off between smaller circuits and less noise. By varying the cost associated with the circuit size, the framework can be tuned to give more accurate results or lower noise levels vs. smaller designs.

4.4 Experiment 3: Basic spatial AM with pulsed input signals

We performed this third set of basic AM experiments with pulsed inputs or spikes. There were two reasons for experimenting with pulse or spike trains: first, because spikes are considered as most common form of communication signals in biological systems [51], and second, we wanted to introduce variation in the input pattern and see how much that effects the evolved AM design. In this set of basic AM experiments, we used positive square pulses of amplitude $0.2V$, pulse width of $1ms$ and time period of $2ms$, again with 20 pulses constituting each signal train. The ideal response for this set of experiments is shown in Figure 4.9. In this experiment, since all we had changed was the input signals, we could reuse the embryo circuit, nodes, and control parameters from experiment 1 (see sections 4.2.1, 4.2.2, and 4.2.4). Minor alteration in the fitness measure were needed and are discussed next.

4.4.1 Fitness measure

In order to evaluate the fitness of a candidate equivalent circuit, we stimulate it with pulsed input signals as shown in Figure 4.9. The target `Output_C` response is compared with the candidate sub-circuits' resulting voltage at probe point `Output_C`. We simulate the candidate circuit in *ngspice* and record data with
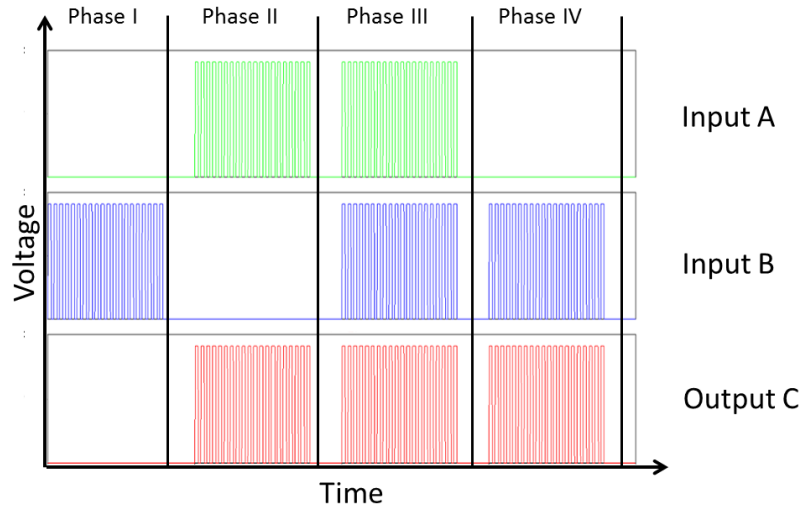
Figure 4.9: The ideal basic AM response to pulsed inputs presented during the four phases of evaluation. In each phase, the inputs are presented as 20 pulse signal trains, each of amplitude $0.2V$, pulse duration of $1ms$ and time period of $2ms$. There are $2,001$ data points for fitness evaluation of candidate circuits. The data points are sampled every $0.1ms$ between $0ms$ and $200ms$.

$0.1ms$ resolution for $200ms$. Each of those $2,001$ data points is compared with the corresponding data points from the target response (Figure 4.9). The *fitness value* is then computed from the sum of the squared-error at each data point weighted against the size of the evolved sub-circuit (see section 2.3.3). The normalization factor for the squared-error was experimentally determined to be 2.2×10^7. The size component of fitness is normalized against the maximum number of components allowed, 200 in our case. A reasonable value of the weight w determined for this experiment is 50%. The weight was chosen such that the circuit size component dominates the evolution only when the squared-error was minimized to an acceptable level of 5.0×10^{-2}. The final *fitness value* is then calculated as:

$$fitness_{normalized} = \frac{\frac{\sum_{i=1}^{2001} (Error_i)^2}{2.2 \times 10^7} + \frac{size_{netlist}}{200}}{2} \tag{4.3}$$

Here, i is the incremental data point step, and $Error_i$ is the difference between the actual voltage at the probe point and the target voltage for i^{th} sample point every $0.1ms$. We divide the sum of the squared-error by 2.2×10^7 in order to normalize the squared-error against the maximum error. The term $\frac{size_{netlist}}{200}$ is the normalized cost associated with the size of the circuit. Then the final $fitness_{normalized}$ is the average of the normalized squared-error and the normalized size cost ($w = 50\%$).

4.4.2 Results for basic spatial AM with pulsed inputs

C3EA evolved simple memristor networks with the AM functionality using spike or pulse trains as the input signal. Figure 4.10 evolved with four two-terminal memristors. With four components and a weight of 50%, the cost of the evolved circuit size evaluates to 2×10^{-2}. Despite the same component count, the best-evolved designs for the pulsed input is fundamentally different from the evolved

designs for the sinusoidal inputs (see Figures 4.4 and 4.8) because of the presence of a feedback-creating memristor.
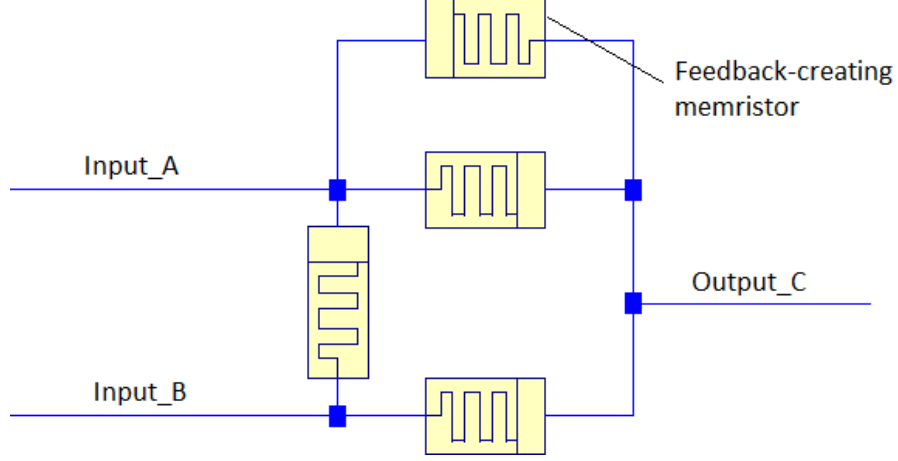
Figure 4.10: The best-evolved equivalent circuit for the basic spatial AM design using pulse train as input. It comprised of four two-terminal memristors. This design is fundamentally different from the basic AM design using sinusoidal inputs because of the presence of a feedback-creating memristor.

Figure 4.11 shows the transient response of the best-evolved equivalent circuit from Figure 4.4 comparing it against the ideal response at `Output_C` from Figure 4.2. For this circuit, the normalized squared-error over $2,001$ data points was 1.3×10^{-2}. The final assigned normalized *fitness value*, calculated from equation 4.3, was 3.3×10^{-2}. The average amplitude of noise observed during *Phase I* of the evaluation was $12mV$ giving a signal-to-noise ratio of 16. In *Phases II-IV*, we observe that the amplitude of stimulation remains within 10% of the target, and also that the amplitude changes incrementally with each pulse in the signal train. This change of amplitude indicates the gradual shift in the state of the memristors, hence we claim that the evolved AM design does exploit the non-linear property of the memristors.

Figure 4.12 shows the best, the average, and the worst fitness averaged over ten
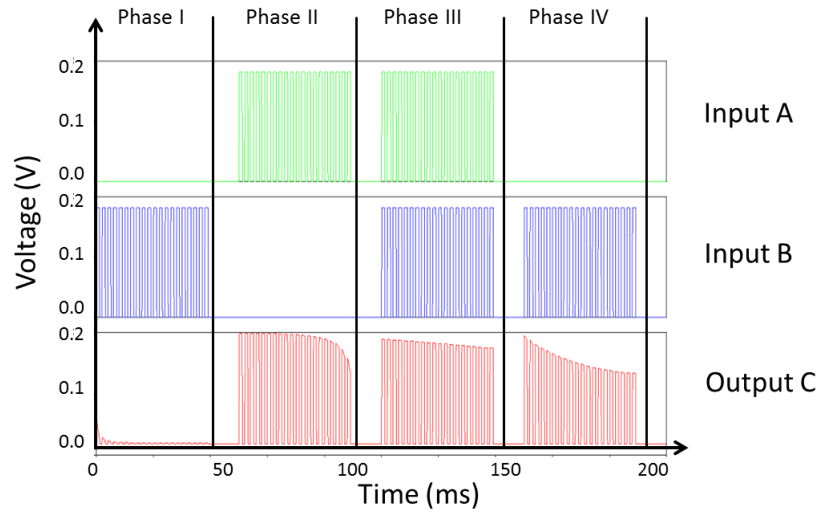
Figure 4.11: The transient response of the best-evolved basic AM circuit. Here the error was observed mostly in the *Phase I* of the basic AM evaluation. The average noise amplitude is observed in *Phase I* is $12mV$ giving a signal-to-noise ratio of 16. We also observe that the amplitude of voltage observed during *Phases II-IV* at the probe point Output_C changes its value incrementally with each pulse in the signal train, indicating that the memristor's non-linearity is being put to use.

GP runs. Due to the random nature of the circuit evolution, some individuals cannot be simulated in *ngspice*. As in the previous experiment, these non-executable individuals are assigned the maximum normalized squared-error of 1. With the weight w set to 50%, the final fitness value for such individuals should be around 0.5. The worst fitness plot stays mostly in the range of 0.5-0.6 indicating that some non-executable individuals were present in each generation for all the GP runs. The noise and high standard deviation on the average fitness plot was expected because we plot the average of ten GP runs or five pairs, each pair with a different replacement strategy. The best fitness of each run converged to final fitness values within the first $4,000$ generations. In Figure 4.13, we plot the best fitness evolution along with error-bars every 200 generations. The error-bars represent the standard deviation among the ten GP runs. Six out of the ten runs converged to the same fitness value. For the other four, the error-bars indicates the final fitness values were within 5% of the best fitness of the six well-converged runs. This implies, all the best-evolved individuals are close in performance.

4.4.3 Discussion

There are some key insights to be gained from this set of experiments.

- Despite the same component count, the evolved circuit for pulsed input signals (see Figure 4.10) were fundamentally different from the evolved circuit for sinusoidal inputs (see Figure 4.4) because of the additional feedback-creating memristor in case of the pulsed input. We observed that feedback-creating memristors were consistently selected during the evolution of pulsed AM designs. This may be because a train of positive pulses can change memristor states drastically and sometimes unnecessarily. A feedback in the
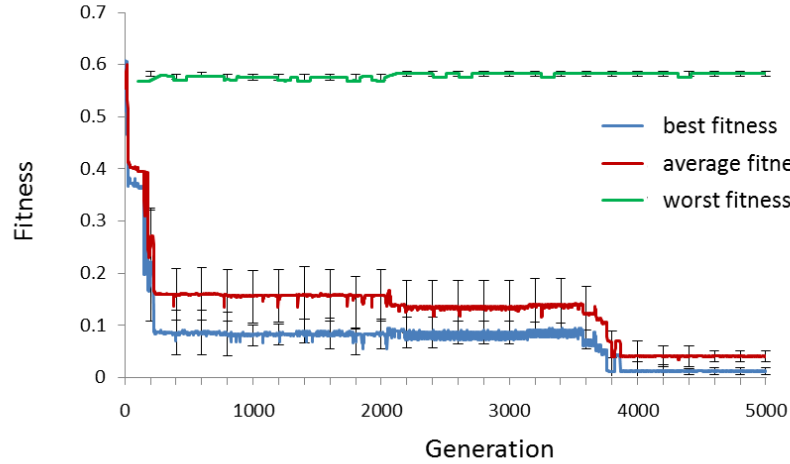
Figure 4.12: The fitness averaged over ten GP runs for evolving the pulsed input spatial AM design. Here, the error-bars represent the standard deviation between the runs.The average fitness plots are noisy because the plots presented are averaged over ten runs each having a different replacement strategy and the evolution varies from one run to the other.

design lowers the potential difference for the remaining components and thus prevents any unnecessary change of memristor states.

- We also observed gradual change in voltage at the probe point `Output_C` (see Figure 4.11). This implies that the memristors were continuously changing their internal state with each pulsed input. This memristor state change was not observed in AM designs with sinusoidal inputs (see Figure 4.5) because the change in state during the positive input cycle was undone by the following negative input cycle.

- The average noise across all four evaluation phases was lower for the spike input AM design than the corresponding sinusoidal input AM designs.
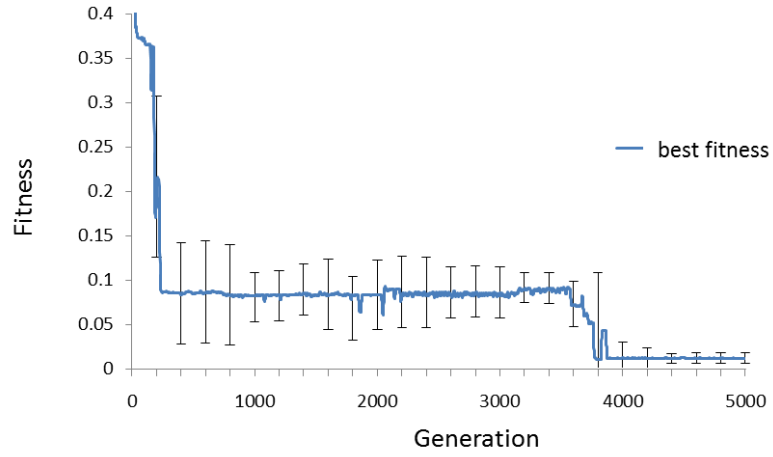
Figure 4.13: The best-evolved fitness averaged over ten GP runs for evolving the pulsed input spatial AM design. Here, the error-bars represent the standard deviation between the runs. Both the fitness value and the error decrease as the evolution progresses and finally converge to a solution around, the $4,000^{th}$ generation.

Before concluding the spike input AM experiments, we tested the evolved circuit for robustness against the noise on inputs. After the four phases of basic AM evaluation, we introduced a sinusoidal signal train on `Input_A`. This can be seen as the phase named *Noise* in Figure 4.14. We re-tested the "association" in the following phases, namely the *Test A* and *Test B*. The phase *Test A* placed the original 0.2V amplitude, 20 pulse spike train on `Input_A`. The circuit behaved normally and stimulated the probe point `Output_C`. But, in *Test B* phase, the original spike train, when placed on `Input_B`, had no stimulation on the `Output_C`. This implied that the noise was destructive for the initial "association" or learning demonstrated in *Phase IV* and that the evolved circuit was not robust. The evolved circuit was functional as a basic AM block, but in order to have robust designs, the robustness-check was to be added in the circuit-evaluation phase of the evolution. The following experiment was performed in order to test if adding a robustness-check in the evaluation phase could evolve memristor-based and noise-tolerant AM designs.

4.5   Experiment 4: Noise tolerant spatial AM

In this set of experiments, the basic AM evaluation phase was extended to include robustness-checks in the target function such that when sinusoidal inputs are presented, the evolved design retains the "association" learned during the first four evaluation phases. This is accomplished by having, a total of seven phases in the evaluation of the transient response of such memories (Figure 4.15):

- **Phase I:** *Input B* does not stimulate *Output C.*

- **Phase II:** *Input A* strongly stimulates *Output C.*
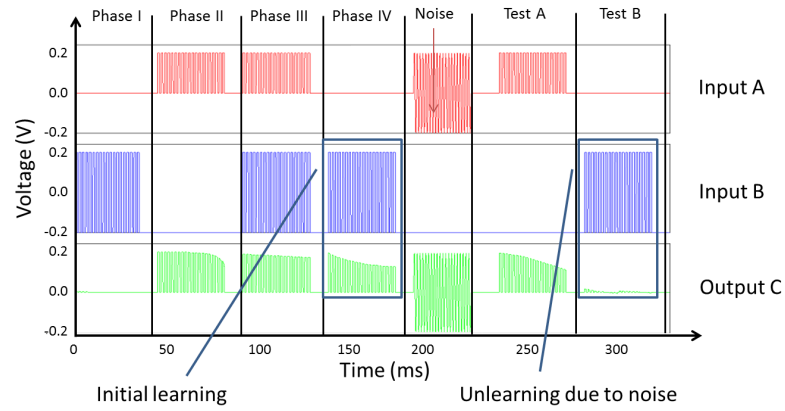
Figure 4.14: The best-evolved spike-based AM design was subjected to sinusoidal noise on Input_A during the phase *Noise*. Subsequently, in phase *Test A* and *Test B*, we tested if the original spike train could retain its initial learning and still stimulate the probe point Output_C. We observed that while Input_A could still stimulate the Output_C, the noise was destructive for Input_B's learnt association. Input_B could no longer stimulate the Output_C. The evolved circuit was not robust against the destructive noise.

- **Phase III:** Training phase, where the inputs become "associated."

- **Phase IV:** *Input B* starts strongly stimulating *Output C.*

- **Noise:** The sinusoidal noise is introduced to the circuit.

- **Test A:** This is the test phase for *Input A*, that it still strongly stimulates *Output C.*

- **Test B:** This is the test phase for *Input B*, i.e., it must retain the learnt "association" and strongly stimulate *Output C.*

The nodes and control parameters are kept the same as in Experiment 1 (see sections 4.2.2 and 4.2.4). We ran these experiments for $15,000$ generations. Some changes required in the embryo circuit and the fitness measure are discussed next.

4.5.1 Embryo circuit

The embryo circuit for the noise-tolerant AM design evolution is shown in Figure 4.16. The purpose of the three additional voltage sources ($V\_Noise\_Input\_A$, V_Test_A, and V_Test_B) is to provide the input signal for the additional three phases in the *ngspice* evaluation of each randomly generated candidate noise-tolerant AM sub-circuit.

4.5.2 Fitness measure

In order to evaluate the fitness of a candidate noise-tolerant design, we stimulate it with input signals as shown in Figure 4.15. The target `Output_C` response is compared with the candidate sub-circuits' resulting voltage at probe point `Output_C`. We simulate the candidate circuit in *ngspice* and record data with $0.1ms$ resolution
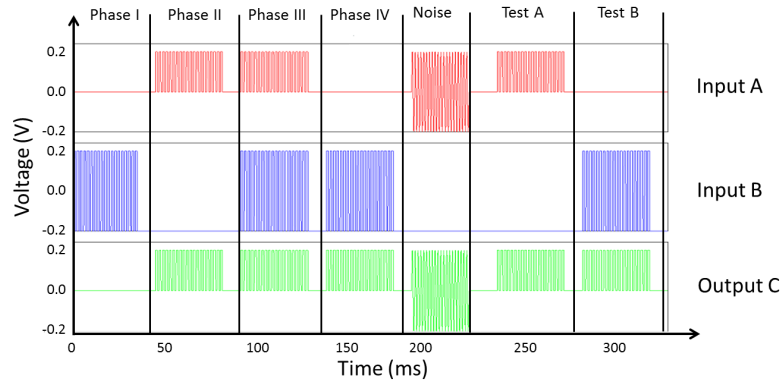
Figure 4.15: The seven phases in the evaluation of a noise-tolerant AM block. *Phases I-IV* are the same as in the basic AM evaluation. The three additional phases are: *Noise*, where the destructive noise is introduced at `Input_A`; *Test A*, where we test whether `Input_A` still strongly stimulates `Output_C`; and *Test B*, where we test that the circuit retains `Input_B`'s learnt "association" and strongly stimulates `Output_C`. The noise-tolerant AM response, at the probe point `Output_C`, to the inputs presented during the seven phases has $3,501$ data points for fitness evaluation of the candidate circuits. The data points are sampled every $0.1ms$ between $0ms$ and $350ms$.
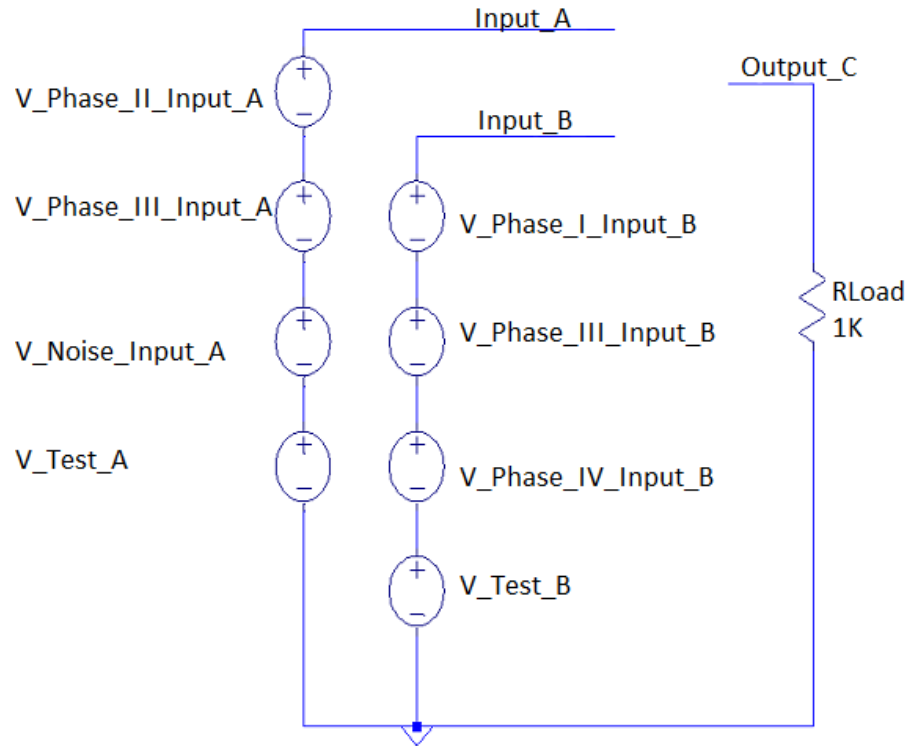
Figure 4.16: The embryo circuit for the noise-tolerant AM experiments. There are three additional voltage sources. The source$V\_Noise\_Input\_A$ creates sinusoidal noise at connection point Input_A during the phase *Noise*. The sources V_Test_A and V_Test_B create excitatory inputs at connection points Input_A and Input_B during the phases *Test A* and *Test B* respectively.

for $350ms$. Each of those $3,501$ data points is compared with the corresponding data points from the target response (Figure 4.15). The *fitness value* is then computed from the sum of the squared-error at each data point weighted against the size of the evolved sub-circuit (see section 2.3.3). The normalization factor for the squared-error was experimentally determined to be 6.3×10^8. The size component of fitness is normalized against the maximum number of components allowed, 200 in our case. The value of the weight w is retained as 50%. The final *fitness value* is then calculated as:

$$fitness_{normalized} = \frac{\frac{\sum_{i=1}^{3501} (Error_i)^2}{6.3 \times 10^8} + \frac{size_{netlist}}{200}}{2} \qquad (4.4)$$

Here, $Error_i$ is the difference between the actual voltage at the probe point and the target voltage for i^{th} sample point every $0.1ms$. We divide the sum of the squared-error by 6.3×10^8 in order to normalize the squared-error against the maximum error. The term $\frac{size_{netlist}}{200}$ is the normalized cost associated with the size of the circuit. Then the final $fitness_{normalized}$ is the average of the normalized squared-error and the normalized size cost ($w = 50\%$).

4.5.3 Results for noise-tolerant AM

C3EA evolved memristor networks with the noise-tolerant AM functionality using a spike train as the input signal for training and test along with a sinusoidal input train introduced as noise. Figure 4.17 evolved with 18 two-terminal memristors. With 18 components and a weight of 50%, the cost of the evolved circuit size evaluates to 9×10^{-2}. The best-evolved noise-tolerant design seems to be evenly split into master and slave sub-circuits with nine memristors each. The master and slave sub-circuits are connected via common nodes. The evolved sub-circuit

shows a lot of feedback in its design.
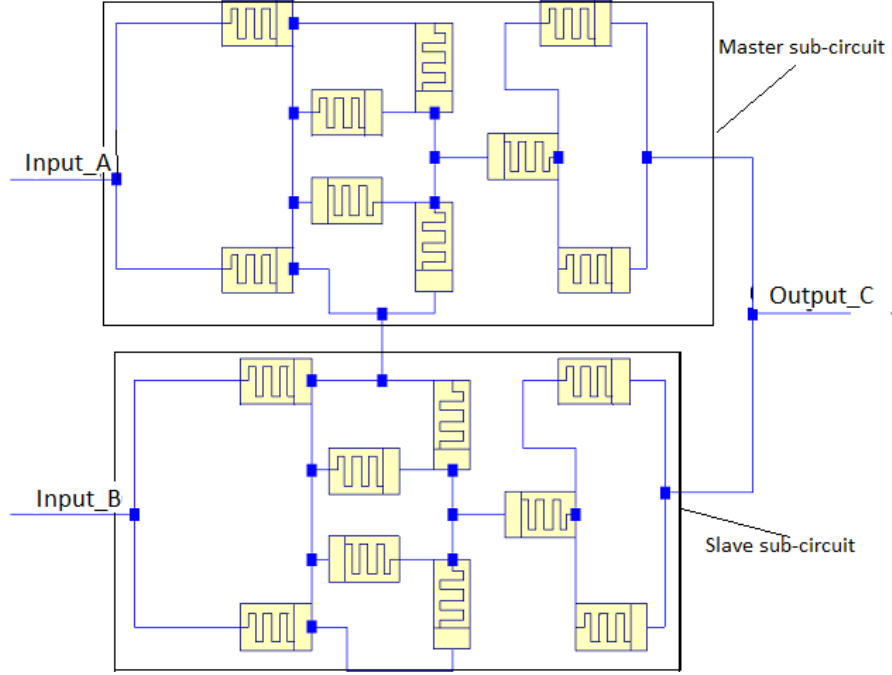
Figure 4.17: The best-evolved equivalent circuit for the noise-tolerant spatial AM design comprises of 18 two-terminal memristors. This design has evolved with components evenly divided between master and slave sub-circuits of nine memristors each. The master and slave sub-circuits are connected via some common nodes. The evolved sub-circuit shows a lot of feedback in its design.

Figure 4.18 shows the transient response of the best-evolved noise-tolerant AM circuit from Figure 4.17 comparing it against the ideal response at `Output_C` from Figure 4.15. For this circuit, the normalized squared-error over $3,501$ data points was 1.71×10^{-3}. The final assigned normalized *fitness value*, calculated from equation 4.4, was 9.17×10^{-2}. The average amplitude of noise observed across all seven evaluation phases was $10mV$ giving a signal-to-noise ratio of 20. We also observe that the initial learning from *Phases I-IV* is retained in the test phases *Test A* and *Test B*. The stimulation amplitude remains within 10% of the target (see Figure 4.15) during all of the seven phases. Also, the amplitude changes

incrementally with each pulse in the signal train. This gradual change of amplitude indicates the shift in the state of the memristors, hence we claim that the evolved noise-tolerant AM design exploits the non-linear property of the memristors.
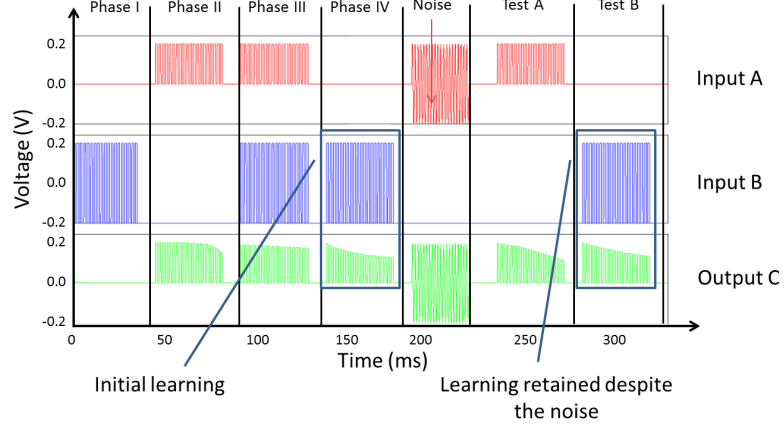
Figure 4.18: The transient response of the best-evolved noise-tolerant AM circuit. Here the transient response was within 10% of the target during all seven of the evaluation phases. The average noise amplitude is observed in *Phase I* is $10mV$ giving a signal-to-noise ratio of 20. We also observe that the amplitude of voltage at the probe point `Output_C` changes its value incrementally with each pulse in the signal train, indicating that the memristor's non-linearity is being put to use.

Figure 4.19 shows the best, the average, and the worst fitness averaged over ten GP runs. Due to the random nature of the circuit evolution, some individuals cannot be simulated in *ngspice*. As in the previous experiments, these non-executable individuals are assigned the maximum normalized squared-error of 1. With the weight w set to 50%, the final fitness value for such individuals should be around 0.5. The worst fitness plot stays mostly in the range of 0.6-0.7, indicating that some large-sized non-executable individuals were present in each generation for all the GP runs. The noise and high standard deviation on the average fitness plot was expected because we plot the average of ten GP runs or five pairs, each pair

89

with a different replacement strategy. The best fitness of each run converged to final fitness values within the first $12,000$ generations. In Figure 4.20, we plot the best fitness evolution along with error-bars every 500 generations. The error-bars represent the standard deviation among the ten GP runs. Four out of the ten runs converged to the same fitness value. For the other six, the error-bars indicates the final fitness values were within 5% of the best fitness of the four well-converged runs. This implies that all the best individuals are close in performance.
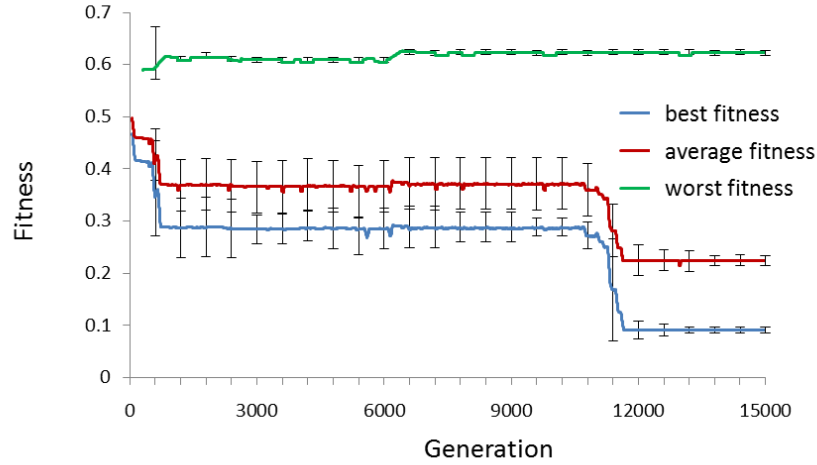
Figure 4.19: The fitness averaged over ten GP runs for evolving the noise-tolerant AM design. Here, the error-bars represent the standard deviation between the runs. The average fitness plots are noisy because the plots presented are averaged over ten runs, each pair with a different replacement strategy and hence the evolution varies from one pair to the other.

4.5.4 Discussion

The result presented here are valid for one kind of noise using a sinusoidal input train. This experiment is a proof-of-concept that if the desired robustness or noise-tolerance could be modeled within the fitness function, then C3EA can
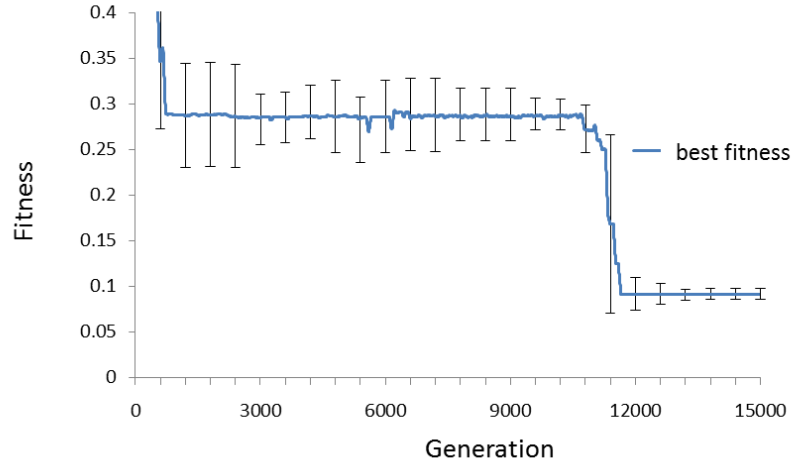
Figure 4.20: The best-evolved fitness averaged over ten GP runs for evolving the noise-tolerant AM design. Here, the error-bars represent the standard deviation between the runs. Both the fitness value and the error decrease as the evolution progresses and finally converge to a solution around, the $12,000^{th}$ generation.

be a handy tool for automated circuit design. The evolved noise-tolerant designs from this experiment exhibit reasonable accuracy but at the cost of more components and more complexity compared to the four-memristor basic AM designs (see section 4.4.2).

4.6 Discussion

In this chapter we have presented our results for memristor-based AM designs. We described the details of each experimental setup along with circuit evolution statistics. One of the framework parameters, w, was varied in order to explore the trade-off between smaller circuits and less noise. HP has demonstrated practical memristors working at 3nm x 3nm sizes. At these densities our evolved AM designs can easily rival even the current sub-25 nm flash memory technology in terms of area. In the absence of a solid design methodology, we believe that automated

91

circuit discovery is a promising tool for memristor-based circuits. Our results show that we can efficiently implement complex functions with few components. The evolved circuit can be used as a building block for more complicated systems. For example, associative memories can be used as a building block for hierarchical learning systems (see section 1.1). The framework could thus be fed the evolved designs from Experiments 1-3 as sub-circuits in order to model higher level systems. We also demonstrated that C3EA can evolve noise-tolerant AM designs if the tolerance-checks are added to the fitness function.

5

Results: Evolving Temporal Associative Memories

In this chapter we present how we used C3EA to evolve analog memristor-based networks that can solve a general problem of recognizing patterns in a time-dependent sequence of input signals. The algorithm uses a patterned set of pulsed signals to train and process the sequence. The stream presented comprised of: a) the input signals, and b) the context signals. The objective is to implement a context-sensitive state-machine. Tank and Hopfield [62] implemented an analog model of a neural network that could process the state information concentrated in time and switch its neural states based on the environment or the context. They proposed that such networks have applications in the area of pattern and speech recognition. For example, the word "the" has a different pronunciation when placed before a consonant (pronounced as 'D@') or a vowel (pronounced 'Di'). We focus on a simplified problem of recognizing the context signal, and consequently switching states, in a continuous stream of coded characters.

5.1 Temporal Association Problem

Kohonen [32] studied neural networks that operate as finite-state-machines that use the present inputs to produce the present outputs, which in turn are used as a partial inputs for the next system iteration. Other related work includes a temporal-associative network by Fukushima [18], hierarchical temopral memories by George and Hawkins [20], and sequence recognition and completion by Rumelhart and McClelland [52]. Memristor-based networks that recognize sequences,

to the best of our knowledge, have not been demonstrated yet. This problem is addressed in the present work. It is illustrated by two signals, in which the letters A and C represent a particular momentary stimulus state. The letter C is assigned the special property of being the context. A model stimulus sequence is presented in the first row below. The second row is the model response expected.

```
Input: A C A A C A A A C C A A
Output: X Y Z X Y Z X X Y Y Z X
```

Here, letter A is assigned a standard response output of letter X (*state 1*). But every time the context letter C is presented, letter A that follows the letter C must output letter Z (*state 2*). The output response for the letter C is Y (a *don't-care* state). The target state-machine is shown in Figure 5.1.

5.2 Experiment 4: Context sensitive system

Figure 5.2 shows the training pattern presented to C3EA with the context signal (letter C) and the input signal (letter A). The two signals are separated in time to ensure the evolved network discovers the temporal association between the context signal and output state. The figure also presents the ideal output signal during the four phases of the network evaluation, which are:

- **Phase I:** The input pulses (coded as letter A) stimulates the output in *state 1* (coded as letter X).

- **Phase II:** The context signal is presented (coded as letter C). We do not evaluate the output during this phase. The output state during this phase
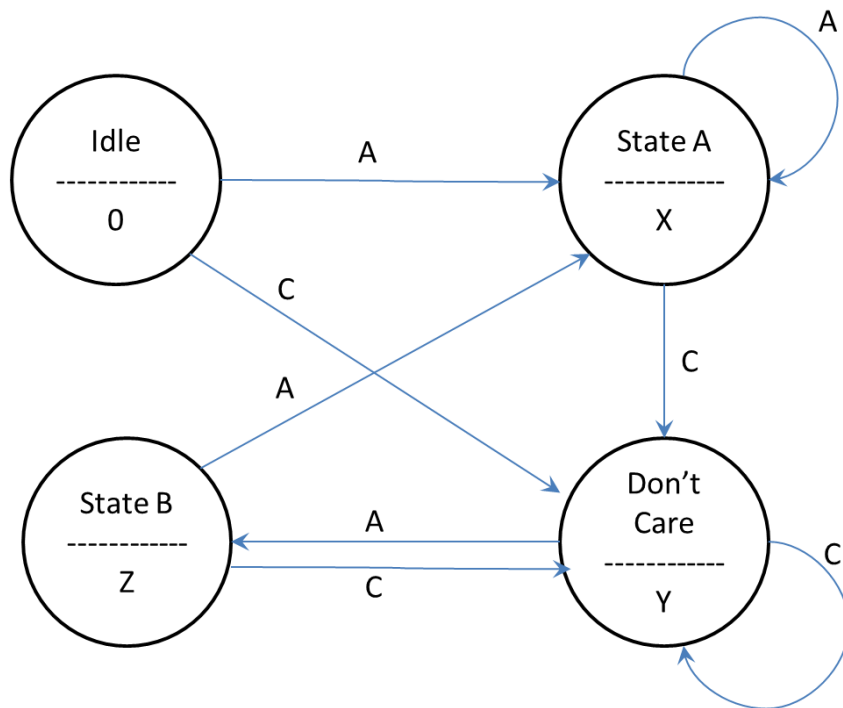
Figure 5.1: The state transition diagram for the task of context-recognition. The transitions occur when the system is presented with either the input signal (letter A) or the context signal (letter C). Letters X, Y, and Z are output signals assigned to *state 1*, *don't care*, and *state 2* respectively.

is essentially *don't-care* (coded as letter `Y`).

- **Phase III:** The context is acknowledged and the input pulses (letter `A`) stimulate the output in *state 2* (coded as letter `Y`).

- **Phase IV:** We present input pulse (letter `A`) again, and this switches the output back to *state 1* (letter `X`).
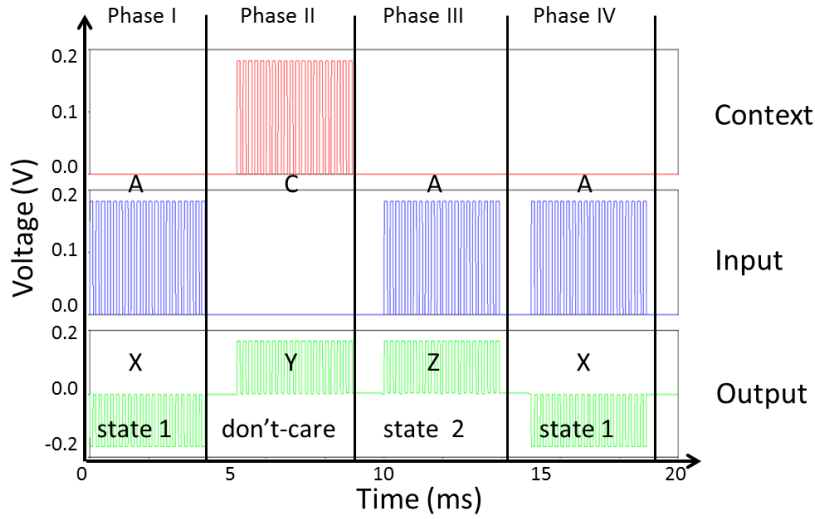
Figure 5.2: The four phases in the evaluation of a context-sensitive AM block. *Phase I*, where the input pulses (coded as letter `A`) stimulates the output in *state 1* (coded as letter `X`). *Phase II*, where the context signal is presented (coded as letter `C`). We do not evaluate the output during this phase. The output state during this phase is essentially *don't-care* (coded as letter `Y`). *Phase III*, the context is acknowledged and the input pulses (letter `A`) stimulate the output in *state 2* (coded as letter `Y`). *Phase IV*, we present input pulse (letter `A`) again, and this switches the output back to *state 1* (letter `X`). This is the target context-sensitive AM response to inputs presented during the four phases as a train of 20 pulsed signals, each of an amplitude $0.2V$, a pulse duration of $0.1ms$ and a time-period of $0.2ms$. There are $2,001$ data points for fitness evaluation of candidate circuits. The data points are sampled every $0.01ms$ between $0ms$ and $20ms$.

5.2.1 Embryo circuit

The purpose of the embryo circuit is to provide the input and the context signal and loads for the *ngspice* evaluation of each randomly generated candidate memristor-based context-sensitive AM network. We used a 10×10 cross-wire structure as the embryo shown in Figure 5.3. This embryo structure was adopted from Tank and Hopfield's analog neural network implementation [62]. There are four voltage sources, each excites the connection points `Input` or `Context` during a designated phases. The output signals are tapped at two connection points (`Output_1` and `Output_2`). The final output at the probe point `Output` is the difference of the voltages at `Output_1` and `Output_2`, amplified with a gain factor of 3. $R1$ is a $1k\Omega$ load resistor isolating the probe terminal `Output` from the ground.

5.2.2 Component, function and terminal nodes

We used the two-terminal memristor-creating component node from section 4.2.2 for this experiment. Each of the 100 cross-points in the 10×10 cross-wire structure is assigned an ordered label by its coordinates. The memristor-creating node randomly selects a cross-point and attaches a memristor to it.

We used the extended set of function nodes presented in section 3.2.2. The function nodes included were: `SERIES`, `PARALLEL`, `FLIP1`, `FLIP2`, `TO_GND1`, `TO_GND2`, `RANDOM1`, and `RANDOM2`. The function nodes `RANDOM1` and `RANDOM2` connects distant points on the network by randomly altering one of the cross-wire identifying coordinate labels. The zero-argument terminal node `END` is maintained for terminating the growing branch of a tree.
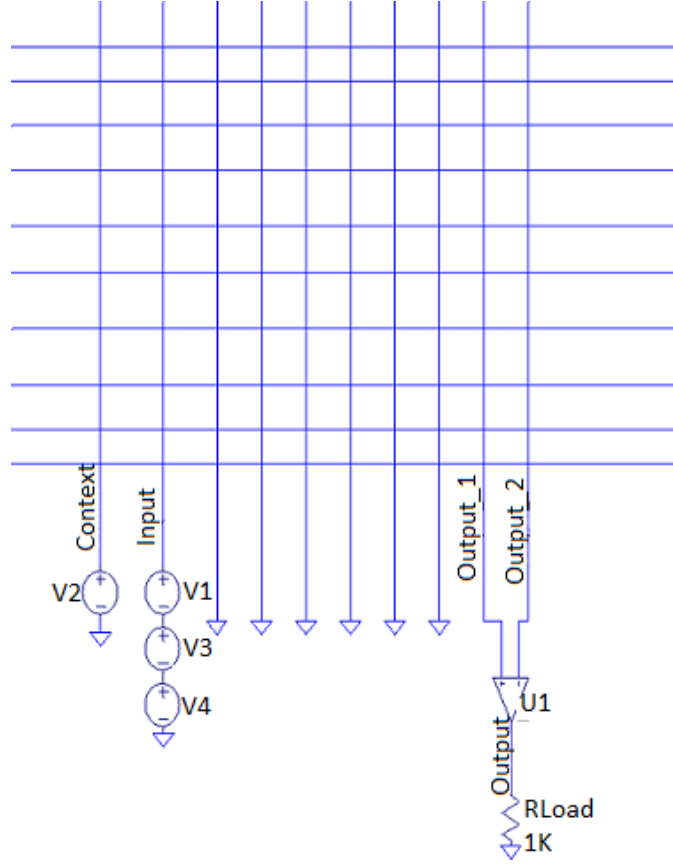
Figure 5.3: The 10×10 cross-wire embryo structure for context-sensitive AM experiments. There are four voltage sources, each creating excitatory inputs at connection point `Input` or `Context` during a designated phase. The final output at the probe point `Output` is the difference of the voltages at `Output_1` and `Output_2`, amplified with a gain factor of 3. $R1$ is a $1k\Omega$ load resistor isolating the probe terminal `Output` from the ground.

5.2.3 Fitness measure

In order to evaluate the fitness of a candidate networks, we stimulate it with pulsed context and input signals in sequence as shown in Figure 5.2. The target output response is compared with the candidate networks' voltage at probe point `Output`. We simulate the candidate network in *ngspice* and record data with $0.01ms$ resolution for $20ms$. Each of those $2,001$ data points are compared with the corresponding data points from the target response (Figure 5.2). The *fitness value* is then computed from the sum of the squared-error at each data point weighted against the size of the evolved sub-circuit (see section 2.3.3). The normalization factor for the squared-error was experimentally determined to be 8×10^7. The size component of fitness is normalized against the maximum number of components in the network, 800 in our case. Note that although the network size for the embryo is 10×10, more components are possible in the network by series or parallel addition. The weight w used for this experiment is 50%. The final *fitness value* is then calculated as:

$$fitness_{normalized} = \frac{\frac{\sum_{i=1}^{2001}(Error_i)^2}{8 \times 10^7} + \frac{size_{netlist}}{800}}{2} \qquad (5.1)$$

Here, i is the incremental data point step, and $Error_i$ is the difference between the actual voltage at the probe point and the target voltage for i^{th} sample point every $0.01ms$. We divide the sum of the squared-error by 8×10^7 in order to normalize the squared-error against maximum error. The term $\frac{size_{netlist}}{800}$ is the normalized cost associated with the size of the circuit. Then the final $fitness_{normalized}$ is the average of the normalized squared-error and the normalized size cost ($w = 50\%$).

5.2.4 Control parameters

We used the default population size of 100. For this problem, the probability of the mutation was set to 30% as determined from the parameter exploration experiments detailed in section 2.5.1 and we used the full 13 mutation functions (see section 2.5.2). We tried all five replacement strategies from section 2.5.4 in different GP runs. Each GP run was terminated at 15,000 generations and the best individual from five runs, each with a different replacement strategy is presented as a result.

5.2.5 Results for context-sensitive system design

C3EA evolved simple memristor networks, shown in Figure 5.4, with the context-sensitive AM functionality using pulse trains as the input and the context signal. With 14 components and a maximum component count of 800, the cost of the evolved circuit size evaluates to 1.75×10^{-2}. All the best-evolved designs from the five GP runs evolved with no feedback-creating memristors.

Figure 5.5 shows the transient response of the best-evolved equivalent circuit from Figure 5.4 comparing it against the ideal response at `Output` from Figure 5.2. For this circuit, the normalized squared-error over $2,001$ data points was 4.1×10^{-3}. The final assigned normalized *fitness value*, calculated from equation 5.1, was 1.08×10^{-2}. In *Phase II*, we observe that the amplitude of stimulation changes incrementally with each pulse in the signal train. This continuous change of amplitude implies a gradual shift in the state of the memristors, which is responsible for the correct response at `Output` during *Phases III and IV*. The observed change in state of the memristors implies that the non-linear and time-dependent properties of the memristor are being used to store and switch states.
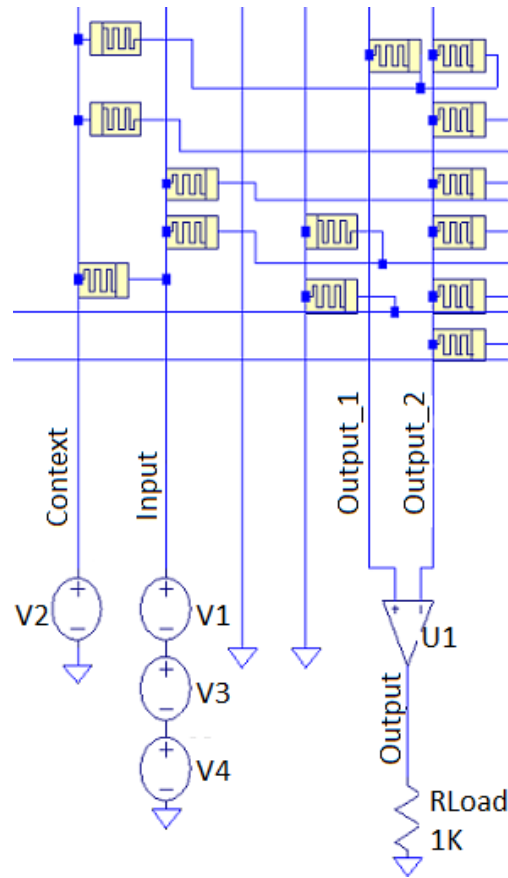
Figure 5.4: The best-evolved equivalent circuit for the context-sensitive AM design. It comprised of 14 memristors. This design evolved with no feedback-creating memristors.
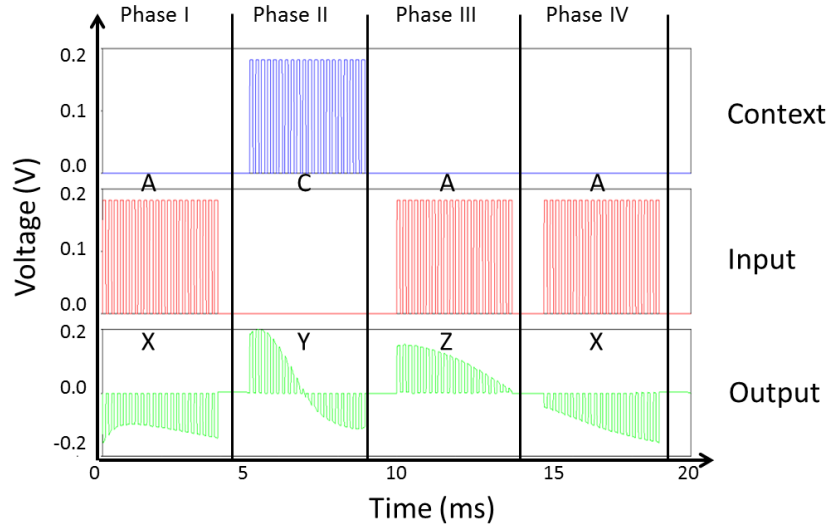
Figure 5.5: The transient response of the best-evolved context-sensitive AM network. At the probe point `Output`, the *state 1* (with negative amplitude) during *Phase I* and *Phase III* is clearly distinguishable from *state 2* (with positive amplitude during *Phase III*. We also observe that the amplitude of the voltage during *Phase II* at the probe point changes its value incrementally with each pulse in the signal train, indicating that the memristor properties of non-linearity and time-dependency are being used to store and switch states.

102

Figure 5.6 shows the best, the average, and the worst fitness averaged over five GP runs. Due to the random nature of the circuit creation, some individuals cannot be simulated in *ngspice*. Like in previous experiment, these non-executable individuals are assigned the maximum normalized squared-error of 1. With the weight w set to 50%, the final fitness value for such individuals should be around 0.5. The worst fitness plot stays mostly in the range of 0.5-0.6 indicating that some non-executable individuals were present in each generation for all the GP runs. The noise and high standard deviation on the average fitness plot was expected because we plot the average of five GP runs, each with a different replacement strategy. The best fitness of each run converged to final fitness values within first $12,000$ generations. In Figure 5.7, we plot the best fitness evolution along with error-bars every 500 generations. The error-bars represent the standard deviation among the five GP runs. All the five runs converged to a fitness value within 3% of the best fitness of all the runs. This implies that all the best-evolved individuals are close in performance.

5.2.6   Discussion

This experiment explicitly uses the non-linear and time-dependent properties of memristors to both store and switch states and the corresponding outputs. The network training was performed with a short sequence of the input and the context signals. We observed that the evolved network used the information from an earlier and current phase to process its outputs. Thus, the evolved networks had the ability to recognize context and have a temporary state for storage over short trains of pulsed inputs.
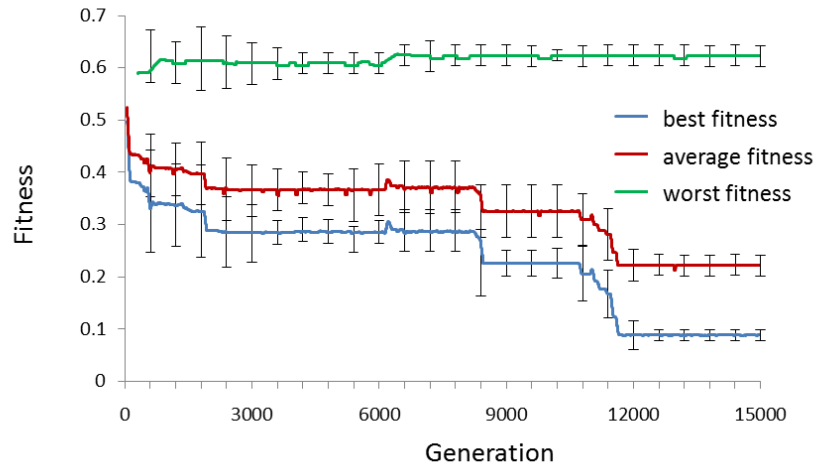
Figure 5.6: The fitness averaged over five GP runs for evolving the context-sensitive AM design. Here, the average fitness plots are noisy because the plots presented are averaged over five runs, each having a different replacement strategy, and hence the evolution varies a lot from one run to another.
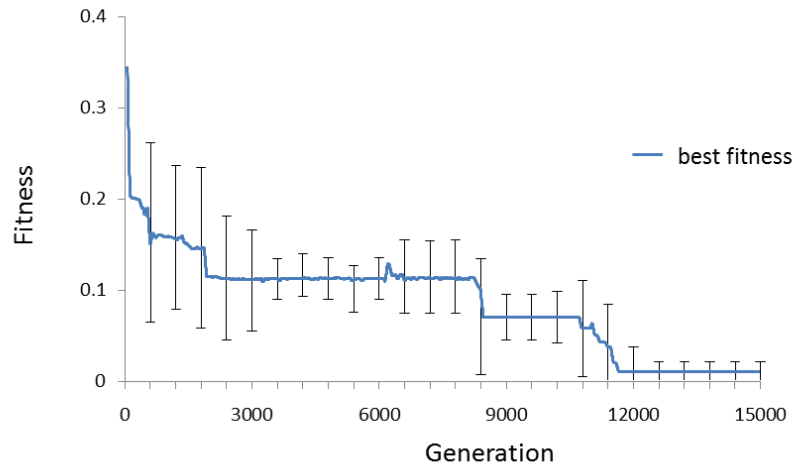
Figure 5.7: The best-evolved fitness averaged over five GP runs for evolving the context-sensitive AM design. Here, the error-bars represent the standard deviation between the runs. Both the fitness value and the error decrease as the evolution progresses and finally converges to a solution around generation 12,000.

5.3 Experiment 5: Sequence test and limitation check

The context-sensitive design evolved in Experiment 4 gave clearly distinguishable states within the pattern it was trained against. In this set of experiments we investigated whether the evolved design could recognize the context letter in a longer stream of random inputs. Additionally, we have also established the tolerances for frequency and voltage in the evolved design.

5.3.1 Experiment 5a: Sequence test

We subjected the evolved network from Figure 5.4 to an input sequence as presented in the first row below. The second row is the expected response.

```
Input:  A C A A C A A A C C A A
Output: X Y Z X Y Z X X Y Y Z X
```

Here, letter A is assigned a standard response output of letter X (*state 1* with negative polarity). But, every time the context letter C is presented, letter A that follows the letter C must output letter Z (*state 2* with positive polarity). The output response for the letter C is Y (a *don't-care* state).

The transient response for the above sequence of inputs is shown in Figure 5.8. The network recognizes the context signal in the stream and switches to a correct state for the following input signal. The test shows, the training pattern from Figure 5.2 was sufficient even for a longer input stream with a random sequence of inputs.
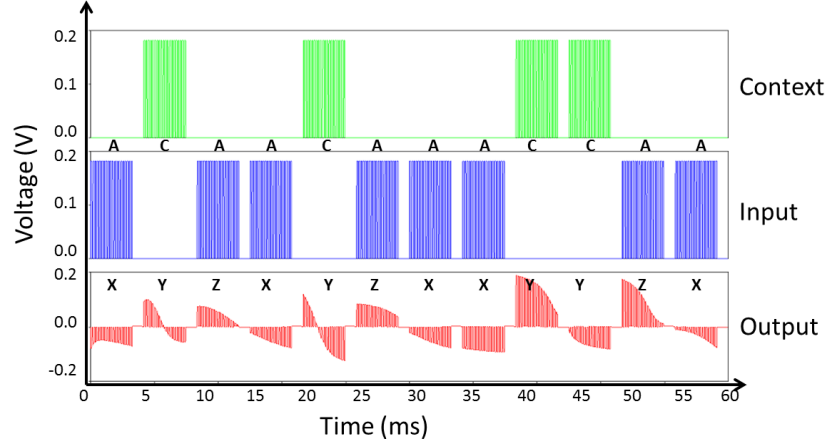
Figure 5.8: The best-evolved circuit from Experiment 4 was subjected to a longer sequence of random inputs (A C A A C A A A C C A A). The network recognizes the context signal in the stream and switches to a correct state for the following input signal (X Y Z X Y Z X X Y Y Z X).

5.3.2 Experiment 5b: Voltage limit test

Next, we performed a tolerance check on the evolved design by varying the amplitude of the input signals. The input pattern (A C A A), was presented with the signal amplitudes of: $0.1V$, $0.2V$, and $0.4V$. Figure 5.9 shows the response of the evolved design. The input streams with $0.1V$ and $0.2V$ amplitudes had the correct response (X Y Z X). But, the response for the input stream with 0.4 V was ambiguous in *Phase III* (X Y ? X). This implies that with $0.4V$ input signal amplitude, we had exceeded the design limit for the evolved network from Experiment 4.

We re-tested the limit by presenting the same input pattern (A C A A), now with the signal amplitudes of: $0.1V$, $0.2V$, and $0.3V$. Figure 5.10 shows that the transient response was correct (X Y Z X) for all three amplitudes. We established that the evolved network from Experiment 4 functions correctly for the signals with amplitudes in the range of $0.1 - 0.3V$.

106
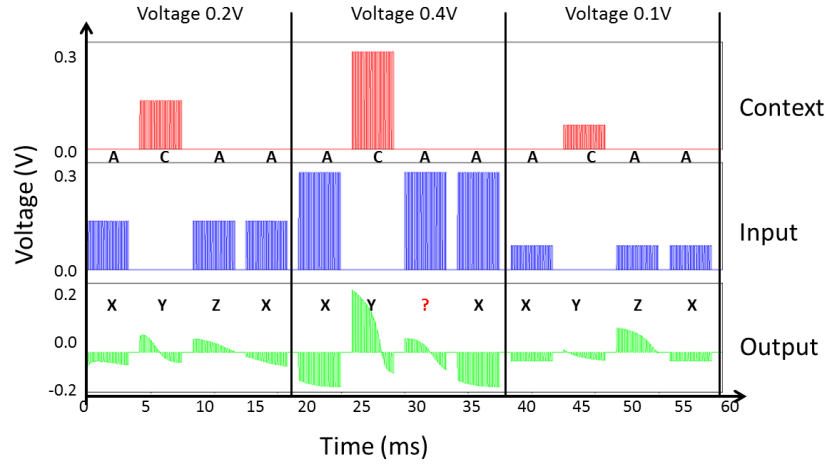
Figure 5.9: The best-evolved circuit from Experiment 4 was subjected to the input pattern (A C A A) with varying amplitudes of: $0.1V$, $0.2V$, and $0.4V$. The network gave the correct output response for the $0.1V$ and $0.2V$ signals (output (X Y Z X), but the response for the $0.4V$ signal was ambiguous in *Phase III* (X Y ? X).
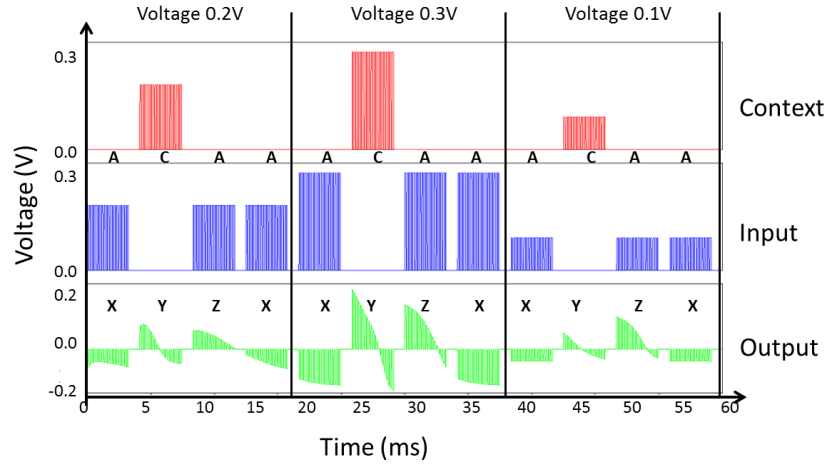
Figure 5.10: The best-evolved circuit from Experiment 4 was subjected to the input pattern (A C A A) with varying amplitudes of: $0.1V$, $0.2V$, and $0.3V$. The network gave the correct output response (X Y Z X) for all three amplitudes. The evolved design was tested to be functional for the signals with amplitudes in the range of $0.1 - 0.3V$.

5.3.3 Experiment 5c: Frequency limit test

Finally, we tested the evolved network from Experiment 4 for signal frequency limits. We applied the input pattern (A C A A) with frequencies of: $5kHz$, $10kHz$, and $2.5kHz$. The evolved design gave correct output response (X Y Z X) for all three cases. The transient response of the circuit is shown in figure 5.11. Thus, we tested the evolved design from Experiment 4 functions correctly for the signals with frequencies in the range of $2.5 - 10kHz$.
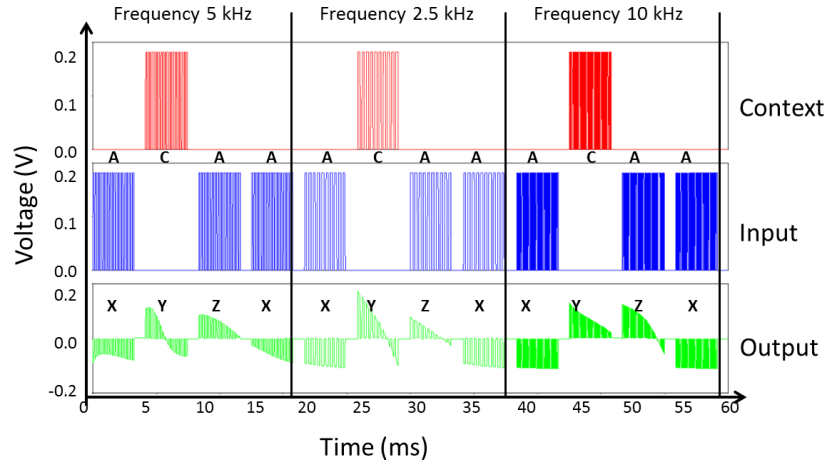
Figure 5.11: The best-evolved circuit from Experiment 4 was subjected to the input pattern (A C A A) with varying frequencies of: $5kHz$, $10kHz$, and $2.5kHz$. The network gave the correct output response (X Y Z X) for all three amplitudes. The evolved design was tested to be functional for the signals with frequencies in the range of $2.5 - 10kHz$.

5.3.4 Discussion

These experiments were conducted on the evolved design from Experiment 4. The evolved 14 memristor context-sensitive network was tested for robustness against variations on the input signals. We gained the following insights from this set of

experiments:

- The sequence of four input training pattern (`A C A A`) was sufficient to train the network for a longer stream of random inputs.

- There was no feedback-creating memristor in the evolved design. Feedback-creating memristors were used in previous spatial AM designs (see section 4.4.3) to suppress unwanted change of states. Given that this context-sensitive system actively used the memristors' change of state, any networks with feedbacks were rejected during the selection process.

- The evolved design could tolerate 100% variation in frequency and 50% variation in the amplitude of the input signal.

5.4   Experiment 6: Evolving variation tolerant AM

Experiment 5b with a 100% variation on signal amplitude failed to give us the desired response. This prompted us to add the 100% variation-tolerance requirement in the fitness function and let C3EA explore the target design.

5.4.1   Experiment 6a: Evolving voltage-tolerant AM

For this experiment, the four phases in the context-sensitive AM evaluation were extended to include the 100% signal amplitude variation-checks. This is accomplished by having three blocks of four phases in the evaluation of the transient response, one block each for the signals with amplitudes: $0.1V$, $0.2V$, and $0.4V$. Figure 5.12) shows the ideal transient response.

The nodes and control parameters are kept the same as those in Experiment 4 (see sections 5.2.2 and 5.2.4). We ran these experiments for $15,000$ generations.

The embryo circuit essentially remains the same 10×10 cross-wire structure. The number of voltage sources is altered to match the input and the context signal pattern shown in Figure 5.12. The only change in the fitness measure from section 5.2.3 equation 5.1 was in the normalization factor for the squared-error which was experimentally determined to be 2.4×10^8.
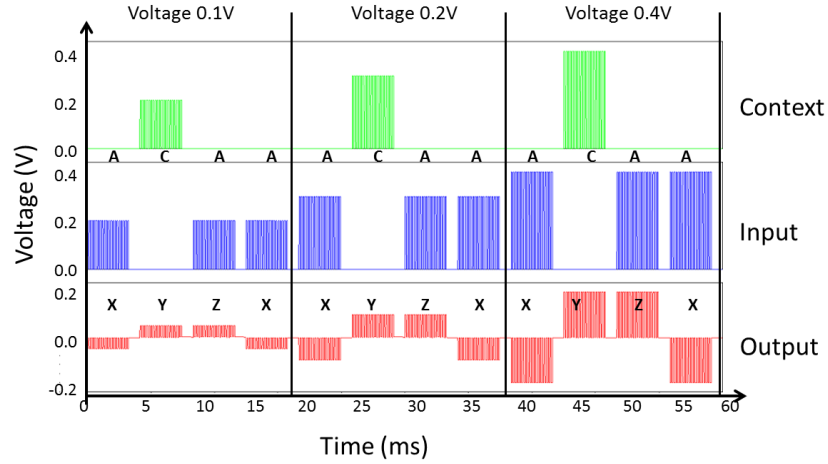
Figure 5.12: Ideal response for the three blocks of the input pattern (A C A A), with signal amplitudes of: $0.1V$, $0.2V$, and $0.4V$. The network must learn to give the correct output response (X Y Z X) for all three amplitudes. The variation-tolerant AM response, at the probe point `Output` has $6,001$ data points for fitness evaluation of candidate networks. The data points are sampled every $0.01ms$ between $0ms$ and $60ms$.

Results for voltage-tolerant AM design

C3EA evolved 100% signal amplitude variation-tolerant networks with the context-sensitive AM functionality as shown in Figure 5.13. With 13 components and a maximum component count of 800, the cost of the evolved circuit size evaluates to 1.625×10^{-2}. All the best-evolved designs from the five GP runs evolved with distant points connected that created feedback in the design.
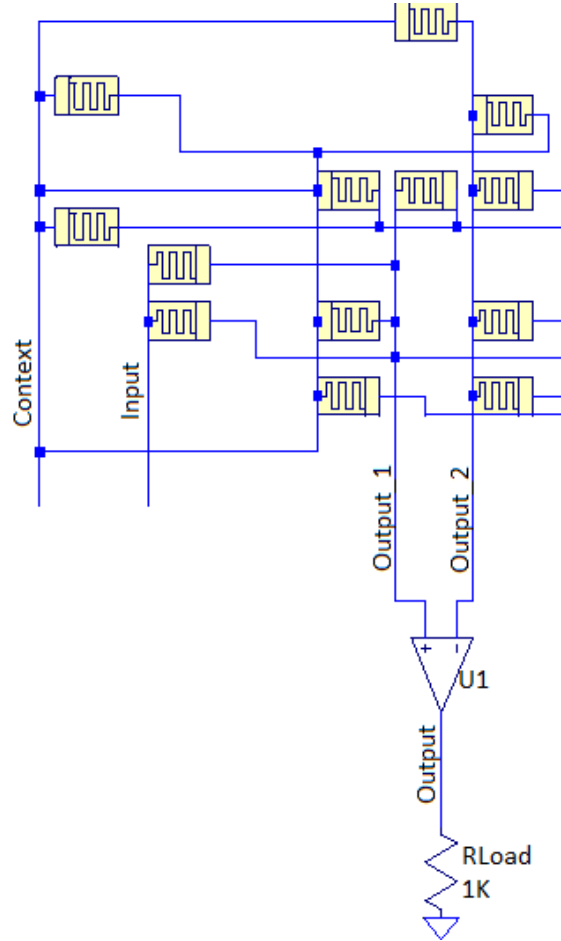
Figure 5.13: The best-evolved circuit for context-sensitive AM design that could tolerate 100% variation in signal amplitude. It comprised of 13 memristors. The evolved circuit had some bridging between distant points that created feedback in the design.

Figure 5.14 shows the transient response of the best-evolved equivalent circuit from Figure 5.13. For this circuit, the normalized squared-error over $6,001$ data points was 0.16. The final assigned normalized *fitness value*, calculated from equation 5.1, was 0.088. The evolved circuit had some bridging between distant points that created feedback in the design. These distant connections that create feedback may be responsible for the voltage tolerance.

Figure 5.14: The transient response of the best-evolved context-sensitive AM network tolerant of 100% variation in the signal amplitude. The input pattern (A C A A) was presented in three blocks with varying amplitudes of: $0.1V$, $0.2V$, and $0.4V$. The network gave the correct output response (X Y Z X) for all three amplitudes. The evolved design show distant connections that create feedback in the design.

Figure 5.15 shows the best, the average, and the worst fitness averaged over five GP runs, each with a different replacement strategy. The evolutionary dynamics are similar to the results from Experiment 4 (see Figure 5.6). The worst fitness plot stays mostly in the range of 0.55-0.65, indicating that some non-executable individuals were present in each generation for all the GP runs. The best fitness of each run converged to its final fitness value around $12,000$ generations. In Figure 5.16, we plot the best fitness evolution along with error-bars every 500

generations. The error-bars represent the standard deviation among the five GP runs. All the five runs converged to a fitness value within 2% of the best fitness of all the runs. This implies that all the best-evolved individuals are very close in performance.
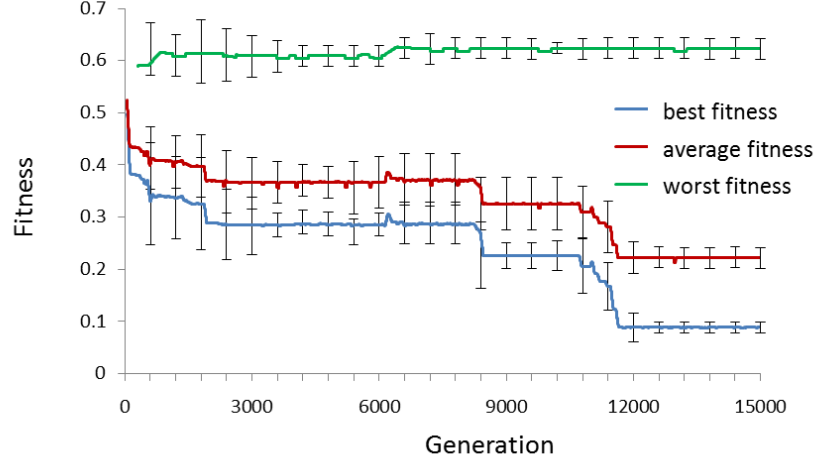
Figure 5.15: The fitness averaged over five GP runs for evolving the context-sensitive AM design, tolerant of a 100% variation in the signal amplitude.

5.4.2   Experiment 6b: Evolving frequency-tolerant AM

For this experiment, the four phases in the context-sensitive AM evaluation were extended to include the 100% signal frequency variation-checks. This is accomplished by having three blocks of four phases in the evaluation of the transient response, one block each for the signals with frequencies: $5kHz$, $10kHz$, and $2.5kHz$. Figure 5.17 shows the ideal transient response.

The nodes and control parameters are kept the same as those in Experiment 4 (see sections 5.2.2 and 5.2.4). We ran these experiments for $15,000$ generations. The embryo circuit essentially remains the same 10×10 cross-wire structure. The

Figure 5.16: The best-evolved fitness averaged over five GP runs for evolving the context-sensitive AM design, tolerant of a 100% variation in the signal amplitude. Here, the error-bars represent the standard deviation between the runs.

number of voltage sources is altered to match the input and the context signal pattern shown in Figure 5.17. The fitness measure was the same as in equation 5.1 with the normalization factor for the squared-error changed to an experimentally determined value of 2.4×10^8.

Results for frequency-tolerant AM design

C3EA evolved 100% signal frequency variation-tolerant networks with the context-sensitive AM functionality as shown in Figure 5.18. With 14 components and a maximum component count of 800, the cost of the evolved circuit size evaluates to 1.75×10^{-2}. All the best-evolved designs from the five GP runs evolved with no feedback-creating memristors.

Figure 5.19 shows the transient response of the best-evolved equivalent circuit from Figure 5.18. For this circuit, the normalized squared-error over $6,001$
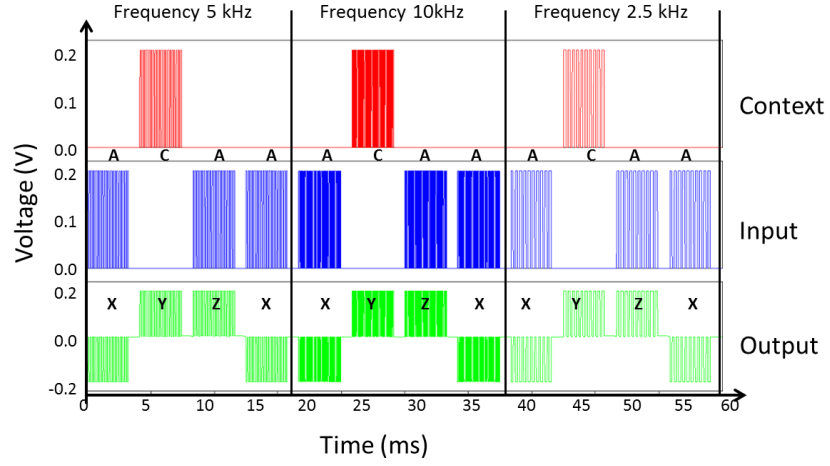
Figure 5.17: Ideal response for the three blocks of the input pattern (A C A A), with signal amplitudes of: $5kHz$, $10kHz$, and $2.5kHz$. The network must learn to give the correct output response (X Y Z X) for all three frequencies. The variation-tolerant AM response, at the probe point Output has $6,001$ data points for fitness evaluation of candidate networks. The data points are sampled every $0.01ms$ between $0ms$ and $60ms$.

data points was 0.16. The final assigned normalized *fitness value*, calculated from equation 5.1, was 3.9×10^{-3}. The evolved design was very similar to the evolved network in Experiment 4, with no observed feedbacks in the design. The best circuit from Experiment 4 was 100% frequency-tolerant (see section 5.3.3). Both of these circuits (Figures 5.4 and 5.18) show no feedback in the evolved design, we thus believe that the presence of feedbacks might be suppressing some necessary memristor state changes, and thus evolved networks with feedback were rejected during the selection process.

Figure 5.20 shows the best, the average, and the worst fitness averaged over five GP runs, each with a different replacement strategy. The evolutionary dynamics are similar to results from Experiment 4 (see Figure 5.6). The worst fitness plot stays mostly in the range of 0.5-0.6 indicating that some non-executable individuals

115
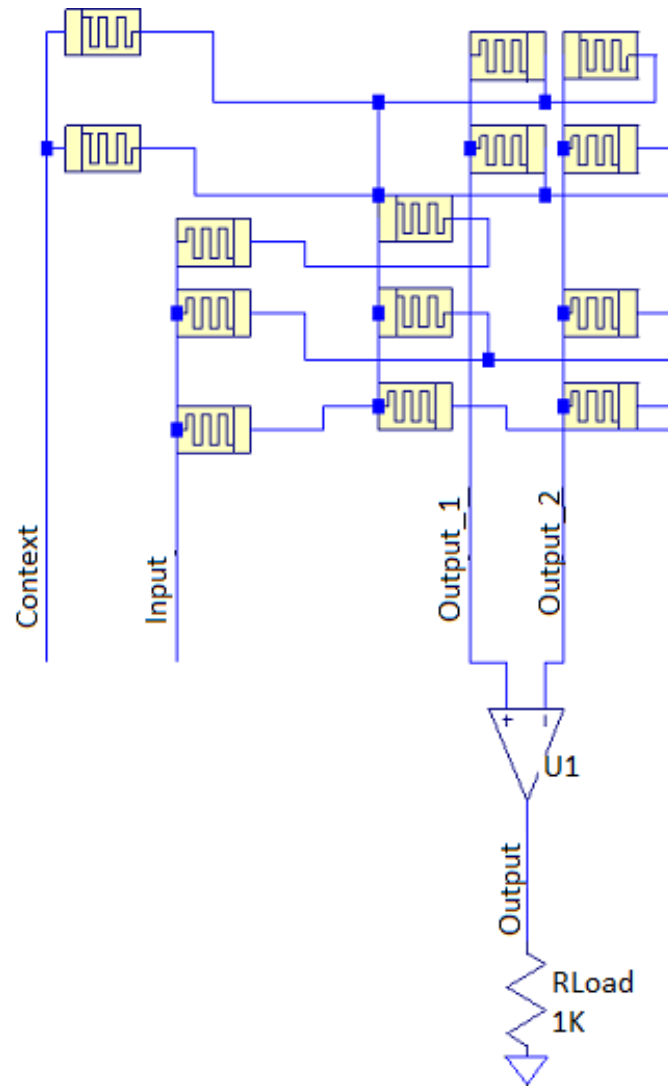
Figure 5.18: The best-evolved circuit for context-sensitive AM design that could tolerate 100% variation in signal amplitude. It is comprised of 13 memristors. The evolved circuit had some bridging between distant points that created feedback in the design.
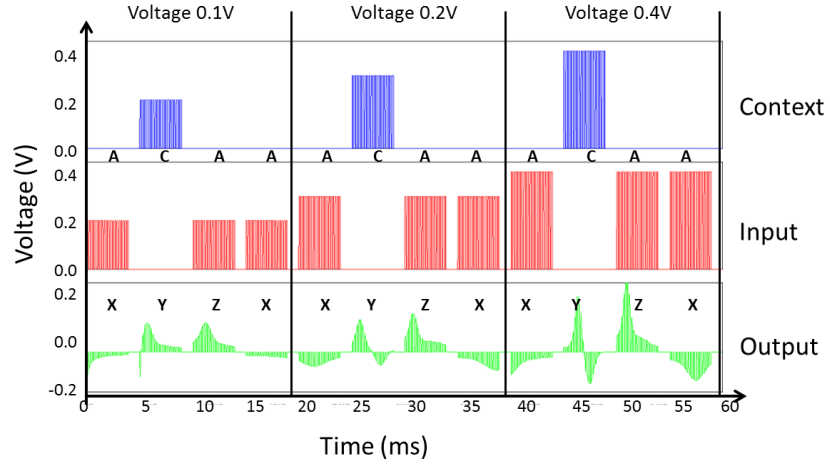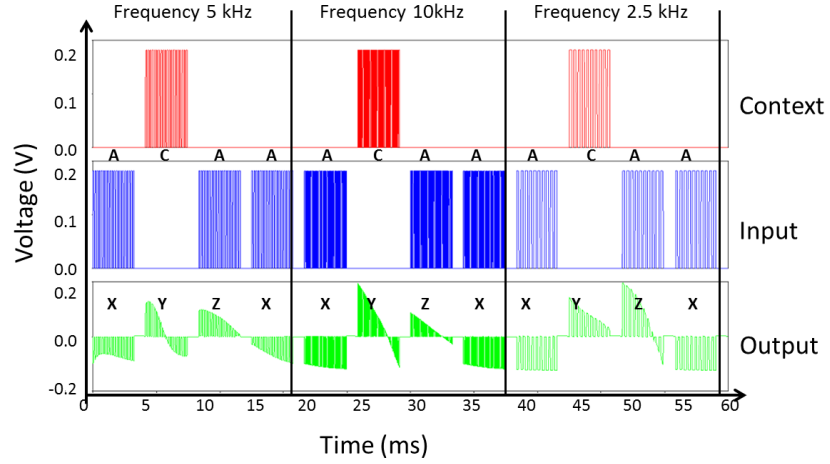
Figure 5.19: The transient response of the best-evolved context-sensitive AM network tolerant of 100% variation in the signal frequency. The input pattern (`A C A A`) was presented in three blocks with varying frequencies of: $5kHz$, $10kHz$, and $2.5kHz$. The network gave the correct output response (`X Y Z X`) for all three amplitudes.

were present in each generation for all the GP runs. The best fitness of each run converged to its final fitness value around $12,000$ generations. In Figure 5.21, we plot the best fitness evolution along with error-bars every 500 generations. The error-bars represent the standard deviation among the five GP runs. All the five runs converged to a fitness value within 3% of the best fitness of all the runs. This implies that all the best-evolved individuals are close in performance.

5.4.3 Discussion

Using C3EA, we could evolve 100% amplitude- and frequency-tolerant context-sensitive AM designs. The amplitude-tolerant design evolved with some feedback-creating memristors. This feedback were not seen in the frequency-tolerant designs. We believe C3EA, during the selection process, accepts or rejects networks with feedbacks in order to minimize the fitness function. The evolved design in each case

117
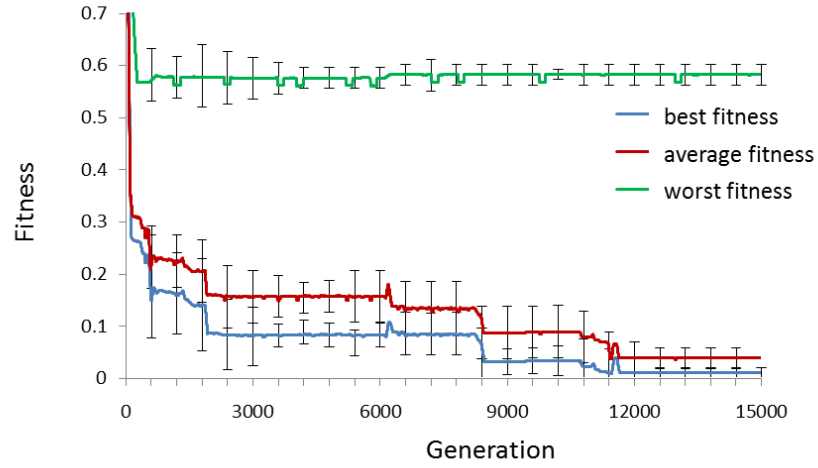
Figure 5.20: The fitness averaged over five GP runs for evolving the context-sensitive AM design, tolerant of a 100% variation in the signal frequency.
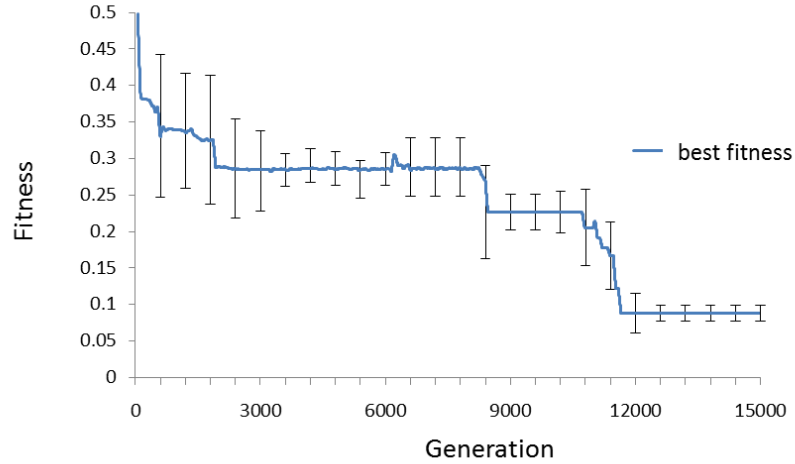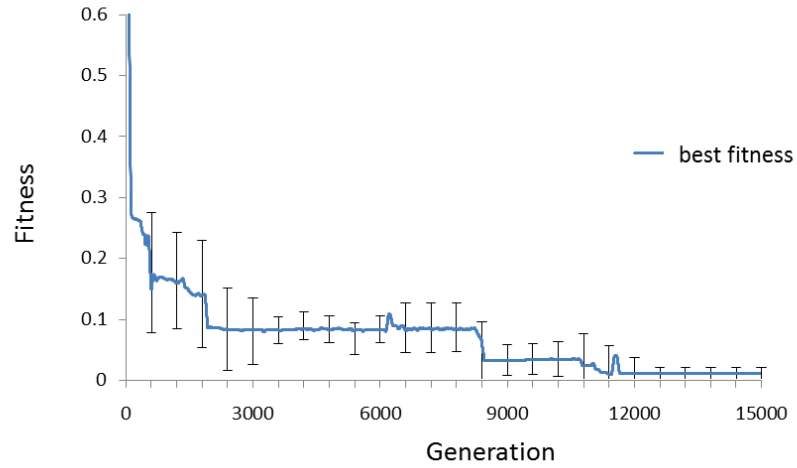

Figure 5.21: The best-evolved fitness averaged over five GP runs for evolving the context-sensitive AM design, tolerant of a 100% variation in the signal frequency. Here, the error-bars represent the standard deviation between the runs.

showed easily discernible output states that followed the target response within 95% accuracy.

5.5   Discussion

The experiments described in this chapter use an organized pattern of signals in time to associate inputs with the environment or the context. The C3EA framework is able to evolve memristor-based networks with a correct solution. C3EA was able to solve this complex task with 14–22 components because of the inherent non-linear and time-dependent properties of the memristor. The non-linearity in memristors not only allows the processing of the current set of information, but also, alters its memristance in response for the next set of information. But, because of the random nature of the circuit evolution, we do not know how the memristors encode the state.

A thorough interpretation of the C3EA evolved designs, in terms of topology and response to variation, can lead to establishing the standardized design rules for memristor based circuits. Such standardized design rules will become important in large scale integration of memristor-based architectures. Additionally, modeling variation into a design becomes more critical as computation moves into the nano-scale paradigm. A design methodology can be formalized to counter these variations. For example, we have shown that feedback in the design could help make the network more robust against amplitude variation. Similar strategies can be formalized by running C3EA on other variation related design problems.

We have shown that simple memristor-based networks are capable of solving the complex tasks like context-recognition by continuously storing and altering its states in response to the input signals presented in time. On testing the evolved

119

network against a random input stream, we realized that a sample input pattern may be sufficient to establish the correct context-recognition over an extended set of random inputs. The circuit need not be trained for all possible sequences.

6

Conclusion

In this thesis, we have closely examined the GP approach to topology generation for automated analog AM design. Methodologies have been developed individually for two kinds of AM design tasks: spatial and temporal. These AM design exploration tasks were further augmented with more refined search to overcome size, noise or variation limitations. We have also developed a circuit size vs. accuracy optimization technique and have applied it successfully in the AM design. The technique is based on weighing the trade-offs within the fitness function. Section 6.1 lists a summary of all the major contributions of this thesis, while section 6.2 enumerates some possible directions for future research.

6.1   Contributions

- GP-based C3EA framework: We have introduced C3EA, a GP-based framework for automated analog circuit design. Representing circuit solutions as trees, the topology for the first generation of circuits are generated randomly. But the values of all the components used therein are assigned from a predefined range. Subsequent solutions are generated through conventional GP reproduction mechanisms involving selection, mutation and replacement. We added five new mutation functions to increase the variation within a population, thus minimizing the chances of getting stuck in a local minima. The

randomly generated structurally incorrect circuits are assigned the worst fitness. This reduces the otherwise useless computational overhead of simulating faulty circuits. For validating C3EA, benchmark analog circuits with known solutions, like L-C-based low-pass filter and an equivalent circuit for the Hodgkin-Huxley model were synthesized using C3EA.

- Memristor-based spatial AM designs were evolved. In the absence of a solid design methodology, we show that automated circuit discovery is a promising tool for memristor-based circuits. Our results show that we can efficiently implement complex tasks, like Pavlov's classical conditioning models, with only a few memristors. We inferred from the evolved 3–5 memristor AM designs, that the rail-to-rail swing requirement in the target function ensured that bigger evolved networks were rejected during the selection process. This implied that our target function for the AM design was self-constraining in terms of size. Additionally, one of the framework parameters, w, explicitly allows the designers to explore the trade-off between smaller circuits and less noise. The evolved circuits for spatial AMs can be used as a building block for more complicated systems.

- We chose the task of context-recognition to demonstrate memristor-based temporal AM designs. The evolved designs were shown to exploit the non-linear and time-dependent property of memristors. We analyzed that feedback in the evolved-design was used by C3EA to suppress any unwanted change in the network states. One of the key observations was that, although the training sequence for context-recognition task was a short sequence of four inputs in time, the design responded correctly even when presented with a

longer sequence of random inputs. Implying that for specific tasks, a well-chosen set of sample training pattern can lead to a generalized learning of the association we are trying to accomplish. We also evolved context-sensitive AM designs that were tolerant to a 100% variation in signal frequency or amplitude.

- One drawback of our GP-based approach is that we do not understand the evolved designs completely. It is difficult to point the critical and redundant elements in the evolved design.

6.2 Future directions

- C3EA could be further optimized by re-organizing the code for GPU processor. In its present state, C3EA's bottleneck lies in interfacing with *ngspice* through files. As a next step, we would try to work on adapting the *ngspice* code to interface directly with C3EA without any file read or writes.

- We would like to explore some new hierarchical designs for multi-input spatial and temporal associations. For the high level networks, we want to eliminate *ngspice* interface altogether and move to a mathematical time-step modeling of memristor networks. The other option could be using the verilog model compiler (VMC) for preliminary fitness evaluation.

- Finally we would like to evolve modular designs that can scale and are device defect tolerant.

References

[1] J. Alspector, B. Gupta, and R. B. Allen. *Performance of a stochastic learning microchip*, pages 66–78. IEEE Press, Piscataway, NJ, USA, 1990.

[2] J. R. Anderson and G. H. Bower. *Human associative memory: a brief edition.* The Experimental Psychology Series/ Arthur W. Melton consulting ed. L. Erlbaum Associates, 1980.

[3] T. Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, 1996.

[4] J. Bailey and D. Hammerstrom. Why VLSI implementations of associative VLCNs require connection multiplexing. In *Neural Networks, 1988., IEEE International Conference on*, pages 173–180 vol.2, 1988.

[5] F. H. Bennett III, J. R. Koza, M. A. Keane, J. Yu, W. Mydlowec, and O. Stiffelman. Evolution by means of genetic programming of analog circuits that perform digital functions. In *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July*, volume 1317, pages 1477–1483, 1999.

[6] H. G. Beyer and H. P. Schwefel. *Evolution strategies: A comprehensive introduction*, volume 1. Kluwer Academic Publishers, Hingham, MA, USA, 2002.

[7] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams. 'Memristive' switches enable 'stateful' logic operations via material implication. *Nature*, 464(7290):873–876, 2010.

[8] G. Changjian, H. Hammerstrom, S. Zhu, and M. Butts. FPGA implementation of very large associative memories - scaling issues. In Ed. Amos Omondi, editor, *FPGA Implementations of Neural Networks*, Boston, MA, USA, 2003. Kluwer Academic Publishers.

[9] L. Chua. Memristor-the missing circuit element. *Circuit Theory, IEEE Transactions on*, 18(5):507–519, 1971.

[10] L. O. Chua and Sung M. K. Memristive devices and systems. *Proceedings of the IEEE*, 64(2):209–223, 1976.

[11] T. Cornforth, K. J. Kim, and H. Lipson. Evolution of analog circuit models of ion channels. In Gianluca Tempesti, Andy Tyrrell, and Julian Miller, editors, *Evolvable Systems: From Biology to Hardware*, volume 6274 of *Lecture Notes in Computer Science*, pages 157–168. Springer Berlin / Heidelberg, 2010.

[12] M. Deshpande. FPGA implementation & acceleration of building blocks for biologically inspired computational models. In *M.S. AAT 1491185*, Portland, OR, USA, 2011. Portland State University.

[13] J. M. Dolsma. *Nonlinear Controller Design based on Genetic Programming. DCT2007.107*. PhD thesis, Technische Universiteit Eindhoven, 2007.

[14] M. Duranton and J. A. Sirat. Learning on VLSI: a general purpose digital neurochip. In *Neural Networks, 1989. IJCNN., International Joint Conference on*, page 613 vol.2, 1989.

[15] E. Eiben. *Introduction to Evolutionary Computing*. Springer, Berlin, 2003.

[16] A. Engelbrecht. *Computational Intelligence*. J. Wiley & Sons, New York, 2002.

[17] V. Erokhin and M. P. Fontana. Electrochemically controlled polymeric device: a memristor (and more) found two years ago. *ArXiv e-prints arXiv:0807.0333*, 2008.

[18] K. Fukushima. A model of associative memory in the brain. *Biological Cybernetics*, 12(2):58–63, 1973.

[19] Z. Gan, Z. Yang, T. Shang, T. Yu, and M. Jiang. Automated synthesis of passive analog filters using graph representation. *Expert Systems with Applications: An International Journal*, 37:1887–1898, 2010.

[20] D. George and J. Hawkins. A hierarchical bayesian model of invariant pattern recognition in the visual cortex. In *Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on*, volume 3, pages 1812–1817, 2005.

[21] D. Hammerstrom, W. Henry, and M. Kuhn. The CNAPS architecture for neural network emulation. *Parallel Digital Implementations of Neural Networks*, pages 107–138, 1993.

[22] D. Hammerstrom and M. S. Zaveri. Prospects for building cortex-scale CMOL/CMOS circuits: A design space exploration. In *NORCHIP*, pages 1–8, 2009.

[23] J. Haugeland. *Artificial Intelligence*. MIT Press, Cambridge, 1985.

[24] D. O. Hebb. *The organization of behavior*, pages 43–54. MIT Press, Cambridge, MA, USA, 1988.

[25] A. Hodgkin and A. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Bulletin of Mathematical Biology*, 52:25–71, 1990. 10.1007/BF02459568.

[26] M. Holler, S. Tam, H. Castro, and R. Benson. An electrically trainable artificial neural network (etann) with 10240 'floating gate' synapses. In *Neural Networks, 1989. IJCNN., International Joint Conference on*, pages 191–196 vol.2, 1989.

[27] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[28] T. Hylton. Systems of neuromorphic adaptive plastic scalable electronics (SyNAPSE) program. url: http://www.darpa.mil/, 2011.

[29] A. Johannet, L. Personnaz, G. Dreyfus, J.-D. Gascuel, and M. Weinfeld. Specification and implementation of a digital Hopfield-type associative memory with on-chip training. *Neural Networks, IEEE Transactions on*, 3(4):529–539, 1992.

[30] W. Kantschik and W. Banzhaf. Linear-graph GP - a new GP structure. In *Proceedings of the 5th European Conference on Genetic Programming*, EuroGP '02, pages 83–92, London, UK, 2002. Springer-Verlag.

[31] M. Karlheinz. Brain-inspired multiscale computationin neuromorphic hybrid systems. url: http://brainscales.kip.uni-heidelberg.de/, 2011.

[32] T. Kohonen. *Self-Organization and Associative Memory*, volume 8. Springer-Verlag, 1989.

[33] B. Kosko. Bidirectional associative memories. *Systems, Man and Cybernetics, IEEE Transactions on*, 18(1):49–60, 1988.

[34] J. R. Koza. *Genetic programming III: darwinian invention and problem solving.* Complex adaptive systems. Morgan Kaufmann, 1999.

[35] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In *Artificial Intelligence in Design*, volume 96, pages 151–170, 1996.

[36] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. Four problems for which a computer program evolved by genetic programming is competitive with human performance. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 1–10, 1996.

[37] J. R. Koza, F. H. Bennett III, D. Andre, M. A. Keane, and F. Dunlap. Automated synthesis of analog electrical circuits by means of genetic programming. *Evolutionary Computation, IEEE Transactions on*, 1(2):109–128, 1997.

[38] M. Laiho and E. Lehtonen. Arithmetic operations within memristor-based analog memory. In *Cellular Nanoscale Networks and Their Applications (CNNA), 2010 12th International Workshop on*, pages 1–4, 2010.

[39] A. Liefooghe, L. Jourdan, and E. G. Talbi. A Unified Model for Evolutionary Multiobjective Optimization and its Implementation in a General Purpose Software Framework: ParadisEO-MOEO. url: http://paradiseo.gforge.inria.fr/. Rapport de recherche RR-6906, INRIA, 2009.

[40] R. F. Lyon and C. Mead. An analog electronic cochlea. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 36(7):1119–1134, 1988.

[41] S. Mackie and J. S. Denker. A digital implementation of a best match classifier. In *Custom Integrated Circuits Conference, 1988., Proceedings of the IEEE 1988*, pages 10.4/1–10.4/4, 1988.

[42] J. McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon. A proposal for the Dartmouth summer research project on artificial intelligence, august 31, 1955. *AI Magazine*, 27(4):12, 2006.

[43] T. M. Mitchell. *Machine Learning*. McGraw-Hill series in computer science. McGraw-Hill, 1997.

[44] P. Nenzi. NGSPICE: mixed mode - mixed level circuit simulator. url: http://ngspice.sourceforge.net/, 2010.

[45] Y. H. Pao. *Adaptive pattern recognition and neural networks*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[46] O. V. Pascal, R. Warren, J. K. Philip, R. S. Duncan, S. Joseph, and R. W. Stanley. Writing to and reading from a nano-scale crossbar memory based on memristors. *Nanotechnology*, 20(42):5204, 2009.

[47] I. P. Pavlov and G. V. Anrep. *Conditioned Reflexes*. Dover Publications, 2003.

[48] Y. V. Pershin, S. La Fontaine, and M. di Ventra. Memristive model of amoeba learning. *Phys. Rev. E*, 80(2):021926, 2009.

[49] D. Poole. *Computational Intelligence.* Oxford University Press, Oxford Oxfordshire, 1997.

[50] A. Rak and G. Cserey. Macromodeling of the memristor in SPICE. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(4):632–636, 2010.

[51] P. D. Roberts and C. C. Bell. Spike timing dependent synaptic plasticity in biological systems. *Biological Cybernetics*, 87(5):392–403, 2002.

[52] D. E. Rumelhart and J. L. McClelland. Parallel distributed processing: explorations in the microstructure of cognition. *Foundations*, 1:358–361, 1986.

[53] S. J. Russell and P. Norvig. *Artifcial Intelligence: A Modern Approach, 2nd Ed.* Prentice Hall, Englewood Cliks, NJ, 2002.

[54] J. P. Sage and R. S. Withers. *Analog nonvolatile memory for neural network implementations*, pages 22–33. IEEE Press, Piscataway, NJ, USA, 1990.

[55] T. Saigusa, A. Tero, T. Nakagaki, and Y. Kuramoto. Amoebae anticipate periodic events. *Phys. Rev. Lett.*, 100(1):018101, 2008.

[56] A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:210–229, 1959.

[57] D. B. Schwartz and R. E. Howard. A programmable analog neural network chip. In *Custom Integrated Circuits Conference, 1988., Proceedings of the IEEE 1988*, pages 10.2/1–10.2/4, 1988.

[58] M. Sivilotti, M. Emerling, and C. Mead. *A novel associative memory implemented using collective computation*. IEEE Press, Piscataway, NJ, USA, 1990.

[59] G. Snider, R. Amerson, D. Carter, H. Abdalla, M. S. Qureshi, J. Le Andveille and, M. Versace, H. Ames, S. Patrick, B. Chandler, A. Gorchetchnikov, and E. Mingolla. From synapses to circuitry: Using memristive memory to explore the electronic brain. *Computer*, 44(2):21–28, 2011.

[60] K. Steinbuch and U. A. W. Piske. Learning matrices and their applications. *Electronic Computers, IEEE Transactions on*, EC-12(6):846–862, 1963.

[61] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[62] D. W. Tank and J. J. Hopfield. Neural computation by concentrating information in time. *Proceedings of the National Academy of Sciences*, 84(7):1896–1900, 1987.

[63] D. S. Touretzky, A. Ladsariya, M. V. Albert, J. W. Johnson, and N. D. Daw. HHsim: an open source, real-time, graphical Hodgkin-Huxley simulator. url: http://www.cs.cmu.edu/∼ dst/HHsim/, 2004.

[64] A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[65] X. Wang, Y. Chen, H. Xi, H. Li, and D. Dimitrov. Spintronic Memristor Through Spin-Torque-Induced Magnetization Motion. *IEEE Electron Device Letters*, 30:294–297, 2009.

[66] R. S. Williams. How we found the missing memristor. *Spectrum, IEEE*, 45(12):28–35, 2008.

[67] R. S. Williams. url: http://www.youtube.com/watch?v=bKGhvKyjgLY, 2010.

[68] B. Wolfgang. *Genetic programming: an introduction on the automatic evolution of computer programs and its applications.* The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann Publishers, 1998.

[69] V. P. Yuriy and D. V. Massimiliano. Experimental demonstration of associative memory with memristive neural networks. *Neural Networks*, 23(7):881–886, 2010.