NTNU
Norwegian University of
Science and Technology

Self-Modifying Dynamical Systems for Reservoir Computing

Kristian Selvik Ekle
Aleksander Skraastad

Master of Science in Informatics
Submission date: June 2017
Supervisor: Gunnar Tufte, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Biology has always been an inspiration in the quest for artificial intelligence. By combining a hierarchy of chemistry, cells, and structures, a biological system able to learn, adapt and perform complex problem solving can emerge. Reservoir Computing (RC) is a highly efficient bio-inspired technique for working with time dependent data. Reservoir computing utilises an untrained recurrent dynamical system as a reservoir of dynamics. A single readout layer can then be trained to correlate the state of the reservoir to some target value. One aspect of biological systems that are of great interest is their ability to self-organise. Self-Modifying Cartesian Genetic Programming (SMCGP) is a genetic programming algorithm that mimics these traits through self-organising topology. This thesis investigates the use of SMCGP as a reservoir of dynamics in a reservoir computing system (SMCGP-RC).

Reservoir Computing is a general framework to design, train and analyse recurrent neural networks. However, any high dimensional dynamic system that can be stimulated by input, has the right dynamic properties, and has readable state, can be used as a reservoir. This sets few constraints on possible implementations. Both digital and physical reservoirs have been widely researched, ranging from Random Boolean Networks (RBN) to a bucket filled with water. SMCGP is a graph-based genetic programming algorithm based on Cartesian Genetic Programming (CGP). Including the mathematical operators in CGP, programs produced by SMCGP contain self-modification operators. These operators enable bio-inspired qualities such as self-regulating structures, scalability, adaptation, and self-repair.

The SMCGP-RC implementation presented in this thesis utilises an evolutionary algorithm to search for SMCGP programs where the topology is continuously changing. These programs are then evaluated against common benchmarking problems. The results show that the dynamics of SMCGP programs with continuous topological changes can be harnessed for computation in an RC environment. The temporal parity experiments presented show that when output from the previous time step is made available for SMCGP programs, they can exhibit the necessary dynamic properties to solve non-linear temporal problems.

Sammendrag

Biologien har alltid vært en inspirasjon i jakten på kunstig intelligens. Ved å kombinere et hierarki av kjemi, celler og strukturer, kan et biologisk system i stand til å lære, tilpasse seg og utføre komplekse beregninger oppstå. Reservoir Computing (RC) er en svært effektiv bio-inspirert teknikk for å arbeide med tidsavhengig data. Reservoir Computing utnytter et utrent rekurrent dynamisk system som et reservoar av dynamikk. Et enkelt avlesningslag kan da trenes opp til å finne korrelasjoner mellom reservoarets tilstand og en gitt målverdi. Ett aspekt ved biologiske system som er av stor interesse er deres evne til å selvorganisere. Self-Modifying Cartesian Genetic Programming (SMCGP) er en genetisk programmeringsalgoritme som hermer etter denne egenskapen gjennom selvorganiserende topologi. Denne oppgaven undersøker bruken av SMCGP som et reservoar av dynamikk for bruk i et Reservoir Computing system (SMGCP-RC).

Reservoir Computing er et generelt rammeverk for å designe, trene og analysere rekurrente nevrale nettverk. Allikevel kan ethvert dynamisk system som kan stimuleres med data, har de riktige dynamiske egenskapene, og har en avlesbar tilstand, brukes som et reservoar. Dette setter få begrensninger for mulige implementasjoner. Både digitale og fysiske reservoar har blitt undersøkt, og omfatter alt fra Random Boolean Networks til en bøtte med vann. SMCGP er en grafbasert genetisk programmeringsalgoritme basert på Cartesian Genetic Programming (CGP). I tillegg til de matematiske operatorene i CGP, innehar programmer utviklet av SMCGP selfmodifiserende operatorer. Disse operatorene muliggjør bio-inspirerte kvaliteter som selvregulerende struktur, skalerbarhet, tilpasning til endrende omgivelser og selvreperasjon.

SMCGP-RC-implementasjonen presentert i denne oppgaven utnytter en evolusjonær algoritme for å søke etter SMCGP-programmer der topologien er i stadig endring. Disse programmene blir deretter evaluert mot kjente referanseproblem. Resultatene viser at dynamikken i SMCGP-programmer der topologien er i stadig endring, kan utnyttes i et Reservoir Computing system. Gjennom tidsavhengig paritet-eksperimentene vises det at når SMCGP-programmer blir presentert med utgangsverdiene fra forrige tidsintervall, kan de utvise de nødvendige dynamiske egenskapene for å løse ikke-lineære tidsavhengige problem.

Preface

This thesis was written by Aleksander Skraastad and Kristian Ekle as part of their masters degree in Artificial Intelligence. The work has been conducted over the semesters, fall 2016 and spring 2017, at the Department of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU).

The authors would like to express gratitude to their advisor, Professor Gunnar Tufte, for his invaluable guidance and feedback during the last year.

Contents

List of Tables

List of Figures

Abbreviations

ANN	=	Artificial Neural Network
CA	=	Cellular Automata
CGP	=	Cartesian Genetic Programming
EA	=	Evolutionary Algorithm
ESN	=	Echo State Network
GA	=	Genetic Algorithm
GP	=	Genetic Programming
IEF	=	Input Expansion Factor
LSM	=	Liquid State Machine
MEA	=	Micro-Electrode Arrays
RBN	=	Random Boolean Networks
RC	=	Reservoir Computing
RNN	=	Recurrent Neural Network
SM	=	Self Modifying
SMCGP	=	Self Modifying Cartesian Genetic Programming
SMCGP-RC	=	SMCGP Reservoir Computing

Chapter 1

Introduction

Man has always looked to nature when faced with new and unseen challenges. Be it on purpose, studying the wings of birds in the quest for human flight, or by chance, as in the famed Newtonian apple.

The field of Artificial Intelligence is no exception. Artificial Neural Networks, inspired by our brain, were originally created to understand information storage and recognition [1]. Evolutionary Algorithms [2] and its sub-field Genetic Algorithms [3] are inspired by the process of evolution and the genetic representation of DNA. Swarm Intelligence [4] is an emergent system where complex behaviour arises from decentralised, self-organising systems such as ant colonies or flocks of birds. Even though the techniques are inspired by biology, they do not necessary aim at mimicking nature. Our engineered solutions are abstractions. Such abstractions can be a product of the possibility to capture only the necessary processes of biological systems [5] or the abstraction may be a product of limitation in current technology and our understanding of the biological solutions.

As apposed to mankind's top-down approaches to engineering, biology solves its problems by using numerous seemingly autonomous parts, which together emerges as a complex dynamic system capable of complicated problem solving. Biological systems are driven by self-organisation and local interactions, which somehow manages to consolidate into a robust system capable of computation, learning and adaptation.

In an effort to understand how biological systems are able to employ these dynamic structures for computation. Researchers in the *NTNU Cyborg* initiative, [6, 7] are trying to use biological cell cultures, to create a system part digital and part biological, a *cyborg*.

Reservoir Computing [8] is a framework to design, train and analyse recurrent neural networks (RNNs)[9]. Reservoir computing today encompasses several techniques, but originates from the independent work on Echo State Networks [10], Liquid State Machines [11], Temporal Recurrent Networks [12] and Backpropagation-Decorrelation [13].

The core concept of reservoir computing is to transform data into a higher dimensional space, often by means of an RNN, and then use a separate mechanism for reading the state of this system. The *reservoir* is an enclosed dynamic system with some computational power and fully or partially readable state. The reservoir is left untrained. Instead, a separate readout layer is trained. This layer is often a simple linear regression on the readable state of the reservoir x_i, and some target y[14].

RC is a practical approach for exploiting the capabilities of RNNs, because it bypasses the complex task of training them. Instead, a separate readout layer is responsible for decoding the dynamics into something useful.

Researchers have studied the use of various dynamic systems as reservoirs. Among the studied, Celluar Automata (CA) [15] and Random Boolean Networks (RBN) [16, 17] have shown to be compatible as reservoirs, and has been frequently used in RC research. Like biological systems, both CAs and RBNs are non-linear dynamic systems which rely on local interactions. However, they do not display the structural plasticity which biological systems do. CAs and RBNs are sparsely connected networks. In principle, a vast number of nodes with a limited number of connections. A graph representation can be an abstract form with sparsely connected network properties.

Self-modifying Cartesian Programming (SMCGP) [18] is such a sparsely connected graph with the ability to modify its own structure. SMCGP is a specialised form of *Cartesian Genetic Programming (CGP)* [19], where programs are allowed to transform employing self-modification operators. Programs in SMCGP are directed acyclic graphs, where nodes are either input nodes, mathematical functions or self-modification nodes. The area of interest for this thesis will be to use SMCGP programs with continuously altering topology as the reservoir in an RC system.

SMCGP programs are non-linear dynamic systems, which, comparable to biological systems, have continuously altering topology due to their self-modification operators. Making them suitable candidates to model biological structures.

The research presented in this thesis, investigates the viability of using the previously unexplored combination of SMCGP and reservoir computing (SMCGP-RC). An Evolutionary Algorithm is used to generate SMCGP programs that exhibit desired dynamic behaviour. These programs are used in a series of experiments to assess their capability for complex computation. The main purpose is to investigate whether it is possible to get reliable computation from these complex and potentially chaotic networks, thus being suitable as a reservoir of dynamics in RC systems. Furthermore, the able programs will be analysed to see if there is a possible correlation between computational capability and program characteristics.

This thesis assumes that the reader is familiar with traditional feed-forward artificial neural networks. Other necessary topics, are explained in chapter 2.
This thesis is divided into seven chapters including the introduction:

Chapter 2: Provides a summary of the literature used in this thesis. This includes previous work for the most central topics.

Chapter 3: Detailed description of how the research was conducted. This chapter introduces three research questions, and the steps taken to investigate these questions.

Chapter 4: Contains the technical aspects of SMCGP and the RC system, the architecture of the implemented experimental platform as well as what design decisions and assumptions were made.

Chapter 5: Presents experiment configuration and results for the temporal parity, temporal density and NARMA10 benchmarking problems.

Chapter 6: Discussion of the experiment results, in relation to theory and by analysing characteristics of individual SMCGP programs.

Chapter 7: Conclusion and evaluation of the work presented, as well as what contributions it has resulted in. A brief summary of what is considered the most interesting future work is included.

Chapter 2

Background

This chapter introduces the topics and previous research used in this thesis.

2.1 NTNU Cyborg

The work presented in this thesis is related to a larger research initiative, NTNU Cyborg. The Cyborg project aims to get a better understanding of the human brain by creating a cyborg, a combination of machine and biological tissue. As stated in [6]:

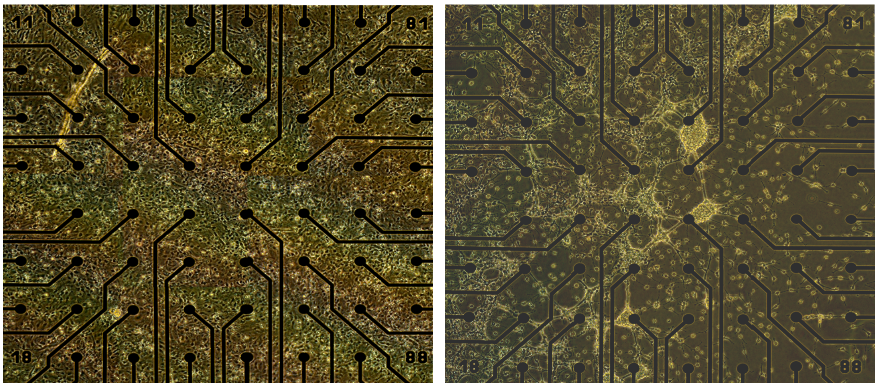
> Through The NTNU Cyborg initiative, a cybernetic (bio-robotic) organism is currently under development. Using neural tissue, cultured on a micro-electrode array (MEA), the goal is to use in-vitro biological neurons to control a robotic platform.

The research team of the Cyborg project has recently, successfully grown living cultures of rat stem cells *in-vitro* [1] over *Micro-Electrode Arrays* (MEA) [6]. These living cell cultures can be stimulated with electrical impulses and the state or potential of the Micro-Electrode Arrays nodes can be recorded. The Cyborg project aims to incorporate these cell cultures in a robot.

The goal is to use the cell cultures in an RC system. The setup would feed sensory data from the robot as electrical signals into the MEA and send the output through a readout network. The final output would then determine what action the robot should perform.

Figure 2.1a and Figure 2.1b depicts rat stem cells grown on top of an MEA. The two figures show the same cell culture at different times, two days after seeding and eight days after seeding, respectively. The image showing the culture after two days has little to no

[1] *In-vitro* (in a glass), means that biological tissue or cells are grown in a laboratory, outside the organism. *in-vivo* refers the opposite. *Source: oxforddictionaries.com*



(a) Cell culture on MEA, 2 days old **(b)** Cell culture on MEA, 8 days old

Figure 2.1: Cell culture on MEA

visual structure between the cells, while the image of the culture after eight days shows that visible structures have formed. Due to their plasticity, the networks formed within the cell cultures will continue to change over time [20, 21].

The motivation for the Cyborg Project is to acquire insight in the biological brain on a neuronal level. As stated in [7], achieving the goals set in the Cyborg Project would lead to significant advances in:

Technology: *bio-artificial computers, cyborg technologies, brain-machine interfaces*

Biology: *regenerative neuroscience, understanding development and repair processes neurodegenerative disease, nano-medicine*

Philosophy: *issues around neuronal functions, studying the mechanisms of; memory, learning, concept formation*

2.2 Dynamical Systems

A dynamical system is an abstract mathematical model that allows for analysis of the state space a system can inhabit. It is a system of equations, which in theory, can predict the future state, given the past [22]. A dynamic system is a system that evolves over time [23]. Dependant on the behaviour of said system, it might settle into stability, go into periodic trajectories or sheer chaos [24].

2.2.1 Attractors

An attractor is one or more states that a system tends to converge on as time$\rightarrow \infty$ if the initial conditions of the system are within the *basin of attraction* (see Figure 2.2)[22, 24]. What state(s) the system converges on are dependent on the properties of the system, the initial conditions and how the system is stimulated.

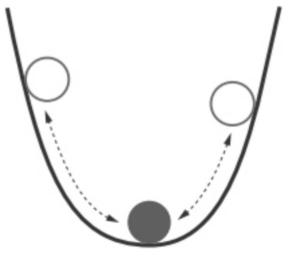


Figure 2.2: Same attractor for different initial conditions

There are three main categories of attractors [22, 24, 25, 26].

Point attractors: Systems that gravitate toward a single state are known as point attractors.

Limit-Cycle attractors: Systems with limit-cycle attractors periodically revisit two or more states.

Chaotic attractors: Systems with chaotic attractors have a sensitive dependence to initial conditions and traverse the state space in aperiodic trajectories.

2.2.2 Computation at the Edge of Chaos

Dynamic systems theory is a field intersecting various disciplines and has been highly researched. Much of the research available from a computational perspective is concerned with systems displaying dynamic behaviour at the border between stability and chaos [24].

It was Wolfram [27] who conjectured that cellular automata (CA) with turing complete capabilities exhibited special dynamic behaviours. In his work he found that all one-dimentional CA can be divided into four classes based on their dynamic behaviour (classes displayed in Figure 2.3) and that the fourth class represented the behaviour capable of complex computation.

In [28], Langton confirms Wolfram's conjecture, and further identifies that systems with dynamics in the transition between *highly ordered* and *highly disordered*, at the critical point of the phase transition, have the best capabilities for complex behaviour. Further, theoretical analysis has indicated that the "edge of chaos" is the effective region of computationally powerful dynamic systems.

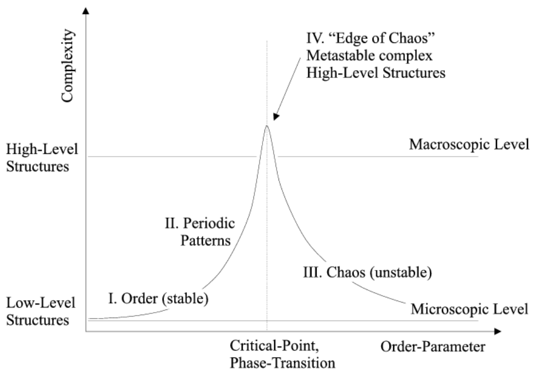


Figure 2.3: Edge of Chaos

Figure 2.3 shows the four classes described by Wolfram. Langton correlated the states introduced by Wolfram to an order parameter, λ, where $\lambda = 0$ corresponds to class 1 (stable), $\lambda = 1$ compares to class 3 (chaotic). Further, he found that λ somewhere between 0.45 and 0.5 was the critical point in the phase transition [28].

Stuart Kauffman, in his work with Random Boolean Networks (RBN) concurred with Wolfram and Langton, that computation lies at the ordered regime of chaos [29]. Random Boolean Networks were developed by Kauffman as a way to study genetic regulatory networks[16]. RBNs (*Kauffman-networks*) are networks consisting of N nodes, were each node is connected to k randomly chosen neighbouring nodes. All nodes have a boolean state which is either "one" or "zero". Each node in the RBN network performs a randomly selected boolean operation which when initiated, propagates through the network and updates the state of the nodes. Together the nodes in the network make up a discrete dynamic system [17]. RBNs are a generalization of Boolean cellular automata, often used as a model when researching dynamic systems [30].

2.3 Recurrent Neural Networks

Recurrent neural networks are an extension of the traditional feed-forward artificial neural networks (ANN). A regular ANN is a directed acyclic graph (DAG). RNNs on the other hand, contain cycles due to recurrent nodes [31]. Figure 2.4 is a simple representation of the differences between a feed-forward neural network and a recurrent neural network.

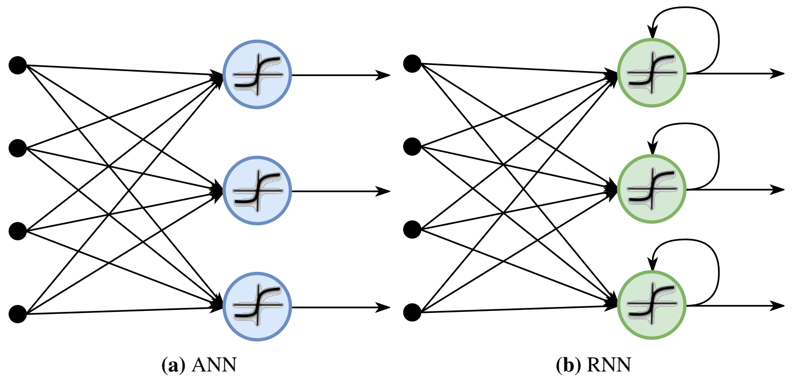


Figure 2.4: ANN and RNN distiction

The motivation for developing RNNs was rooted in the limitations of traditional ANNs. ANNs are a very powerful tool for learning the features of a data set, as long as it is solvable in a static (non-temporal) context. However, ANNs are not well suited for dealing with temporal problems [14]. Temporal problems are dependent on preceding input to predict the current output. Solving such problems requires a system that has some memory or feedback, which feedforward ANNs do not possess.

In addition to all incoming edges a regular ANN has, RNNs also have a recurrent connection pointing back on itself. These recurrent connections provide RNNs with memory of preceding states, a residual signal. The recurrent connection is usually weighted by a global decay or growth factor, U. This enables the RNN to find temporal correlations across time-dependent data [32].

Figure 2.5 shows an RNN node unrolled across multiple time steps. As illustrated, a node in an RNN is a recursive function. For each time step t, the output h_t is dependant on the current input x_t and the previous step $t-1$.

Many real world problems require temporal information to solve. Weather prediction, adaptive filtering, noise reduction, vision and speech processing and much more. These problems cannot be solved by traditional non-dynamic feed forward systems such as traditional ANNs[14].

RNN is a powerful tool for these types of problems. But in practice, they are hard to work with. RNNs have proven to be notoriously challenging and time-consuming to train as the

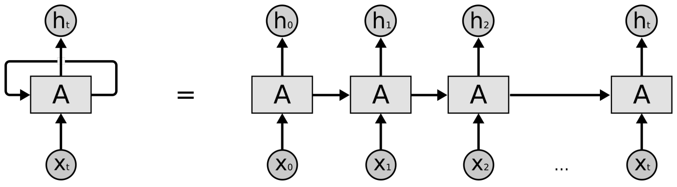


Figure 2.5: Unfolded recurrence of RNN node

networks grow beyond a couple of layers [14, 33, 34].

The reason for RNNs being so hard to train is due to a phenomenon called *vanishing and exploding gradient (vanishing gradient)* [32, 33]. The phenomenon is due to the relationship between parameters and hidden states during RNN training. The result is exponential growth in either the long term or short term error signals, where one is dominating the other. The outcome is that the RNN is not able to correlate events close to, or distant from each other in time [32]. This means that traditional ANN training methods such as *backpropagation* with *gradient descent* are not very practical for deep RNNs.

New, more recent ways to train RNNs have been developed to combat the difficulties explained. Some examples that frequently appear in the literature are:

- In [35] an approach called *Long Short-term Memory*, changes the architecture of the RNN by introducing memory cells using logistical gates to determine when and for how long information should be stored.
- *Hessian Free Optimiser* [36] is able to avoid the vanishing gradient problem by better detecting small gradients and curvatures.
- *Echo State Networks* [10], later absorbed under the Reservoir Computing umbrella, avoids altering the RNN at all. Instead, it uses a separate readout layer that is trained by reading the untrained state of the fixed RNN. [10, 14]. Further explained in section 2.5.
- *Backpropagation-Decorrelation* [13], also an RC technique, uses the same principles. Here, only one backpropagation step is used, and memory is based on the temporal dynamics of the network by means of decorrelation of activations.

2.4 Evolutionary Algorithms

Evolutionary Algorithms, known as EAs, is a field of artificial intelligence that focuses on exploring a solution space by mimicing the biological process of evolution[37]. In short, these types of algorithms start off with a set of randomly generated solutions to a given problem, stores the best ones, applies some mutation operation and repeats the selection. These steps are repeated until an acceptable error rate is achieved or the program reaches a predefined number of iterations, know as epochs [37, 38].

EAs search through a problem space and finds a solution by balancing *exploration* and *exploitation*[39]. Exploration is achieved by introducing randomness, pushing the search in new unexplored directions, while exploitation refers to retaining good solutions and making them more precise. The two qualities need to be balanced for the search to be optimal. In EA, candidate solutions are normally called phenotypes and are encoded by a set of genes called the genotype. A specific development function is required to create a phenotype from a genotype. One could say that the genotype is the blueprint or DNA while the phenotype is the living individual.

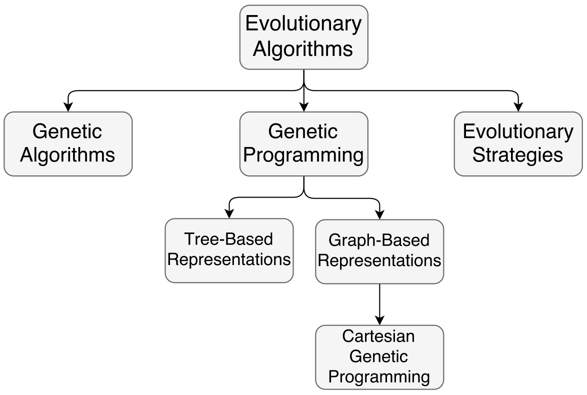


Figure 2.6: Taxonomy of a few types of Evolutionary Algorithms

EA is a collective term describing search techniques inspired by biological evolution[40]. Figure 2.6 shows a taxonomy of a few evolutionary algorithm types. They all rely on evolution, but differ in genetic representation and implementation details [38].

The particular type of EA used in this thesis is called Cartesian Genetic Programming (CGP). CGP is a sub-field of Genetic Programming (GP)[41]. In GP the genetic representations are usually data structures like trees or graphs, while in GA and ES they are usually represented by sequences of binaries, integers or real values[40]. Figure 2.7 shows how the genotypes and phenotypes in CGP are represented.

Before utilising any form of EA on a problem, one has to figure out how to represent the solution as genotypes and phenotypes. In the case of CGP and other algorithms which require graph structures, the genotype must encode how many nodes the graph consists

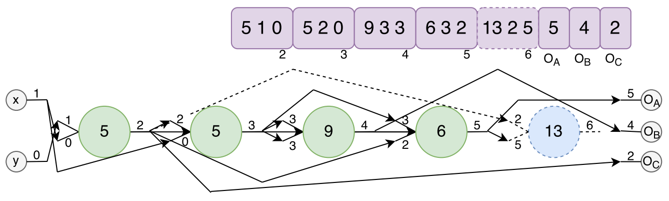


Figure 2.7: Genotype (top) and Phenotype (bottom) in CGP, from [41]

of, which nodes are connected to each other, and all the necessary information needed to construct the graph. For example in the travelling salesman problem, the genotype could be an array of integers where the ordering of the integers represents the order of cities visited[42].

As depicted in Figure 2.7, in CGP the solutions are represented as directed acyclic graphs consisting of computational nodes. Each section of the genotype encodes a node in the graph, while the last sections are translated to outputs[41].

To evaluate how good a solution is, a scoring function is used. This function is called the *fitness function* and it is responsible for guiding the search in the right direction. When all phenotypes have received a fitness score, a parent selection is conducted[38]. There are numerous strategies on how to select the phenotypes whose offspring will make up the next generation of solutions. The methods differ in the range from totally random to always select the solutions with the best fitness. Choosing parents at random will introduce just exploration in the search. This might lead to never finding an optimal solution in a reasonable amount of time. This brute-force style search lacks exploitation, preventing it from investigating similar phenotypes further. On the other hand, always selecting the best individual might end up in a local maximum by the lack of exploration.

After parent selection, new solutions are created from the parent phenotypes. Usually, children are a copy of their parents with some slight change (mutation) or a combination of the parents (crossover). Both techniques can be used in unison and provide enough randomness to explore new areas in the problem space, while still retaining prior good solutions. For instance, if the phenotypes are modular in nature, crossover might merge one or more high-value sections of the parent genotypes to form a superior one.

2.5 Reservoir Computing

Reservoir computing is a framework that facilitates computation through an intrinsic reservoir of dynamics, that can be perturbed (stimulated) by input. The reservoir can be seen as a temporal kernel function, which transforms input into a higher dimensional space. It aims to combat the limitations of RNNs on non-linear modelling, where RNNs have had slow progress [43].

Normally, an RC system is created by generating a random, fixed RNN to be used as a reservoir. Then, a separate readout layer is trained. Normally by means of a linear regression of the reservoir state and some target value. However, any dynamic system that has fading memory and can separate input streams should, in theory, be usable as a reservoir [34]. Some examples are elaborated in subsection 2.5.1.

The two main advantages of RC, are being able to make use of a wide variety of innate dynamic systems, and that training the readout layer is computationally cheap. This makes RC a very interesting approach when combined with physical reservoirs that might be faster or cheaper than their digital counterparts.

As briefly explained in chapter 1, the independent work on ESNs and LSMs formed the basis for reservoir computing. ESNs were initially proposed as an alternative RNN approach. They were used with success for control tasks, as waveform recognisers, word- and pattern recognizers to name a few [10], while LSMs were used as a model for generic cortical microcircuits [44].

In general, RC aims to overcome the main disadvantages of traditional RNN training methods, such as slow convergence, learning disruption caused by bifurcations[2], difficulty learning long temporal dependencies due to vanishing or exploding gradient, as well as requiring experience for tuning the many parameters [43].

Both Jaeger [10] and Maas [26] stated that ESNs and LSMs should be scaled to the edge of stability for optimal performance.

For a dynamic system to be suitable as a reservoir, it has to have what is known as *fading memory* and the *separation property*. To be suitable for computation, the reservoir must eventually forget past pertubations, while still being able to separate two distinct input streams [45]. Reacting differently for two distinct input streams is important for short-term dependencies, and is covered in the separation property, while fading memory is necessary for long-term dependencies and the system's ability to generalise [46]. These two are opposing effects, and Natschläger et al. [45] found that systems with the greatest difference in separation and fading memory happened to coincide with dynamics at the edge of chaos (critical dynamics).

Figure 2.8 shows a schematic of a traditional RC system. Input enters from the left, the dynamics in the middle transform the data and the readout layer on the right decodes it.

[2]A *bifurcation* is when a smooth gradual change to some parameters, called the *bifurcation parameters*, causes an abrupt topological or other qualitative change to the system.

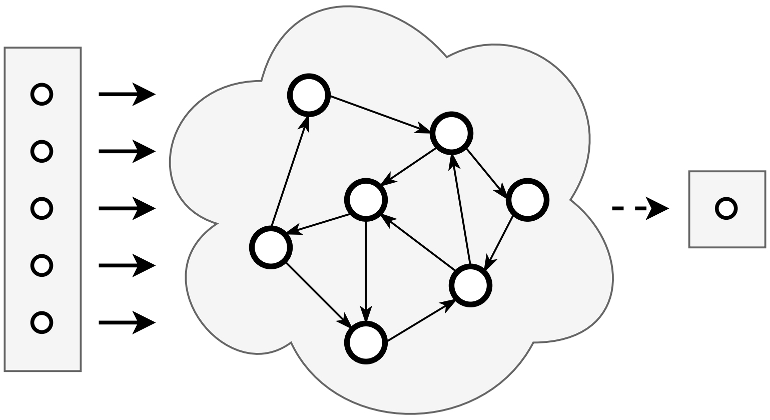


Figure 2.8: Reservoir Computing

2.5.1 Previous Work

RC has become a widely researched topic. Much of the literature has been focused on identifying characteristics of well-performing reservoirs and performance measures. A tuned reservoir should, after all, perform better than a randomly generated one. Expansions to the general idea, by exploring different reservoir and readout types have also seen a large amount of research.

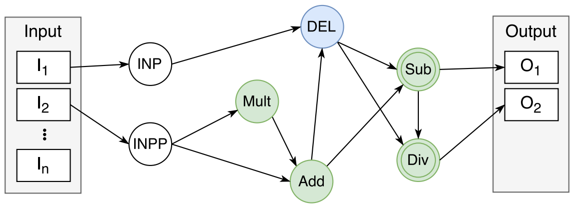
Traditional reservoir computing has done well in digital signal processing (such as speech recognition and noise modelling), chaotic time series prediction, dynamic pattern classification, and autonomous sine generation [14]. The RC approach in many cases consistently outperforms state-of-the-art RNN training methods [43].

Perhaps one of the most interesting tendencies within RC is using analog structures or mediums as reservoirs. In [47] the authors successfully implemented a reservoir system that could distinguish the words ”zero” and ”one” using a bucket filled with water as the reservoir. A linear readout layer and the waves of the water were used to classify the data. The research in [48] studies using the visual cortex of cats as a reservoir to get information of various visual stimuli. Other research show using radiation, electric signals and bacteria as reservoirs[49].

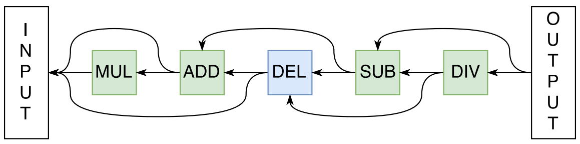
2.6 SMCGP

Self-modifying Cartesian Genetic Programming is an expansion of the field CGP [41, 50]. CGP and SMCGP are similar, but SMCGP contains special self-modifying nodes which alter the program during execution. Like CGP, SMCGP is defined as a GP algorithm and is driven by evolution [18]. As EAs in general, SMCGP generate different solution candidates within the problem space for a given task, which are iteratively closer to the optimal solution.

The solutions in CGP and SMCGP, the phenotypes, are directed graphs. Where regular CGP only consists of nodes with mathematical operators, SMCGP is expanded with self-modifying operations [18]. The genotype encodes all the nodes in the graph as fixed length sections of the genotype containing connection information, as well as which function it represents and other parameters.



(a) SMCGP Graph



(b) SMCGP Schematic

Figure 2.9: SMCGP Program

Figure 2.9 illustrates a simple program produced by SMCGP. The example has two outputs, which can be decoded to the following two equations:

$$\frac{input_i + (input_i * input_i) - input_i^{delete}}{input_i^{delete}} \tag{2.1}$$

$$input_i + (input_i * input_i) - input_i^{delete} \tag{2.2}$$

When executing an SMCGP program, passing one data sample through the graph is called an *iteration*. This involves reading the value of one or more output nodes, which triggers a recursive read on all the connected nodes in the graph. Recursion is terminated at input nodes, when attempting to connect beyond the graph (which returns zero), or when reading from a constant node.

All inputs to an SMCGP graph are passed through *input functions*[51]. When executing an SMCGP program, an internal input pointer is stored. Each call to an input function will return the input value at the current pointer and based on the input function, either *increment, decrement or skip/stride* the pointer. This means that each successive input variable in Equation 2.1 and Equation 2.2 represent different values. For Figure 2.9 two outputs result in calling an input function 9 times. If the input pointer points out of bounds it will wrap around, pointing at the first input again[52].

Contrary to CGP, SMCGP graphs are dynamic. Nodes are indirectly connected to each other through *relative addressing*[50, 51]. Each node in a graph has internal connection values dictating how far to the left they should connect. The value has to be larger than zero to avoid cyclic connections. This means that the graph is not required to self-assemble after a modification, because all edges are relative to the absolute position of the node. This simplifies moving or duplicating sub-graphs, and reduces the computational footprint.

Not all nodes in the graph are active at all times. Due to the connection pointers of each node, some nodes are skipped and are therefore disconnected from the output node spanning trees. Which nodes are active in a phenotype may change after each iteration. Meaning that the same genotype can encode a multitude of different phenotypes[53]. The disconnected nodes in a phenotype may become active at a later point in time. These ”non-coding” genes may still carry mutations from previous iterations, and give rise to interesting dynamics when they occasionally become active.

In standard SMCGP, creating a phenotype that is able to solve the problem at hand is done by an evolutionary process. An incremental fitness evaluation is performed in each EA epoch, where each successive epoch increases the number of iterations of the phenotype. Unwanted phenotypes are often killed off immediately by being given a harsh fitness penalty. This ensures that the phenotypes in the population have at least managed to give acceptable output up until the present epoch/iteration.

The main difference between SMCGP and CGP is the **DEL** node in the middle of the Figure 2.9. It is denoted $input_i^{delete}$, where $_i$ represents the current input pointer. The ^{delete} notation indicates that this might trigger a delete operation. If the **DEL** node is activated it will alter the phenotype by removing node(s) based on its internal parameters[54]. Self-modification nodes always return the value of their 2nd incoming connection, which in this case is an input node. Whether or not they are activated is determined by their incoming connections. If the return value from connection $i < j$, then the node is considered to be active and added to what is called a *Todo-list*. This list is controlled by a maximum size parameter, limiting the amount of modification to the phenotype per iteration[51]. The Todo-list is executed after the value of all output nodes have been determined.

One important thing to note is that the SM alterations are online, meaning that the changes happen real time and on the phenotype only, while the genotype (which is used to generate offspring) is unaffected. A consequence of this is that it is possible to evolve SMCGP solutions which during their lifespan evolves to read varying amount of inputs and outputs. This means that within the solution space of a given problem, there might exist a solution which iteratively can expand itself to a general solution. An example of this is the n bit parity solution reviewed in [18], where the best solutions could solve all parity problems up to 22 bits. See chapter 4 for the specific SMCGP implementation used in this thesis.

2.6.1 Previous Work

Much of the previous work on SMCGP revolves around showing its potential in evolving circuits and solving numerical approximation or series generation problems. In [54] the authors are able to evolve a SMCGP-graph to a general solution for the n bit parity problem using only boolean and SM operators. [53] shows SMCGP outperforming CGP when generating *Fibonacci sequences, square sequences, sum of inputs*, and *power function*, evolving general solutions for all of them. In [55] SMCGP is used to find arbitrary amounts of decimals of both e and π. In [52] SMCGP tackles the French flag problem, where it is successful in approximating the solution.

A newer version of SMCGP was introduced in [56]. Here the programs are represented by a 2D grid system, like what is used in CGP. The grid is generated based on two configurable parameters *height* and *width*[57]. In [50] SMCGP2 is shown to be faster on the even parity and binary addition tasks than standard SMCGP, as well as having a simplified function set. Based on available material and to reduce complexity in the experimental setup, standard SMCGP is used in this thesis.

Chapter 3

Methodology

This chapter describes a set of research questions as well as experiments conducted in order to investigate these questions. The goal of this thesis is to evaluate the use of SMCGP programs as reservoirs in an RC system. An experiment platform was therefore devised to investigate the following research questions:

1. Is it viable to use SMCGP programs with continuously altering topology as a reservoir in a Reservoir Computing system?
2. What, if any, are the characteristics of SMCGP programs that are suitable as reservoirs?
3. How does SMCGP programs with continuously altering topology perform on the temporal parity, temporal density and NARMA10 benchmarking problems?

3.1 Experimental platform

Figure 3.1 shows an overview of the experimental platform. It is built from three principal components, where **A** is an evolutionary search module, evolving SMCGP programs as specified in [18, 41]. **B** and **C** represent an RC system (reservoir and readout) based on the general description in [43]. The SMCGP was implemented from the bottom up, as it granted full control of all parameters and implementation details, like being able to record all internal processes. For the readout layer an existing neural network library was chosen, this allowed the readout layers to be configured with a variety of tried and tested updaters, loss functions and activation functions. As SMCGP does not possess intrinsic memory or feedback in the general sense (see detailed explanation in section 4.2), output from time step $t - 1$ was made available to the SMCGP programs, providing the necessary memory for performing time-dependent tasks.

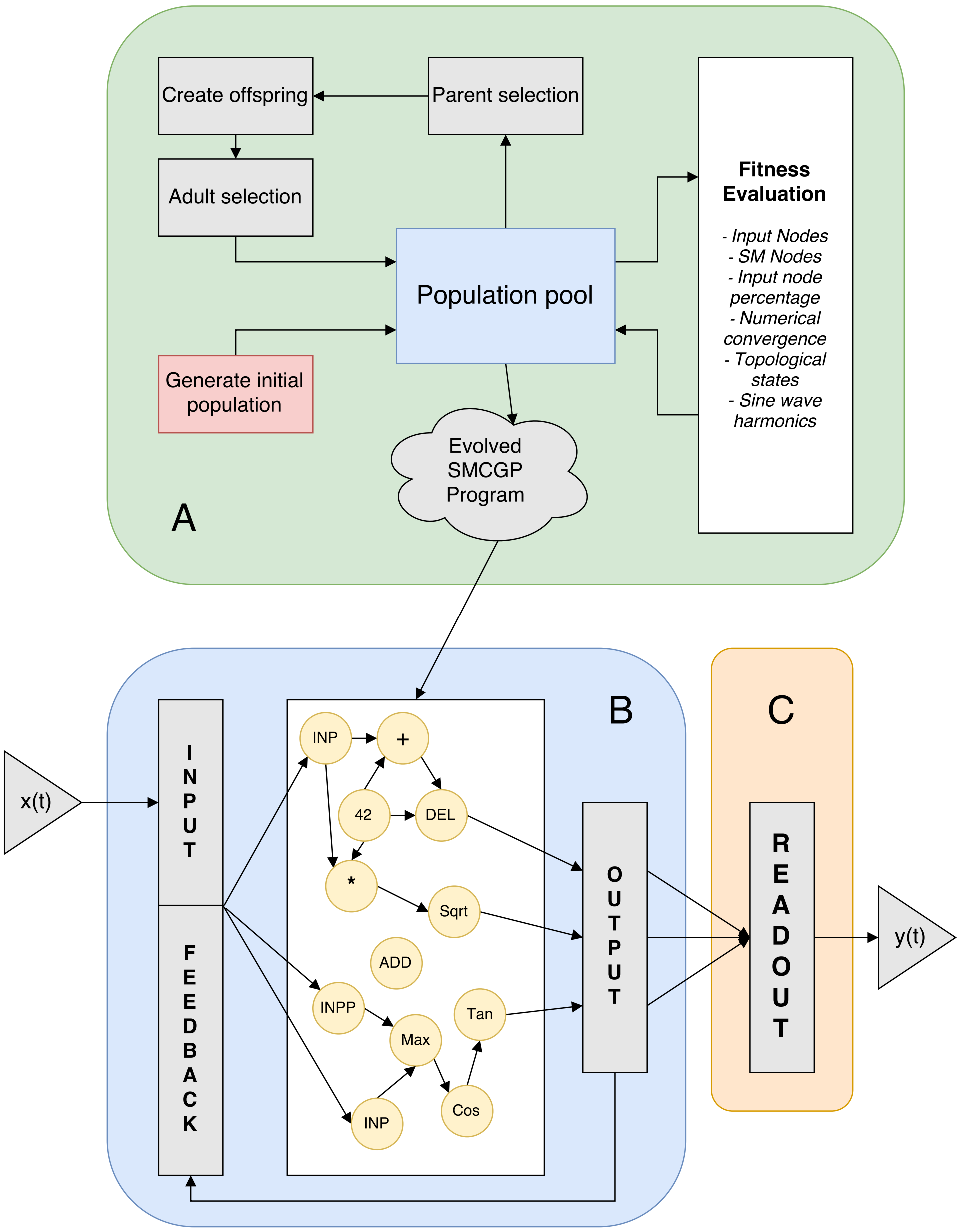


Figure 3.1: Experimental platform

When conducting the experiments, the control flow during normal operation is illustrated in Figure 3.2. There are two main modes of operation, either generate new SMCGP programs, or evaluate existing programs. In the latter case, only the blue (B) and orange (C) components are in use, and the process starts at ”Create SMCGPReservoir”.

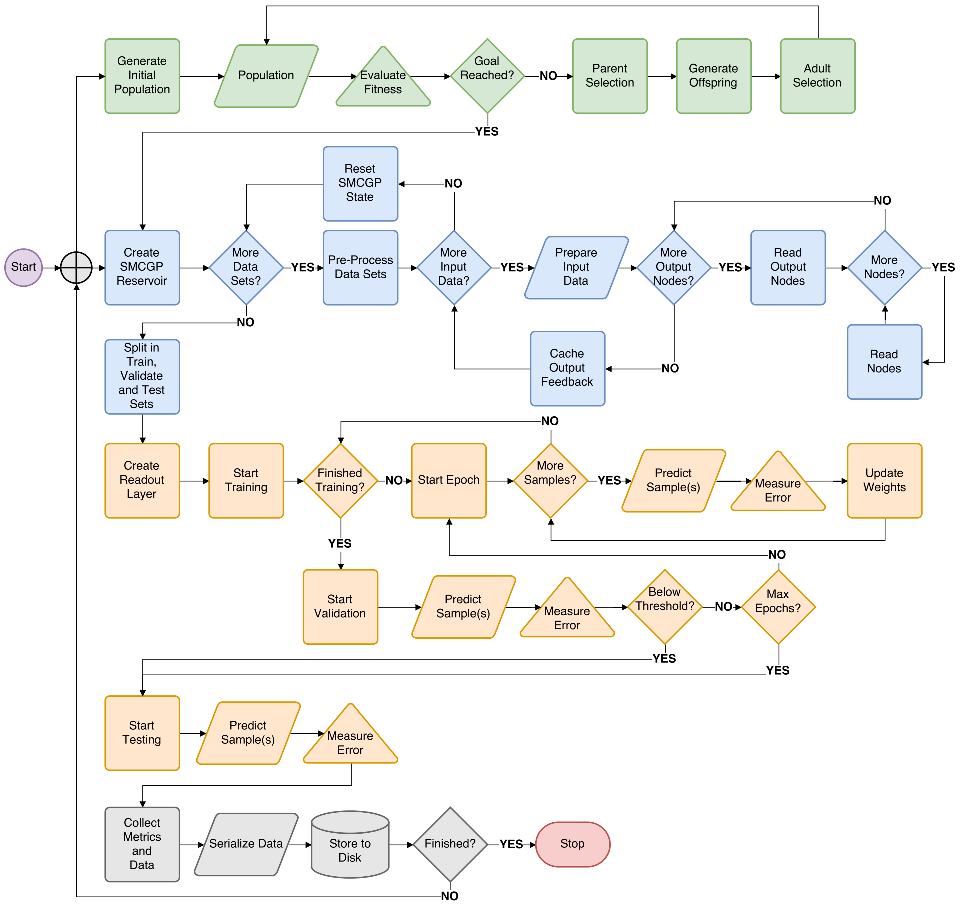


Figure 3.2: Experimental Platform Control Sequence

When generating new programs, an initial population of randomly generated SMCGP programs is created. An evaluation of the fitness of each phenotype is then performed, to check if anyone meets the criteria explained in detail in section 3.2. If there does not yet exist a satisfactory phenotype, a parent selection is performed from the population pool. New offspring are created from these selected parents, before an adult selection process determines which phenotypes will make up the next generation.

When a satisfactory phenotype has been discovered, it is used to construct an SMCGP-Reservoir instance. The data sets for the current task are then pre-processed by iterating the SMCGP program. This involves assembling the input for the current time step, and reading the value of each output node in the program. After all output nodes have been evaluated, the output is cached for use as feedback during the next time step. When all data sets have been processed, the resulting raw outputs from the reservoir and corresponding target values are split into training, validation and testing data sets.

A readout layer is then constructed and training is initiated on the training set. An epoch is started and the readout layer weights are updated after each predicted sample, based on the degree of error. When all samples in the training set have been processed, predictions are made using the validation set. If the measured error is below a set threshold (or if the maximum number of epochs was reached), a final evaluation is performed using a test data set.

Finally, the measured error from the test data set, as well as all runtime statistics (described in subsection 4.2.1) are collected. Together with the program genotype, raw output from the reservoir, state history and other related information, a complete data object for this program is serialized to disk. The entire process is repeated until a given number of programs have been evolved or all existing programs have been evaluated, depending on the mode of operation.

3.2 An incremental approach

The search space for these experiments is unfathomably large. As an example, consider an SMCGP program genotype consisting of 3 nodes. Constraints on combinations, function arity and other specific details are ignored for the sake of simplicity. For starters, each node can be one of the total number of functions available from the implemented function set. A reasonable number may be 30, which includes input and SM functions. Before considering connections, that leaves 27000 possible programs. Each node may connect leftward anywhere from 1 to out of bounds (leading to Zero Input). Assuming an in-degree of 2, and since the order of connections is important for the mathematical functions, this increases the number to 243,000 for the 3 node network. Next, each of the nodes may or may not be an output node, indicated by the last bit in the genome. This increases the number to 1,701,000. Increasing the number of nodes in the program makes it exponentially more difficult. Then consider the fact that the genome parameters P_0, P_1 and P_2 (explained in chapter 4) are real valued, continuous numbers constrained by a configurable lower and upper bound. An exhaustive search quickly becomes impossible. Even with discretization of the real valued parameters, all bets are off when the program is iterated, opening up to self-modification.

In [50], it is noted that the issue of SM activation as it stands is problematic. In most of the problems studied, only a single input has been given at each iteration, causing activation to be an effect of the intrinsic dynamics instead of the data. No matter how many times the input functions are called, the value will always be the same when only a single number is used. However, when introduced to potential bias and feedback from the reservoir system, this may lead to a different phenotype for each data instance.

However, it is also stated in [50] that evolution can find a way around this, leading to some degree of order. To handle this challenge, a strict composite fitness function was used, where the evolved candidate programs had to pass a minimum of several criteria which significantly reduced the search space. Determining the parameter configurations that produce somewhat consistent results was done by performing trial runs on the experimental platform. Programs that succeed in the temporal parity 3 task (section A.1) with **stable** topology were evolved as a first step to confirm the validity of the platform. The configuration parameters from these trial runs were then used as a basis for further investigation once the requirement of continuously altering topology was enforced in the evolutionary search.

3.2.1 Finding Candidate SMCGP Programs

In the evolutionary search (component A in Figure 3.1), an incremental fitness function comprised of several independent assessments of the candidate is used. Table B.2 in Appendix B lists the default configuration parameters that are used in the evolutionary search. The following paragraph describes the most important configurable parameters in the EA search.

Before evolution is started, an initial population is required. This *initial population size* is set to twice the amount of the normal population size to increase the basin of exploration. *Elitism* is a term used to describe the mechanism of retaining the best solution(s) from one epoch to the next. This is enabled in the evolve cycle by transferring the best individual from a generation to the next without any selection or mutation. The *mutation rate* controls the mutation probability of each gene in the genotype. *Population size* dictates the size of each generation. Tournament selection is used as *parent selection scheme* during the experiments. The tournament selections are configured to randomly pick κ (5) individuals from the parent pool, and the best of them is selected as a parent for the next generation. This process is repeated until all the *parent pool size* slots are filled. Offspring are created from the parent pool using mutation and crossover operators. Mutation is performed on a per-node basis, meaning only one node parameter mutation can occur per epoch. After performing some trial runs, a mutation rate of 0.22 was settled on. This rate is very high for traditional SMCGP, where around 0.01 is normal [50]. However, this was a balanced trade-off between search speed and search stability in the setup. Finally, an *adult selection* is performed, where which individuals that are going to represent the next generation is determined. The only adult selection scheme available in the experimental platform is *full generational replacement*. In essence, the entire population is replaced with the offspring created from the parent pool.

To determine what properties of a candidate are interesting or acceptable, one must take the motive described in chapter 1 into consideration. Programs that solve the task on their own are not the target programs, as that would render the reservoir obsolete. Neither are programs that are able to solve the task, but are topologically stable. Although interesting, the latter does not fit well as a model of plasticity. Neither does it provide much insight into the computational capability of dynamic, self-organising structures. The following SMCGP program properties are therefore evaluated in the composite fitness function to limit the search space.

Active input

After each perturbation of a phenotype (program iteration) there has to exist at least one input function in the set of active nodes. Let P_t be the entire phenotype, and A_t be a set of active nodes $A_t \subseteq P_t$ at time step t. Then $\exists i_t \in A(Input(i_t))$. This check immediately terminates graphs that no longer are input responsive.

Active self-modification

As previously stated, the main property of interest is that a program continuously self-modifies. Let P_t be the entire phenotype, and A_t be a set of active nodes $A_t \subseteq P_t$. For each time step t, then $\exists i_t \in A_t(SM(i_t))$. If this condition is violated, in other words, that the program removes its own ability to self-modify, evaluation is immediately aborted.

Numerical convergence

A graph might have input nodes in the set of active nodes, but they might be intercepted by constant or boolean nodes which nullify the values. Output feedback might also cause the signal to converge towards a fixed point. Let n be the number of output nodes, τ be a configurable number of time steps back in time to check, t be the current time step, $O_{i,j}$ be the output for output node i at time step j and ϵ be a convergence threshold.

$$Different(i,j) = \begin{cases} 1, & \text{if } |O_{i,j} - O_{i,j-1}| > \epsilon \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

$$Converged(i) = \begin{cases} 1, & \text{if } \sum_{j=0}^{\tau-1} Different(i, t-j) = 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.2}$$

Then the following inequality must be upheld by the program to prevent a fitness penalty.

$$\frac{n}{3} \geq \sum_{i=0}^{n} Converged(i) \tag{3.3}$$

For a program to pass this test, it can have no more than $\frac{1}{3}$ of numerically converged output columns. This check is performed after a complete input data set has been passed through.

Minimum input node percentage

Another property parameter that the fitness function checks is the *minimum input node percentage*. Let μ be the minimum input node factor, PI_t be the set of input nodes in a phenotype at time step t, T be the length of each input stream, and P_t be the set of all nodes at t. Then for each input stream, Equation 3.4 must hold true.

$$0 = \sum_{i=0}^{T} f(t), \text{ where } f(t) = \begin{cases} 1, & \text{if } |PI_t| < \mu|P_t| \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

This check ensures that the number of input nodes never drops below the specified threshold.

Topological states

As opposed to Random Boolean Networks, where each node can either be in a state of 0 or 1, the outputs of SMCGP nodes can be continuous real valued numbers. Another difference is that an SMCGP node does not have a definite value. It may be read several times during the calculation of a single output node, each time with a different set of inputs. This can be viewed as the node changing its state several times in the course of one state inspection. To properly know the state of each node one would have to record every calculation, which depending on the size of the program, could mean reading upward of 100 node values multiple times for each iteration. That many read calls would be very computationally expensive and not practical. Thus the internal values of the nodes were considered to be a ”black box”. Searching for attractors in the values of nodes is therefore limited to the output nodes. One could plot the values from each output node and look for interesting attractors. Feedback from the previous time step is fed back into the reservoir and can potentially lead to interesting patterns. However, the hope is to achieve computation in programs that alter their structure. For this purpose the numeric output of the program is ignored in this context, and instead, the topological state transitions are assessed.

Let A_t be the set of active nodes in a phenotype at time step t, H be a hash function, and $S(t)$ be the state of the active nodes at time step t. The topological state of the program is then given by:

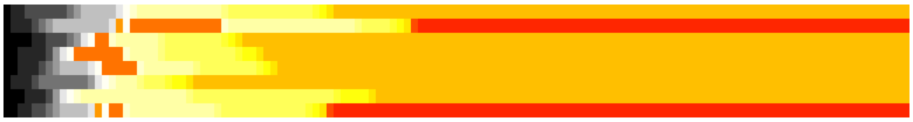
$$S(t) = H(A_t) \tag{3.5}$$

The total number of unique states seen after the reservoir has been perturbed by all input streams $i \in I$ is then $|S|$.

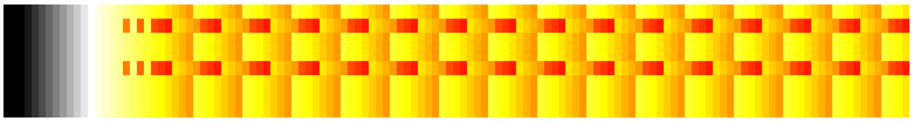
A state is a snapshot of the active nodes at a given time step. Any change to node index (ordering), function type, connections, parameters $P_0 ... P_2$ or output bit will result in a new hash code. The state at each time step is recorded. As mentioned in chapter 2 a system is thought to be at its most computationally capable at the *”edge of chaos”*. Constantly altering topology might give rise to highly non-linear, potentially chaotic output, and is therefore an area of interest in the experiments.

Since a multitude of different states are expected, a black and white or grey-scale visual aid might not suffice. Instead a visualisation using a bounded colour scale was created, where each new discovered state is assigned the next colour increment. This illustrates cycles in the state transition diagram well, as it will often appear as a natural gradient. Transition diagrams with many different trajectories will appear more chaotic. In these visualisations, each row on the y axis represents its own data set. Time t flows from left to right on the x axis. Within the same image, identical colours mean an identical state, even across the different data sets shown in each row. The starting state of the phenotype is equal for all input streams. Here the notion of initial conditions or initial state with respect to basins of attraction refers to the pattern of inputs the system is perturbed with.

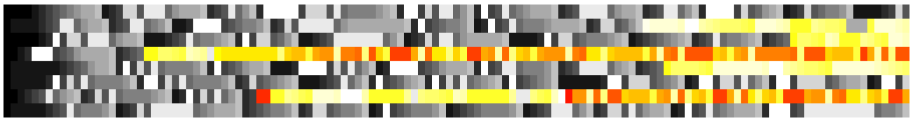
Figure 3.3a shows the system stabilising on one of two point attractors. The input patterns in row 2 and 8 belong to separate basins of attraction compared to the remaining rows. In Figure 3.3b, the system enters one of two cyclic attractors. Again displaying that the input patterns belong to two separate basins of attraction. Finally, Figure 3.3c shows a system that due to the input pertubation fails to fall into any distinct pattern within the observed time steps.



(a) Point attractor



(b) Limit cycle



(c) Chaotic

Figure 3.3: Topological state attractors

All three examples in Figure 3.3' were perturbed with 8 separate streams of 128 randomly generated bits, as well as output feedback from the previous time step.

After each input stream is finished, the state history is analysed. If the state history contains two or less unique states, the phenotype is immediately terminated with a low fitness score. Due to prior fitness checks, there must be at least one SM function in the set of active nodes at each iteration. This SM function might not be activated, or be activated but alter nodes in a disconnected part of the graph. Even though an SM operation has been performed at each iteration, the result is effectively a $no - op$.

Furthermore, after all input streams are finished, the set of observed states is assessed, to ensure that $|S| \geq \sigma$, where σ is a configurable minimum number of unique states. Let I be a set of input streams, T be the length of each stream, $S_{i,t}$ be the state of the phenotype on input stream i at time step t, and τ be a configurable number of time steps to check. Then Equation 3.7 must hold true.

$$Different(i,j) = \begin{cases} 1, & \text{if } S_{i,j} \neq S_{i,j-1} \\ 0, & \text{otherwise} \end{cases} \tag{3.6}$$

$$1 \leq \sum_{i=0}^{|I|} \sum_{t=0}^{\tau-1} Different(i, T-t) \tag{3.7}$$

This ensures that there have been at least two distinct states during the last τ time steps, across all input streams.

Sine wave transformation

A sampled sine wave signal is used to iterate the program, while recording the results from each output node. Each output node represents its own sink for the signal and is analysed independently. When transforming the signal into a higher dimensional representation, the reservoir may add more information from its intrinsic dynamics. The resulting output signal may be the combination of several sinusoidal wave functions. This can be used as an indicator that the system might be able to model a multitude of functions, giving rise to some computational capability. By utilising *Fast Fourier Transform*, one can inspect the power spectrum from each output, to see what frequency components the resulting signals consist of.

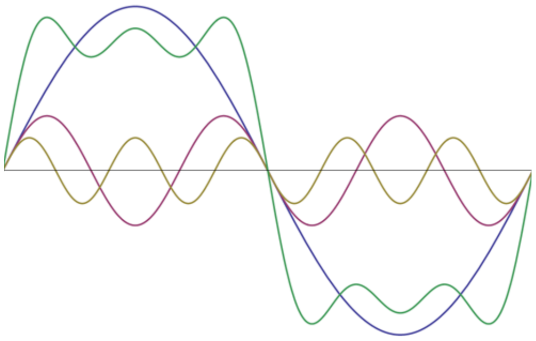


Figure 3.4: Harmonics of sine save

The *fundamental angular frequency* w_0 of a periodic signal is defined as $\frac{2\pi}{T}$ for a period T. A periodic function is said to be *harmonic* whenever it relates to another periodic function by an integer n (Figure 3.4). Periodic functions with periods of T are also periodic at nT for all positive n. Likewise, for functions with $\frac{T}{n}$ periods, they are also periodic at intervals of T. This property is used when looking at the spectrum, as spikes in harmonic frequencies indicate transformations of the original signal.

According to the Nyquist (Nyquist-Shannon) sampling theorem, to sufficiently represent a continuous signal $f(t)$ in a computer, it is necessary to sample it at a high enough rate [58]. It states that to represent a signal, the sampling frequency F_s must be greater than or equal to twice the maximum frequency F_m of the signal. Increasing the sampling frequency increases the accuracy of the signal.

This fitness check is performed by iterating the phenotype with 128 samples of a periodic signal $f(t)$ described in Equation 3.8 with frequency $F = 1Hz$, amplitude $a = 1.0$ and phase shift $p = 0.0$ recorded over $2T$ (2s, sampling rate $F_s = 64Hz$).

$$f(t) = a * sin(2 * \pi * F * t + p) \tag{3.8}$$

Let O be the set of output nodes, n be a configurable minimum number of harmonics, γ be a minimum power coefficient, $f(t)$ be a periodic signal at time t, and $FD_{o,j}$ be frequency bin i of the frequency domain of signal $f(t)$ for output node o.

$$h(o) = \sum_{j=0}^{|FD(o)|} \begin{cases} 1, & \text{if } FD_{o,j} \geq \gamma \tilde{FD_o} \\ 0, & \text{otherwise} \end{cases} \tag{3.9}$$

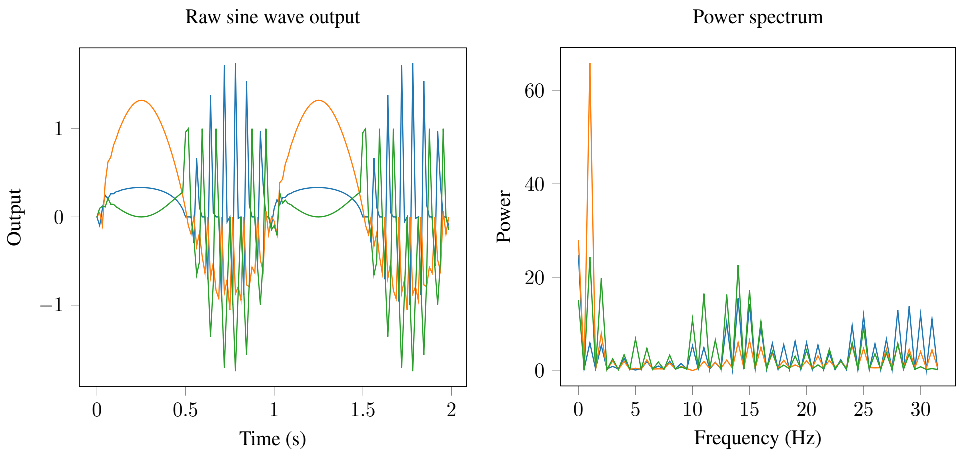
$|FD_o|$ is the number of frequency bins in the power spectrum of output node o, and $\tilde{FD_o}$ is the median of the power spectrum of output node o.

$$n \leq \sum_{i=0}^{|O|} h(i) \tag{3.10}$$

In this check, Equation 3.10 must hold true, or else the phenotype is given a fitness penalty. If the sum of the number of harmonics for all output nodes is less than the configured number n, the phenotype is penalised. Signal power is considered strong enough if it surpasses a factor of γ of the median signal power.

Figure 3.5a is the raw output from an SMCGP program with 3 output nodes. Here, we see that the sine wave signal has been transformed. Except for some fluctuations when the output is negative, the orange output node maintains much of the original signal. The two remaining output nodes have been more transformed, with the blue output node barely having negative output.

Figure 3.5b shows the corresponding power spectrum. Here we can see that most of the signal power from the orange output node is at the 1Hz bin, as expected given how much of the original signal remains. For the other two nodes, we see that the majority of the signal now consists of a multitude of different frequency components.



(a) Raw sine wave output **(b)** Power Spectrum

Figure 3.5: Spectrum analysis of SMCGP program with 3 output nodes

Figure 3.6 shows the corresponding state history of the phenotype in question, when iterated with a random bit stream.

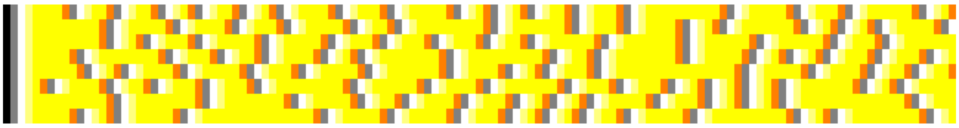


Figure 3.6: State history of SMCGP program with 3 outputs

The complete fitness procedure is displayed in Algorithm 1. Functions *hasActiveInputAndSM* and *isNumericallyConverged* return early with the current fitness if they fail to yield maximum value.

Algorithm 1 Fitness Evaluation

```
procedure EVALUATEFITNESS(phenotype)
    $F \leftarrow 0$
    $F$ += $hasActiveInputAndSM(phenotype)$
    $F$ += $isNumericalConverged(phenotype)$
    $F$ += $inputPercentage(phenotype)$
    $F$ += $topologicalStates(phenotype)$
    $F$ += $sineWaveHarmonics(phenotype)$
    return $F$
end procedure
```

3.2.2 Investigating the Viability of the Candidate

After a phenotype was evolved, it was used as a reservoir (component B in Figure 3.1) and pre-processed with data, before training a readout layer (component C in Figure 3.1) on the temporal parity task with window size $n = 3$. A multi-layer readout was used, due to the nonlinear nature of the SMCGP program outputs. The readout was configured to use a hidden layer with sigmoid activation function, and a Softmax (Equation 3.11) classification layer as output. The Softmax function squashes the K dimensions of the incoming vector z, producing a probability distribution summing to 1.0.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{3.11}$$

Training the readout is done using *Stochastic Gradient Descent (SGD)*. The updater used is a variant of the standard momentum based updater, called Nesterov's accelerated gradient (NAG) [59], and the loss function is *Multi-Class Cross Entropy*. For regression problems the loss function is changed to *Mean Squared Error (MSE)* and output activation function is the identity function.

Model accuracy is defined as the number of correctly classified samples c divided by the total number of samples $|D|$ in the data set, $\frac{c}{|D|}$.

One assumption made about how input is read by the SMCGP program, was that when the number of output nodes grows large, the relative presence of the input value diminishes. This is due to output from the previous time step being fed back into the reservoir. For a given program with 15 outputs and 1 input value, then the input data from the training set would account for $\frac{1}{16}$ of the input array read by the program. To investigate this relationship, a feature called input expansion was added to the platform. This is a configurable setting that may vary in the experiments. An input expansion factor of 3 will mean that for a program with 9 output nodes, the input bit will represent $\frac{3}{12} = \frac{1}{4}$ of the input array available to the program.

3.2.3 Exploring the Computational Capability of the Candidate

Programs that had shown good model accuracy, specifically above 90% on the temporal parity 3 task, were selected for additional tasks. Each program was then ran on the temporal parity 5 and 7 tasks, as well as temporal density 3, 5 and 7 (Appendix A.1).

Additionally, each of the candidates evolved in subsection 3.2.1 were tested using the NARMA10 benchmark (Appendix A.2).

3.2.4 Exploring the Characteristics of Well Performing Programs

Finally, characteristics of the best-performing programs were investigated. From the genotype and iterated phenotypes, patterns and frequencies of function types were assessed. The executed SM-operations were studied to see if there were any correlation between the well-performing programs and self-modification arrangement and type. The topological states were analysed, in an effort to look for tendencies in the state trajectories. This was done by generating state transition diagrams and inspecting the state history using the aforementioned visualisations. Finally, the sine harmonics from the sine wave transformation were analysed to see whether some harmonics could be correlated with the well-performing programs.

Chapter 4

Implementation

This chapter describes the implementation details of the experimental platform. It describes in detail how SMCGP was implemented and how it is integrated with the RC part of the platform.

4.1 Self-Modifying Genetic Cartesian Programming

The SMCGP implemented in the system is based on the approach described in *Developments in cartesian genetic programming: Self-Modifying CGP* [18] with additional insights from the 2011 version of J. Miller's book *Cartesian Genetic Programming* [41] and considerations from [52, 53, 54, 55]. The implementation was responsible for generating and evolving SMCGP programs which could be tested as a reservoir in a RC system.

4.1.1 Genotype

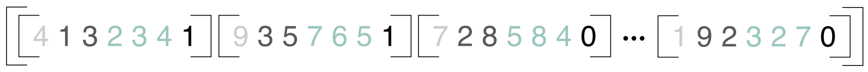
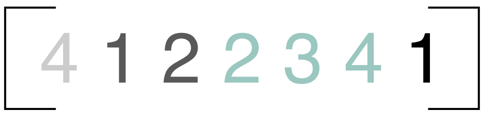


Figure 4.1: Genotype

Figure 4.1 shows how the genotypes in the implementation are represented. The genotypes are stored as two-dimensional arrays which are responsible for encoding the initial phenotypes. The size of the outer array determines the size of the initial phenotype program. Initial size is controlled by a configuration parameter, *startNodes* (see Appendix B for all available configuration options). Figure 4.2 is an example of an inner array. Each of the inner arrays is a list of numbers encoding one node in the graph. In the system, genotypes are created in three ways, either randomly generated to create the initial population for the EA search or it is a crossover/recombination of two parent genotypes, or a copy of a single parent possibly carrying mutations.



The first number dictates what function type the node represents

The two next numbers tells the node which other nodes it should connect to

The three next numbers are randomly generated real numbers that are used in the various SM operations, denoted as P0, P1 and P2

The last number indicates if this node is an output node

Figure 4.2: A single node represented in genotype

4.1.2 Phenotype

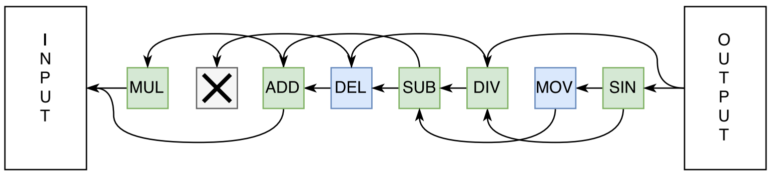


Figure 4.3: SMCGP Program

The phenotype is an SMCGP program and is created by decoding a genotype. Figure 4.3 is a graphical representation of a phenotype. Each node in Figure 4.3 is developed from the inner arrays of the genotype (shown in Figure 4.2).

4.1.3 Development

Internally the phenotype represents the graph just as the genotype does, using two-dimensional arrays. When a phenotype is developed from a genotype, it copies the arrays (so the original genotype is not altered when the phenotype self-modifies) and identifies the output nodes. The output nodes are located by iterating over the array of nodes from end to start until enough nodes with the output flag (the last index in Figure 4.2) enabled are found. Each time the program self-modifies this process has to be repeated in case that the output nodes are deleted or have changed position.

4.1.4 Relative Addressing and Reading Output

The input-output mapping is done through recursion from output to input. The *readNode* method propagates through the graph until an input node is reached, the node connections point outside the graph (returning zero) or a constant node is encountered. Reading the value of a given node N_i at absolute index i, with function F_i and incoming connections A_i and B_i is defined in Equation 4.1.

$$f(i) = F_i(f(i - A_i), f(i - B_i)) \tag{4.1}$$

This is the general case, but some functions in F will only use A_i or only B_i, and the constant and input functions will use neither. The locations of the incoming connections are offsets of the absolute position of the current node, i.

4.1.5 Inputs

The only way for an SMCGP program to read input is through nodes with input functions. Input values are organised as a queue, with a pointer to the next value to be fetched from the queue. Each of the three input functions returns the value which the pointer is currently pointed at, but they differ in how they move the pointer for the next time an input function is called. Table 4.1 shows how the pointer is altered for each function.

Name	**Description**
INP	*Return the value at the current pointer position and increment the pointer by 1*
INPP	*Return the value at the current pointer position and decrement the pointer by 1*
SKIP	*Return the value at the current pointer position and increment the pointer by* P_0

Table 4.1: Input Functions

4.1.6 Mathematical Functions

The nodes with mathematical functions are ultimately responsible for transforming the input. When a node with a mathematical function is encountered, the operation is loaded from a hash table containing all available mathematical functions (exhaustive list Table 4.2). The two input values (I_1 and I_2) are collected by reading the neighbouring nodes, before returning the result of the mathematical operation. Due to their nature, functions like *(*Sine, Cosine and Square root), only use the first input connection, where *Constant* use neither.

Name	**Return Value**
Add	$I_1 + I_2$
Subtract	$I_1 - I_2$
Divide	$\frac{I_1}{I_2}$
Multiply	$I_1 * I_2$
Sine	*Sine of I_1 given in radians*
Cosine	*Cosine of I_1 given in radians*
Tan	*Tangent of I_1 given in radians*
Tanh	*Hyperbolic tangent of I_1*
Constant	P_1
Max	*Highest value of I_1 and I_2*
Min	*Lowest value of I_1 and I_2*
Mean	$\frac{I_1+I_2}{2}$
Modulo	$I_1 \mod I_2$
Pow	$I_1^{I_2}$
SquareRoot	$\sqrt{I_1}$
And	$I_1 > 0 \wedge I_2 > 0$
Or	$I_1 > 0 \vee I_2 > 0$
NotAnd	$\cancel{(}I_1 > 0 \wedge I_2 > 0)$
LargerThan	$I_1 > I_2$
SmallerThan	$I_1 < I_2$

Table 4.2: Mathematical Functions, where I_1 and I_2 are the values read from the connected nodes.

4.1.7 Self-modifying Operations

Like mathematical nodes, SM-nodes load their corresponding SM-function from a hash table containing all available SM-operations. An SM-operation is added to the list of pending modifications, the (*Todo-list*), if it is activated. Depicted in Figure 4.4, the SM-node is activated if incoming connection $I_2 > I_1$.

When the value of all output nodes have been determined, the SM-operations in the Todo-list are executed. The number of allowed operations per iteration is limited by a configurable setting, *todo-size*. As in [18] the SM-operations are executed left-to-right, meaning that the Todo-list is sorted based on node position before execution.

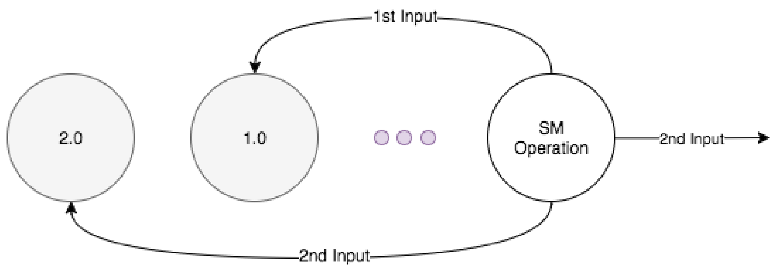


Figure 4.4: SM-node activation

When it comes to the ToDo-list execution, the implementation differs slightly from what is described in [18] and [41]. There are guards set in place to avoid that programs either grow out of control, or shrink so much that it has fewer nodes than the required output nodes. The growth is controlled by the configurable option, *maxGrowthFactor* and *maxNodesMovedOrDuplicated*. These limit the amount of nodes that can be operated on at once, by the move, duplicate and add functions. This was done to allow the platform to be configured for high speed evaluation.

Name	**Description**
Add	*Add P_1 new random nodes at P_0*
Delete	*Delete nodes from P_0 to P_1*
Change Function	*Update function type of node at position P_0 to P_1*
Change Parameters	*Update SM-parameter $P_1 mod 3$ of node at position P_0 to P_2*
Change Connections	*Update connection $P_1 mod 2$ of node at position P_0 to P_2*
Shift Connections	*Increment connections of node at position P_0 by P_2*
Move	*Move nodes between x + P_0 and x + P_0 + P_1 and insert after x + P_0 + P_2*
Duplicate	*Copy nodes between x + P_0 and x + P_0 + P_1 and insert after x + P_0 + P_2*

Table 4.3: Self-modifying Operations

Table 4.3 lists all available SM-operations. All operations require parameters from the activated node in their alterations. The parameters are randomly generated in the genotype within the boundaries of the system settings, *realNumberLower* and *realNumberUpper*. When these parameters are used for node indexing, they are first transformed by Equation 4.2 to ensure that the parameters are not out of bounds in programs where the number of nodes has shrunk. Similarly, wrapping is done for other ranges such as function lists, or node parameter index.

$$P' = P_x \bmod |N| \tag{4.2}$$

Were P_x is the used parameter (P_0, P_1 or P_2) and $|N|$ is the total size of the SMCGP program, both active and inactive nodes.

4.1.8 Evolution

As stated in chapter 3 the evolutionary process chosen for the implementation was *tournament selection* with separate parent and population pools. The selection scheme used is the same as in [18, 41], where the phenotype are ranked by fitness first, then generation. Meaning that if two phenotypes have the same fitness, the youngest is preferred.

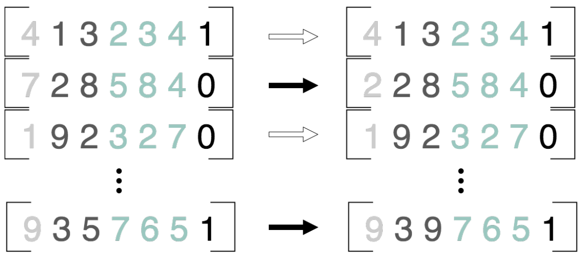


Figure 4.5: Genotype mutation per genome

The key components of the EA cycle is shown in Algorithm 2. Crossover is implemented as single point crossover, whereas mutation is done on a per-node basis. As depicted in Figure 4.5, this means that a node might or might not mutate at all, but if a mutation occurs, another random selection is done to determine what type of mutation is performed. The types of mutation relate to the different node properties. These mutation types are function mutation, connection mutations, internal parameter mutations and flipping the output flag.

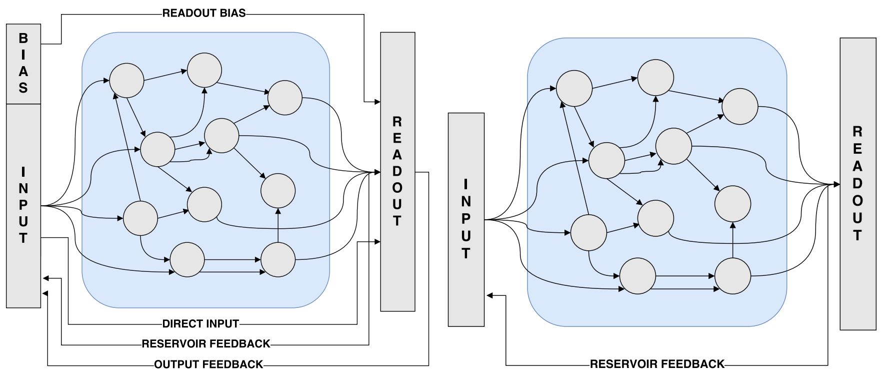
Algorithm 2 Evolution Cycle

```
procedure EVOLVE
    $P \leftarrow$ initializePopulation()
    fitnessEvaluation($P$)
    $C \leftarrow$ getBestIndividual()
    while $C.fitness \neq target$ do
        $S \leftarrow$ parentSelection($P$)
        $O \leftarrow$ createOffspring($S$)
        $A \leftarrow$ adultSelection($O$)
        $P \leftarrow A$
        fitnessEvaluation($P$)
        $C \leftarrow$ getBestIndividual()
    end while
    return $C$
end procedure
```

4.2 Reservoir Computing System

The reservoir system was implemented based on the description of generic RC systems in [43]. Even though both SMCGP and the reservoir system were developed to be used in unison, the two components were not tightly integrated, allowing the SMCGP program to be iterated as a standalone program without feedback.

When constructing a reservoir, an *SMCGPReservoir* instance is made using the program in question. The two are then automatically connected and any input passed to the reservoir will iterate the program.



(a) Generic Reservoir System **(b)** Implemented Reservoir System

Figure 4.6: Reservoir computing system overview

The reservoir implementation in the experimental platform (Figure 4.6b) only uses a subset of the features provided by the generic reservoir scheme (Figure 4.6a).

Training is performed in an offline fashion. This allows the platform to accumulate raw output data from the reservoir and construct data sets that can be batched and shuffled.

As briefly mentioned in section 3.1, SMCGP does not possess feedback or intrinsic memory in the general sense. To allow the SMCGP program to more easily solve time dependent tasks, the system caches reservoir output for an arbitrary amount of time steps. These are made available as input for the reservoir at the next time step. The number of cached time steps are controlled by a configurable parameter $numCachedTimeSteps$, and is set to 1 as default. Without this feedback, the only intrinsic form of memory in SMCGP would be either purely by topological changes affecting program behaviour, or by means of *Constant* nodes in combination with *SM-ChangeParameter* nodes. The latter would work by changing the value of the constant nodes, changing the numerical values in the computation, rather than the functions involved. As ChangeParameter is an SM-node and therefore input driven, these numerical changes would also be input driven. ChangeParameter alters parameter P_1 *mod* 3 for node P_0 to the value of its own P_2. This may result in a change of parameter P_1 of a Constant node, changing its return value. Disregarding the effects of *SM-Add*, this form of memory would be limited by the set of distinct P_2 values in the original genotype. Without SM-Add, no new P_2 parameters can be introduced to the program.

Competing ChangeParameter functions may alternate between changing the value of one or more constant nodes. The changing return value of these Constant nodes could represent some limited memory. Given a set of unique P_2 values P with size $s = |P| = 3$ and a number of Constant nodes $c = 5$, a total of $3^5 = 243$ different states could be encoded this way. Nonetheless, this is an ideal scenario, where the different ChangeParameter functions are frequently not being activated, changing the contents of the Todo-list. In turn, this allows the remaining ChangeParameter functions to use their P_2 values. Neither does it take into account that this effect could very well change the other two parameters of competing functions, removing any competition. In the long term it is therefore reasonable to assume that one single P_2 value would in the end dominate, as there are statistically more ways to destroy the effect than to maintain it. This very specialised form of memory would require a very select set of circumstances to work properly. Some form of memory encoded purely in topological changes seems more likely. Regardless, both these examples would require a tremendous number of evolutionary steps to tune into handling complex temporal tasks. Hence, from a statistical perspective, providing memory through reservoir feedback was the most promising approach.

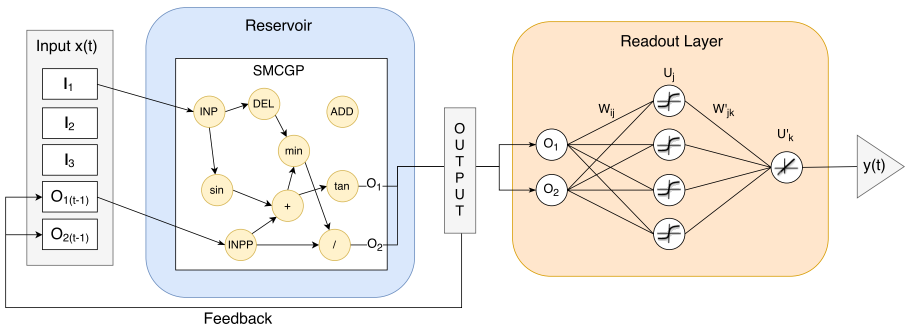


Figure 4.7: SMCGP Reservoir Integration

Figure 4.7 shows how an SMCGP program is integrated in the reservoir system.

An additional feature not found in generic RC systems is implemented in the experimental platform. This *input expansion* feature allows single input values to saturate a larger portion of the input data array before being passed into the reservoir. SMCGP input functions may increment, decrement or skip several inputs depending on its parameters. Very erratic read patterns may occur. The *input expansion factor* configuration parameter tells the RC system how many input array slots the input data should be copied into.

Figure 4.8 shows the input array passed to the reservoir with input expansion factor $f = 1$, when the current input $i_t = 1.0$ and the outputs from time step $t - 1$ is 0.12, 11.3, -1.0 and 3.14.

Figure 4.9 illustrates how the layout of the input array looks when using an input expansion factor $f = 3$.

This allows the effects of relative input versus feedback size to be investigated by experiments. The degree of feedback may affect performance on short-term vs long-term time

$$[1.0, 0.12, 11.3, -1.0, 3.14]$$

Figure 4.8: Standard input concatenation

$$[1.0, 1.0, 1.0, 0.12, 11.3, -1.0, 3.14]$$

Figure 4.9: Input concatenation with input expansion factor of 3

dependencies. When increasing the number of outputs, feedback has the effect of increasing the input array size by a factor of the number of cached time steps c. Without using input expansion, this would quickly make the feedback signal dominate the input data. The program may also end up not being able to read all the input before all output nodes have been evaluated, unless the phenotype is sufficiently large and maximum connection length for the nodes is relatively short.

Data sets are created in batches (several independent input streams), each of a specific length depending on problem type. The SMCGP program is reset to its original state before processing each of these input streams. This is done to expose the program to a range of different starting input patterns, hopefully identifying different basins of attraction.

4.2.1 Metrics Collection

A vital part of the implementation is facilitating the collection of statistics as the SMCGP programs are evolved and the reservoir is perturbed by input data. Using a collection of listener interfaces receiving a wide range of events, the platform is able to log several effects of each program iteration. The platform will gather the following metrics during operation.

– Frequencies of fitness penalties for each terminated phenotype
– Total number of evaluations and epochs for each successfully evolved phenotype
– Population max/mean/min/std fitness for the evolutionary search
– For each iteration the phenotype size, size of active nodes set, topological state and potential SM operations will be recorded. This is used to calculate minimum/mean/-max size of the active nodes during the complete iteration of a program, as well as generate state transition plots and diagrams.
– Timings for each input processing session are also recorded to assess the computational footprint of the program.
– Training set and test set readout layer accuracy at a configurable interval.
– Readout layer training speed in epochs per second

Chapter 5

Experiments

In this chapter, the experiment results from the phases described in chapter 3 are presented. Formal definitions for the benchmarks used are listed in Appendix A. For the temporal parity and temporal density experiments, readout layer accuracy is defined as the number of correctly labelled classes c divided by the total number of samples n in the data set $\frac{c}{n}$. For the temporal parity and temporal density tasks, training was stopped if accuracy reached ≥ 0.98.

A complete list of available configuration parameters is listed in Appendix B, section B.1. The configuration parameters defined in Table B.2 and Table B.3 were common for all experiments.

Note: *the notation X-Y, is used throughout this chapter and is a shorthand for a configuration, where X is the number of output nodes in a program and Y is the input expansion factor (IEF).*

5.1 Candidate Search

This experiment evolves SMCGP programs using an evolutionary search, and tests the performance of the resulting programs on the temporal parity task with window size $n = 3$.

Table 5.1 lists the configuration parameters used for the SMCGP programs and fitness function.

Table 5.2 lists the configuration parameters used for the readout layer.

Setting	**5-1**	**7-1**	**7-2**	**10-1**	**10-3**	**13-1**	**16-3**	**16-5**
startNodes	50	100	100	200	200	200	300	300
levelsBack	25	50	50	100	100	100	150	150
feedBack	$t-1$	$t-1$	$t-1$	$t-1$	$t-1$	$t-1$	$t-1$	$t-1$
minStates (σ)	8	8	8	8	8	8	8	8
statesSteps (τ)	16	16	16	16	16	16	16	16
minHarmonics (n)	4	6	6	8	8	12	16	16
spectrum (γ)	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3
convergence (τ)	16	16	16	16	16	16	16	16
convergence (ϵ)	10^{-10}	10^{-10}	10^{-10}	10^{-10}	10^{-10}	10^{-10}	10^{-10}	10^{-10}
inputNodeFactor (μ)	0.12	0.12	0.12	0.1	0.1	0.1	0.1	0.1

Table 5.1: Configuration parameters used in the evolutionary search

Setting	**Value**
Max epochs	2000
Learning rate	0.03
Hidden layer activation	Sigmoid
Output layer activation	Softmax
Hidden layer nodes	16
Loss function	Multi-Class Cross Entropy
Updater	Nesterov's Accelerated Gradient (NAG)
Momentum μ	0.9

Table 5.2: Temporal parity and temporal density readout parameters

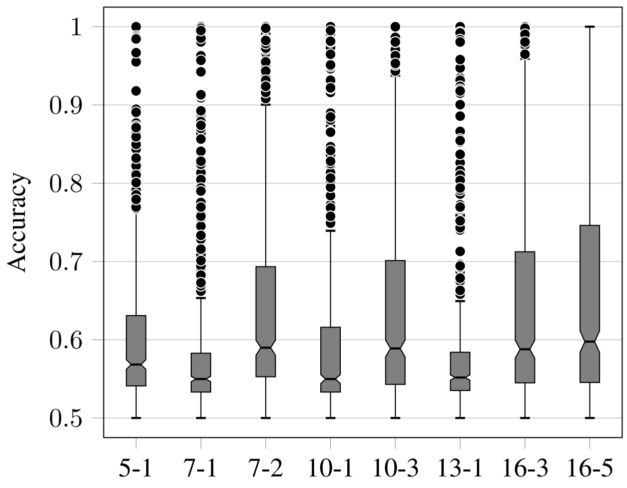


Figure 5.1: Readout layer accuracy vs program configuration on temporal parity 3

A total of 4240 SMCGP programs were evolved, 530 of each group of output nodes and input expansion factor.

Figure 5.1 shows the readout layer accuracy for SMCGP programs with different number of output nodes and IEF on the temporal parity 3 task.

The results were gathered by iterating each program with 24 bit streams of 128 random bits each, then splitting into a training set of 2048 samples and validation and test sets of 512 samples respectively. The readout layer was trained for 2000 epochs or until an accuracy ≥ 0.98 was reached.

Outputs	**IEF**	**Amount**	**Accuracy** $\geq 90\%$
5	1	530	12 (2,3%)
7	1	530	27 (5,1%)
7	2	530	36 (6,8%)
10	1	530	30 (5,7%)
10	3	530	36 (6,8%)
13	1	530	30 (5,7%)
16	3	530	30 (5,7%)
16	5	530	55 (10,4%)

Table 5.3: SMCGP programs with over 90% readout layer accuracy on temporal parity 3

Table 5.3 lists the number of programs for each configuration that achieved an readout layer accuracy above 90%. Out of all 4240 evolved programs, 256 managed to get over 90% accuracy.

5.2 Evaluating Performance with Benchmarks

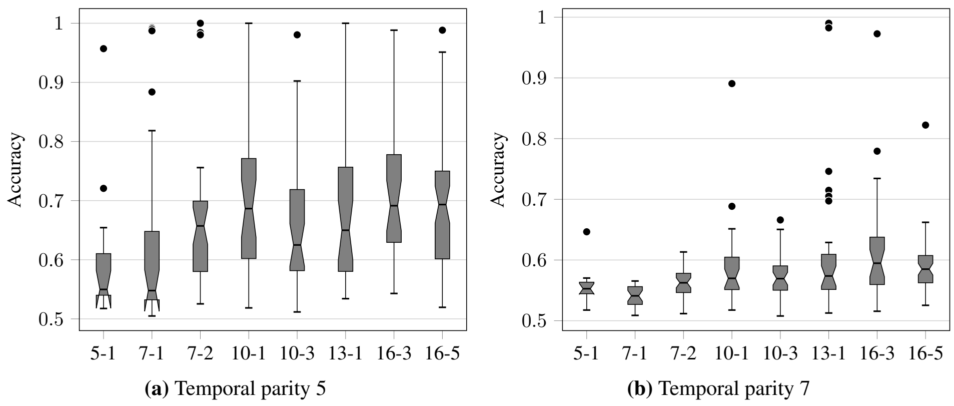
5.2.1 Temporal Parity

This experiment tests the performance of the 256 programs that achieved above 90% at on the temporal parity 3 task. Each program is evaluated using temporal parity 5 and temporal parity 7 benchmarks. As scoring high on temporal parity 3 was a requirement for candidacy in these benchmarks, that test was not performed a second time.

Table 5.4 lists the readout layer configuration parameters used for this experiment.

Setting	**Value**
Max epochs	4000
Learning rate	0.03
Hidden layer activation	Sigmoid
Output layer activation	Softmax
Hidden layer nodes	16
Loss function	Multi-Class Cross Entropy
Updater	Nesterov's Accelerated Gradient (NAG)
Momentum μ	0.9

Table 5.4: Temporal parity readout parameters



(a) Temporal parity 5 **(b)** Temporal parity 7

Figure 5.2: Readout layer accuracy vs program configuration on temporal parity 5 and temporal parity 7

Figure 5.2 shows the readout layer accuracy for SMCGP programs with different number of output nodes and IEF on the temporal parity 5 and temporal parity 7 tasks.

The results were gathered by iterating each program with 24 bit streams of 128 random bits each, then splitting into a training set of 2048 samples and validation and test sets of 512 samples respectively. The readout layer was trained for 4000 epochs or until an accuracy ≥ 0.98 was reached.

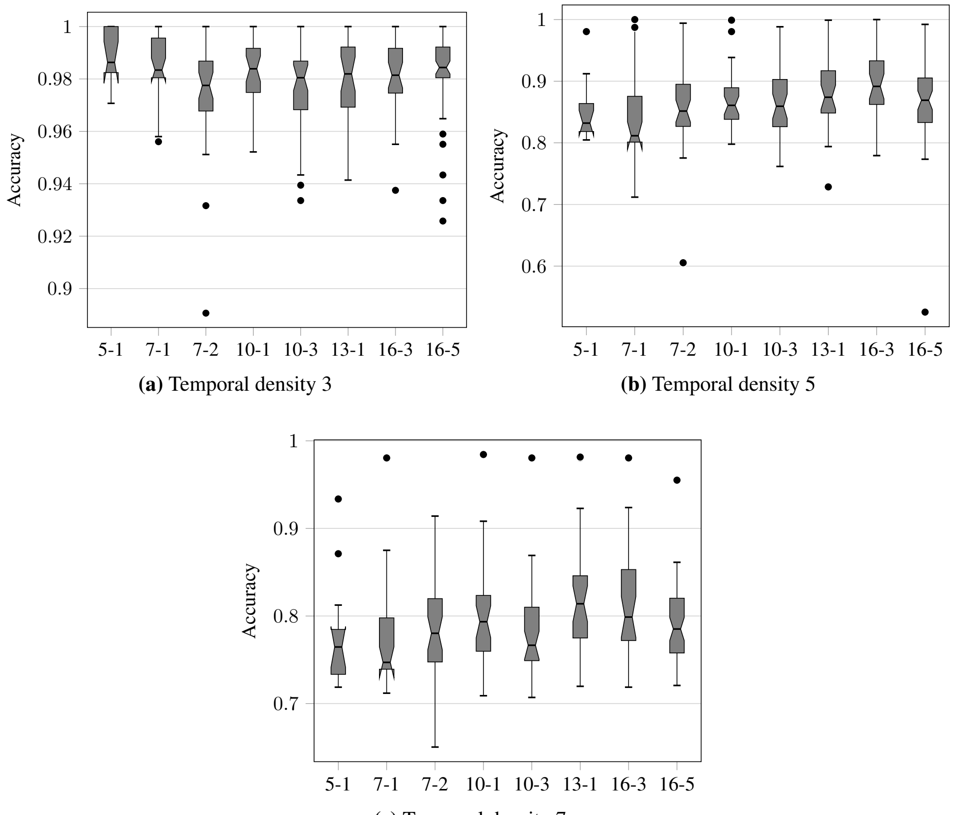
5.2.2 Temporal Density

This experiment tests the performance of the 256 programs that achieved above 90% at on the temporal parity 3 task. Each program was evaluated using temporal density 3, 5 and 7.

Table 5.4 lists the configuration parameters used for the readout layer.

Setting	**Value**
Max epochs	4000
Learning rate	0.03
Hidden layer activation	Sigmoid
Output layer activation	Softmax
Hidden layer nodes	16
Loss function	Multi-Class Cross Entropy
Updater	Nesterov's Accelerated Gradient (NAG)
Momentum μ	0.9

Table 5.5: Temporal density readout parameters



(a) Temporal density 3

(b) Temporal density 5

(c) Temporal density 7

Figure 5.3: Readout layer accuracy vs program configuration on temporal density 3, 5 and 7

Figure 5.3 shows the readout layer accuracy for SMCGP programs with different number of output nodes and IEF on the temporal density 3, 5 and 7 tasks.

The results were gathered by iterating each program with 24 bit streams of 128 random bits each, then splitting into a training set of 2048 samples and validation and test sets of 512 samples respectively. The readout layer was trained for 4000 epochs or until an accuracy ≥ 0.98 was reached.

5.2.3 NARMA 10

This experiment tests the performance of 530 evolved programs from 8 different configurations, a total of 4240, on the NARMA10 benchmark. The programs used were the same programs evolved in experiment 5.1. This experiment uses a single regression layer as readout.

Table 5.6 lists the configuration parameters used for the readout layer.

Setting	**Value**
Max epochs	2000
Learning rate	0.035
Hidden layer activation	None
Output layer activation	Identity
Hidden layer nodes	None
Loss function	Mean Squared Error
Updater	SGD
Momentum μ	None

Table 5.6: NARMA10 readout parameters

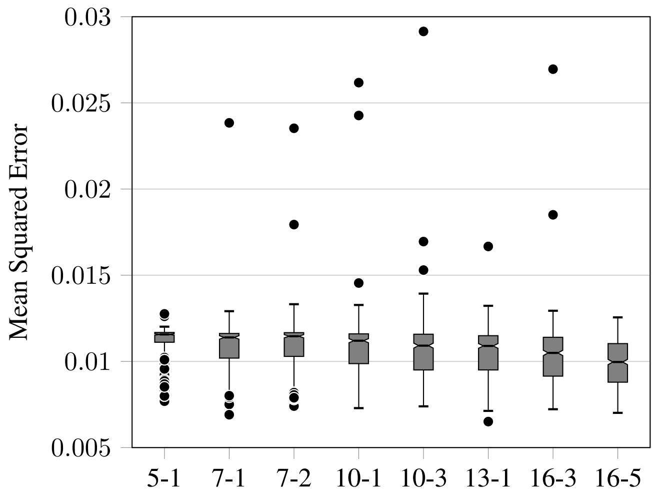


Figure 5.4: Mean Squared Error (MSE) vs program configuration on NARMA10

Figure 5.4 shows the Mean Squared Error (MSE) with different number of output nodes and IEF on 10th order NARMA task.

The results were gathered by iterating each program with 16 NARMA10 time series of length 256. The readout layer was trained 15 times, each for 2000 epochs, and randomly split into a training set of 2048 samples and validation and test sets of 1024 samples respectively. The error values were then averaged over these 15 runs.

Table 5.7 lists the mean and standard deviation for each group of 530 programs in terms of Mean Squared Error (Equation 5.1), Root Mean Squared Error (Equation 5.2), Normalized Root Mean Squared Error (Equation 5.3) and Mean Absolute Error (Equation 5.4).

Group	**MSE**	**RMSE**	**NRMSE**	**MAE**
5-1	0.011081±0.001066	0.105108±0.005307	0.136401±0.007155	0.083567±0.004835
7-1	0.010844±0.001277	0.103920±0.006009	0.134718±0.008020	0.082425±0.005202
7-2	0.010945±0.001943	0.104224±0.007150	0.135233±0.009473	0.082528±0.005385
10-1	0.010830±0.003501	0.103315±0.008372	0.133806±0.011022	0.081663±0.005594
10-3	0.010565±0.001535	0.102461±0.006737	0.132822±0.008845	0.080983±0.005633
13-1	0.010542±0.002840	0.102106±0.008360	0.132478±0.010976	0.080786±0.005908
16-3	0.010316±0.001494	0.101215±0.006575	0.131187±0.008705	0.079868±0.005708
16-5	0.009908±0.001264	0.099305±0.006375	0.128826±0.008398	0.078223±0.005755

Table 5.7: Error measure vs program configuration on NARMA10

The four error measures used in Table 5.7 are described in the following equations. y is the prediction made by the readout layer, $\hat{y}$ is the target value and n is the number of samples in the data set.

$$MSE = \frac{1}{n}\sum_{t=1}^{n}(y-\hat{y})^2 \tag{5.1}$$

$$RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(y-\hat{y})^2} \tag{5.2}$$

$$NRMSE = \frac{\sqrt{\frac{1}{n}\sum_{t=1}^{n}(y-\hat{y})^2)}}{max(\hat{y}) - min(\hat{y})} \tag{5.3}$$

$$MAE = \frac{1}{n}\sum_{t=1}^{n}|(y-\hat{y})| \tag{5.4}$$

Chapter 6

Discussion

In this chapter the results from the experiments will be reviewed. The results from the temporal parity and temporal density experiments are discussed in light of the work of Langton and Wolfram. Characteristics of the best performing programs are inspected in terms of spectral analysis, state transition and self-modification operators.

6.1 Experiment Results

The results from chapter 5 show that it is possible to evolve SMCGP programs possessing enough complexity to solve the temporal parity 7 task. Although most of the programs only managed to solve the simpler tasks, a few performed better. On average the same programs fared better in the temporal density problem for all sliding window sizes, as expected given the highly non-linear nature of the parity problem. Of all the programs, only a handful were able to achieve an accuracy over 90% in temporal parity $n = 5$, and just 3 programs on difficulty $n = 7$. The average was about 65% for $n = 5$ and 55% for $n = 7$, just above random.

On the 10th-order NARMA benchmark the best programs achieved error measures of 0.0065 and 0.0069 (Mean Squared Error) respectively. Collectively, the different configurations averaged at about 0.012. This shows that none of the 4240 programs evolved with the configuration in this experiment possessed enough difference in *separation* and *fading memory* to be comparable to other, state-of-the-art reservoirs. However, different configurations with different dynamics may be evolved that perform better on this problem.

6.2 Searching for the Edge of Chaos

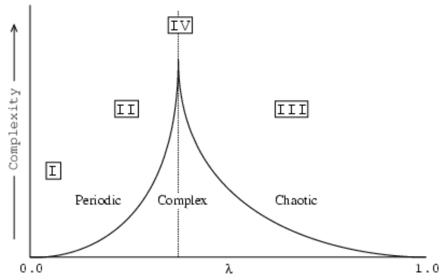
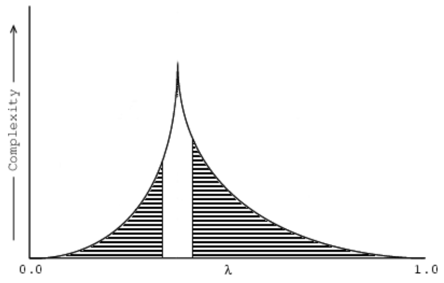
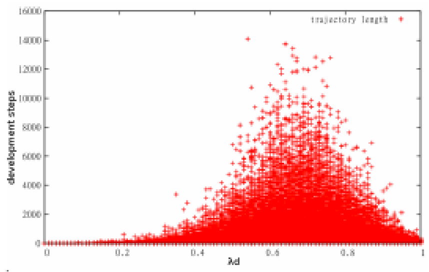


Figure 6.1: Phase transition search space

As stated in chapter 2, the dynamic systems with the highest potential for complex computation, reside in the phase transition between order and chaos, at the very top of the graph shown in Figure 6.1. Meaning that in a set of all possible dynamic behaviours, these are very sparse and the search space very large.



(a) Reduced search space by narrowing paramter, λ



(b) Density of solution space (from [60])

Figure 6.2: Search Space of ”the edge of chaos”

In order to reduce the set of possible dynamic behaviour in the experiments, the EA search was equipped with a strict composite fitness function used to ensure that the SMCGP programs were as close to the critical point as possible before they were tested. Figure 6.2a illustrates an example of how the search space was constricted by the EA. The striped area represent SMCGP programs who got penalised due to undesired dynamic, while the white area represent the programs who passed the EA search. Figure 6.3 shows the average number of phenotypes evaluated before a candidate which passed the fitness function was found, these are the ones caught in the striped area.

Even though the EA search greatly narrowed down the set of possible systems, the white area in Figure 6.2a still represent a large amount. The level of complexity in a program was estimated using the aforementioned benchmarks. Thus the EA only provided the means to search for programs by their order of chaos, not complexity. Further, depicted

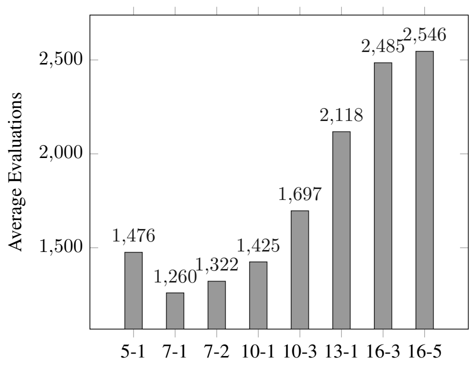


Figure 6.3: Average EA evaluations per evolved program

in Figure 6.2b, the density of dynamic systems with low complexity is far greater than the ones with high complexity. Meaning that even though the system is able to narrow down the phase transition, programs with high complexity are still infrequent, and hard to find.

Considering the above, one might say that it is not surprising that the programs which managed to solve the more difficult problems were outliers, while the majority of the programs possessed too little complexity. It seems to fall into what is expected on the basis of the work of Wolfram and Langton.

Outputs-IEF	**Evaluated**	**Stable**	**Converged**	**Few states**	**Input SM**	**INP**	**SH**	**T**	**UV**
5-1	73799	1758	2579	61863	1429	211	114	2874	3159
7-1	62995	1543	1811	56135	1256	135	100	725	1290
7-2	64763	1182	1601	59761	139	191	181	550	1150
10-1	71236	718	2003	66675	61	246	135	798	600
10-3	86167	526	2282	79610	1200	716	545	465	809
13-1	105859	449	3721	99250	277	295	316	974	577
16-3	124977	234	4943	116680	330	249	290	1918	310
16-6	129420	35	5131	118719	0	303	26	5180	23

Table 6.1: EA Search Terminations

Table 6.1 lists a breakdown of termination reasons for 50 evolved phenotypes. The majority (91.6%) of terminations are due to the program not performing the minimum required number of self-modifications. The reasons listed are:

Stable: Enough states have been seen, but no changes were made during the last τ time steps

Converged: More than $\frac{1}{3}$ of output nodes had converged within the threshold ϵ

Few States: Fewer than the minimum configured number of states were seen across all input streams

Input/SM: The phenotype removed its own input or SM functions

INP: The percentage of input nodes in the graph was too low

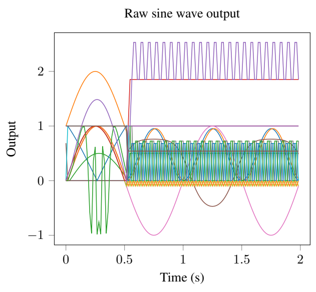
SH: Less than the required number of harmonic frequencies were detected

T: The phenotype timed out during evaluation (short average connection length or heavy SM operations)

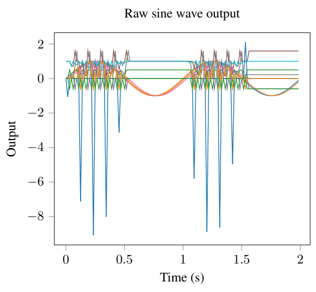
UV: Too few unique output values were detected after processing all input streams

6.3 Program Characteristics

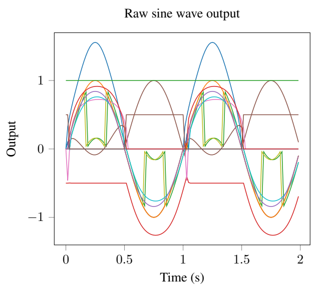
In this section, 50 programs with the best performance in the temporal parity and density experiments were selected for further analysis.



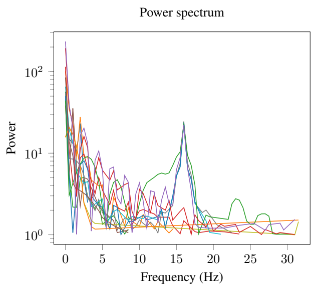
(a) Sine Wave 1



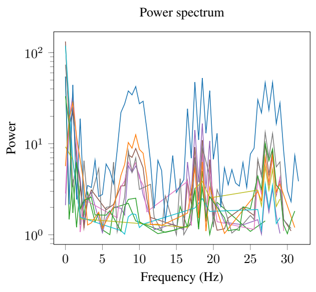
(b) Sine Wave 2



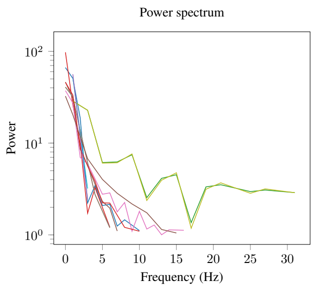
(c) Sine Wave 3



(d) Power Spectrum 1



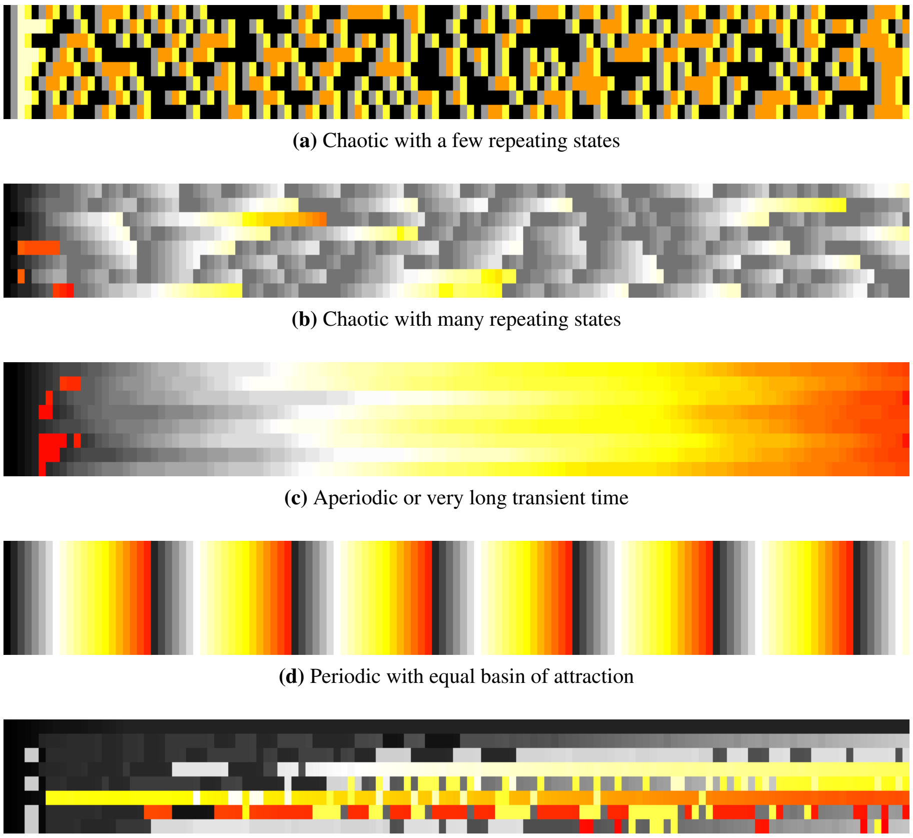
(e) Power Spectrum 2



(f) Power Spectrum 3

Figure 6.4: Power spectrum and Sine waves from a sample of the best programs

Based on the 50 programs that were analysed, there was no clear pattern of sine wave transformation or distribution of harmonic frequencies. Many of the programs did exhibit alternating patterns of amplitude, like seen in Figure 6.4a. Slight phase shift or changes purely in amplitude were also common. However, the number of output nodes with these common traits varied greatly among the programs. The analysis failed to detect any correlation between these occurrences and program performance on temporal parity and temporal density. Figure 6.4 shows three examples from the analysis, illustrating the great differences between the programs, in raw wave output and power spectrum density.


(a) Chaotic with a few repeating states

(b) Chaotic with many repeating states

(c) Aperiodic or very long transient time

(d) Periodic with equal basin of attraction

(e) Chaotic with different basins of attraction

Figure 6.5: Most common patterns of the 50 best performing programs

Figure 6.5 lists five types of topological change patterns that appeared at least 4 times in the sample set of 50 programs. The different types are listed in order from most frequent, to least frequent. There were still many other unique patterns in the set, but the ones presented were similar to at least 3 other programs.

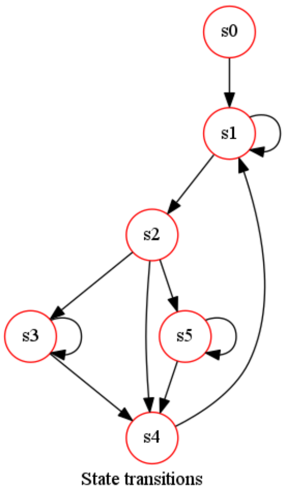


Figure 6.6: State transition diagram for Figure 6.5a

Figure 6.5a is by far the most common pattern, and its variants make up just under 30% of the programs analysed. Figure 6.6 shows the state transition diagram for Figure 6.5a. Most of the programs with this type of change pattern have small and highly connected transition diagrams. However, beyond this slight trend, no indicative correlation between state change patterns and performance could be found. All these programs performed relatively well, with high performing programs found in most of the pattern types. It must be said that the sample size here is very low, and these groups may very well be smeared together in a gradient of differing patterns if the sample size is increased. The programs also differ by several parameters like output node count and input expansion factor, complicating the matter further.

When assessing frequency of SM node activation, three types make up the majority of SM operations. These are *SM-Duplicate*, *SM-Move* and *SM-ShiftConnections*. The remaining functions have more sporadic appearances in the activation logs. This is most likely due to the nature of how these functions operate, on ranges of nodes. *SM-Move* may move a block of nodes five indexes forward, shifting the graph. Next iteration it moves a separate group of nodes, while at the third iteration, the nodes from the first move has been shifted back in place. This type of rotation would pass fitness checks, while often being stable enough to allow for computation. The same type of behaviour is found in ShiftConnections, where node connections are shifted in ranges. The effect of Duplicate is a bit more uncertain, as one also has to consider the growth factor of the program.

In summary, no clear correlation could be found between the characteristics presented and performance of the analysed programs on the benchmarks. A more thorough investigation with greatly reduced diversity in program configuration would have fared a better chance of identifying some correlation.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The research presented in this thesis is centred on the use of SMCGP, networks with self-organising topology, in a reservoir computing environment.

Dynamic systems, especially the ones with dynamic behaviour in the region known as ”the edge of chaos”, are thought to have the most computational capability. Finding a way of identifying, constructing and utilising these systems efficiently can lead to major advances in artificial intelligence and computing.

The motivation behind this research was that SMCGP programs might be a suitable model for the plasticity exhibited by biological systems. Should this be the case, further research into SMCGP-RC could be beneficial for several topics, especially biological-digital hybrid platforms.

In work on this thesis, the authors have developed an experimental platform (described in chapter 4) capable of generating SMCGP programs that possess certain dynamic properties. These programs were then tested by performing different benchmarks to evaluate the viability of SMCGP-RC.

The results presented in chapter 5 show that SMCGP programs with continuously altering topology can be used as a reservoir of dynamics and utilised for computation. The aforementioned is demonstrated for the temporal parity, temporal density and NARMA10 problems with the disclosed parameters.

Although possible, finding the SMCGP programs which solve the benchmarking problems proved to be hard. As outlined in chapter 3 the combination of all components that make up a SMCGP program is vast, and the results show that only a small part of that vastness has high enough complexity to solve the more difficult variants of the benchmark problems.

The authors fail to identify any strong correlation between the programs performing best on benchmarks and characteristics such as state transition patterns and power spectrum harmonics. A wide range of different characteristics were found in programs that were able to solve the temporal parity 5 task. However, the most frequently activated SM-nodes in the experiments were *Move*, *Duplicate* and *ShiftConnections*.

7.2 Contributions

The results in this thesis have shown that it is possible to accomplish computation by exploiting the dynamics of continuously altering SMCGP programs in a reservoir context. This was demonstrated by evolving programs that frequently underwent topological changes, while still being able to exhibit some computational capability.

The programs that managed to solve the temporal parity problems demonstrates that when presented with output from the previous time step, SMCGP programs can exhibit the necessary dynamic properties to solve non-linear temporal problems.

By implementing feedback, these experiments have also explored the effects of using multiple, possibly unique input values, in SMCGP programs. As noted in [50], the effect of multiple inputs on SM-activation is ”problematic”. The experimental results confirm the assumption that evolution can find solutions with stable activation patterns.

As stated in chapter 1, the authors could find no prior publications that cover the use of SMCGP in a reservoir computing environment. These results could therefore inspire others to investigate the possibilities of SMCGP-RC.

The testing platform described in chapter 3 and chapter 4 is open source, and publicly available[1]. This can be used by other researchers for future or related work .

7.3 Future Work

From the experiments shown in chapter 5, relatively few graph configurations were used compared to the other experiments in the literature, where only SMCGP is concerned [53, 55]. An exploratory search of graph configuration and function set would be highly beneficial. An additional note is that the maximum connection length for each node was set to half the size of the graph. This results in an average recursion depth of 4, and was chosen to reduce the average time needed to evaluate each output node. An investigation into the effects of lowering this connection length is also an area of much interest.

During the experiments many of the parameters were kept unchanged for the sake of simplicity, some of these parameters might prove to be very influential. An effort to improve the current configuration with a parameter search could prove beneficial.

[1] https://github.com/krekle/cerebrum-medulla

In chapter 3, it is mentioned that SMCGP does not possess inherent memory to solve the benchmarking problems without a ”helping” hand from the feedback channel of the reservoir. This provides memory, but can also interfere with reading of input, hence the inclusion of the input expansion factor. One thought worth investigating is adding a new type of input node to the SMCGP programs, which is responsible for reading the feedback signal. That way the traditional input nodes are not be affected, and evolution can find an appropriate balance on its own. Expanding the SM and mathematical function sets are also something that should be investigated. Several additional functions are presented in [61] which could improve the ability for complex behaviour in the SMCGP programs.

The EA search used to evolve SMCGP programs in this thesis is a basic implementation consisting only of point based mutation and single point crossover. This process could be made more efficient by introducing more information to the EA, as well as improving the genotypic representation. The notion of self-repair, growth regulation and mimicing gene regulatory networks in general could allow the EA to develop its own representation best suited for the target behaviour. Evolutionary developmental biology (*Evo-Devo*) is an EA technique inspired from how biological organisms develop from a single cell to a developed individual. [60, 62] outlines how EvoDevo systems with evolvable genome and regulatory processes can be used to develop candidates that stand a high chance of fulfilling a target goal. Utilising such an approach could greatly improve the SMGCP evolutionary search.

The chosen SMCGP version implemented was the original, not SMCGP2 [56]. SMCGP2 is said to have advantages such as increased performance on the even parity task, and a simplified function set. A comparative analysis of the two in an RC environment could prove beneficial.

Bibliography

[1] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[2] J. H. Holland, "Adaptation in natural and artificial systems. an introductory analysis with application to biology, control, and artificial intelligence," *Ann Arbor, MI: University of Michigan Press*, 1975.

[3] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," *Machine learning*, vol. 3, no. 2, pp. 95–99, 1988.

[4] M. M. Millonas, "Swarms, phase transitions, and collective intelligence," in *SANTA FE INSTITUTE STUDIES IN THE SCIENCES OF COMPLEXITY-PROCEEDINGS VOLUME-*, vol. 17, pp. 417–417, ADDISON-WESLEY PUBLISHING CO, 1994.

[5] P. J. Bentley, *Digital Biology: How Nature Is Transforming Our Technology and Our Lives*. Simon & Schuster Trade, 2002.

[6] S. H. Martinius Knudsen, "Ntnu cyborg: A study into embodying neuronal cultures through robotic systems," Norwegian University of Science and Technology, Department of Engineering Cybernetics, 2016.

[7] R. V. S. N. I. S. G. T. . M. Knudsen, O. Ramstad, "Towards making a cyborg: A closed-loop reservoir-neuro system," Norwegian University of Science and Technology, Department of Engineering Cybernetics, 2017 (submitted).

[8] D. Verstraeten, B. Schrauwen, M. d'Haene, and D. Stroobandt, "An experimental unification of reservoir computing methods," *Neural networks*, vol. 20, no. 3, pp. 391–403, 2007.

[9] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *Neural Networks, 1996., IEEE International Conference on*, vol. 1, pp. 347–352, IEEE, 1996.

[10] H. Jaeger, “The “echo state” approach to analysing and training recurrent neural networks-with an erratum note,” *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, vol. 148, p. 34, 2001.

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

[12] P. F. Dominey, “Complex sensory-motor sequence learning based on recurrent state representation and reinforcement learning,” *Biological cybernetics*, vol. 73, no. 3, pp. 265–274, 1995.

[13] J. J. Steil, “Backpropagation-decorrelation: Online recurrent learning with o (n) complexity,” in *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, vol. 2, pp. 843–848, IEEE, 2004.

[14] D. V. Benjamin Schrauwen and J. V. Campenhout, “An overview of reservoir computing: theory, applications and implementations,” Electronics and Information Systems Department, Ghent University, Belgium, 2002.

[15] B. Chopard and M. Droz, *Cellular automata.* Springer, 1998.

[16] S. A. Kauffman, “Metabolic stability and epigenesis in randomly constructed genetic nets,” *Journal of theoretical biology*, vol. 22, no. 3, pp. 437–467, 1969.

[17] S. A. Kauffman, *The origins of order: Self-organization and selection in evolution.* Oxford University Press, USA, 1993.

[18] S. Harding, J. F. Miller, and W. Banzhaf, “Developments in cartesian genetic programming: self-modifying cgp,” *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 397–439, 2010.

[19] J. F. Miller and P. Thomson, “Cartesian genetic programming,” in *European Conference on Genetic Programming*, pp. 121–132, Springer, 2000.

[20] T. B. D. Steve M. Potter *, “A new approach to neural cell culture for long-term studies,” 2001.

[21] M. C. Paolo Massobrio, Jacopo Tessadori and M. Ghirardi, “In vitro studies of neuronal networks and synaptic plasticity in invertebrates and in mammals using multielectrode arrays,” 2015.

[22] C. Grebogi, E. Ott, and J. A. Yorke, “Chaos, strange attractors, and fractal basin boundaries in nonlinear dynamics,” *Non-Linear Physics for Begginers: Fractals, Chaos, Pattern Formation, Solutions, Cellular Automata and Complex Systems*, pp. 111–117, 1998.

[23] S. Strogatz, M. Friedman, A. J. Mallinckrodt, S. McKay, *et al.*, “Nonlinear dynamics and chaos: With applications to physics, biology, chemistry, and engineering,” *Computers in Physics*, vol. 8, no. 5, pp. 532–532, 1994.

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

[25] S. H. Strogatz, “Exploring complex networks,” *Nature*, vol. 410, no. 6825, pp. 268–276, 2001.

[26] R. Legenstein and W. Maass, “Edge of chaos and prediction of computational performance for neural circuit models,” *Neural Networks*, vol. 20, no. 3, pp. 323–334, 2007.

[27] S. Wolfram *et al.*, “Cellular automata as models of complexity,” *Nature*, vol. 311, no. 5985, pp. 419–424, 1984.

[28] C. G. Langton, “Computation at the edge of chaos: phase transitions and emergent computation,” *Physica D: Nonlinear Phenomena*, vol. 42, no. 1-3, pp. 12–37, 1990.

[29] S. A. Kauffman, *Investigations*. Oxford University Press, 2000.

[30] C. Gershenson, “Introduction to random boolean networks,” *arXiv preprint nlin/0408006*, 2004.

[31] B. A. Pearlmutter, “Gradient calculations for dynamic recurrent neural networks: A survey,” *IEEE Transactions on Neural networks*, vol. 6, no. 5, pp. 1212–1228, 1995.

[32] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks.,” *ICML (3)*, vol. 28, pp. 1310–1318, 2013.

[33] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[34] M. Lukoševičius, H. Jaeger, and B. Schrauwen, “Reservoir computing trends,” *KI - Kunstliche Intelligenz*, no. 4, pp. 365–371, 2012.

[35] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[36] J. Martens and I. Sutskever, “Learning recurrent neural networks with hessian-free optimization,” in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1033–1040, 2011.

[37] T. Back, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[38] D. Whitley, “An overview of evolutionary algorithms: practical issues and common pitfalls,” *Information and software technology*, vol. 43, no. 14, pp. 817–831, 2001.

[39] M. Črepinšek, S.-H. Liu, and M. Mernik, “Exploration and exploitation in evolutionary algorithms: A survey,” *ACM Computing Surveys (CSUR)*, vol. 45, no. 3, p. 35, 2013.

[40] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*, vol. 1. MIT press, 1992.

[41] J. F. Miller, “Cartesian genetic programming,” in *Cartesian Genetic Programming*, pp. 101–123, Springer, 2011.

[42] J. Grefenstette, R. Gopal, B. Rosmaita, and D. Van Gucht, “Genetic algorithms for the traveling salesman problem,” in *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pp. 160–165, 1985.

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

[44] W. Maass, T. Natschläger, and H. Markram, “Computational models for generic cortical microcircuits,” *Computational neuroscience: A comprehensive approach*, vol. 18, p. 575, 2004.

[45] T. Natschläger, N. Bertschinger, and R. Legenstein, “At the edge of chaos: Real-time computations and self-organized criticality in recurrent neural networks,” in *Proc. of NIPS*, pp. 145–152, 2004.

[46] D. Snyder, A. Goudarzi, and C. Teuscher, “Computational capabilities of random automata networks for reservoir computing,” *Physical Review E*, vol. 87, no. 4, p. 042808, 2013.

[47] C. Fernando and S. Sojakka, “Pattern recognition in a bucket,” in *Advances in artificial life*, pp. 588–597, Springer, 2003.

[48] D. Nikolic, S. Haeusler, W. Singer, and W. Maass, “Temporal dynamics of information content carried by neurons in the primary visual cortex,” in *NIPS*, pp. 1041–1048, 2006.

[49] T. E. Gibbons, “Reservoir computing: a rich area for undergraduate research,” in *Midwest Instruction and Computing Symposium*, pp. 16–17.

[50] S. L. Harding, J. F. Miller, and W. Banzhaf, “Self-modifying cartesian genetic programming,” in *Cartesian Genetic Programming*, pp. 101–124, Springer, 2011.

[51] S. Harding, J. F. Miller, and W. Banzhaf, “Developments in cartesian genetic programming: self-modifying cgp,” *Genetic Programming and Evolvable Machines*, vol. 11, no. 3, pp. 397–439, 2010.

[52] S. Harding, J. F. Miller, and W. Banzhaf, “A survey of self modifying cartesian genetic programming,” in *Genetic Programming Theory and Practice VIII*, pp. 91–107, Springer, 2010.

[53] S. Harding, J. F. Miller, and W. Banzhaf, “Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing,” in *European Conference on Genetic Programming*, pp. 133–144, Springer, 2009.

[54] S. Harding, J. F. Miller, and W. Banzhaf, "Self modifying cartesian genetic programming: Parity," 2009.

[55] S. Harding, J. F. Miller, and W. Banzhaf, "Self modifying cartesian genetic programming: finding algorithms that calculate pi and e to arbitrary precision," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pp. 579–586, ACM, 2010.

[56] S. Harding, J. F. Miller, and W. Banzhaf, "Smcgp2: self modifying cartesian genetic programming in two dimensions," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pp. 1491–1498, ACM, 2011.

[57] S. Harding, J. F. Miller, and W. Banzhaf, "Smcgp2: finding algorithms that approximate numerical constants using quaternions and complex numbers," in *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pp. 197–198, ACM, 2011.

[58] H. Nyquist, "Certain topics in telegraph transmission theory," *Transactions of the American Institute of Electrical Engineers*, vol. 47, no. 2, pp. 617–644, 1928.

[59] Y. Nesterov *et al.*, "Gradient methods for minimizing composite objective function," 2007.

[60] G. Tufte and S. Nichele, "On the correlations between developmental diversity and genomic composition," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pp. 1507–1514, ACM, 2011.

[61] S. Harding, J. F. Miller, and W. Banzhaf, "Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing," in *European Conference on Genetic Programming*, pp. 133–144, Springer, 2009.

[62] S. Nichele, A. Giskeødegård, and G. Tufte, "Evolutionary growth of genome representations on artificial cellular organisms with indirect encodings," *Artificial life*, 2016.

[63] P. Whittle, *Hypothesis testing in time series analysis*, vol. 4. Almqvist & Wiksells, 1951.

Appendix A

Benchmarking problems

A.1 Temporal Parity and Temporal Density

In [46], two tasks are formulated to determine the ability of a system to separate input streams and exhibit fading memory. These tasks are known as *temporal parity* and *temporal density*.

The *temporal parity* task determines if a number of input bits n in a bit stream u have had an even or odd number of ”1”s. The sliding window n can be delayed by a number of time steps τ. The formal definition for a bit stream u where $|u| = T$, a delay τ and a sliding window $n \geq 1$ is

$$A_n(t) = \begin{cases} u(t-\tau), & \text{if } n = 1 \\ \oplus_{i=0}^{n-1} u(t-\tau-i), & \text{otherwise} \end{cases} \tag{A.1}$$

where $\tau + n \leq t \leq T - \tau - n$.

Figure A.1 illustrates the correct input to output mapping for the temporal parity task with a sliding window size of 3 and no delay.

Input: $0,0,1,0,1,1,1,0,0,1,0,1,0,0,1,0,1,1,0,0$
Output: $0,0,1,1,0,0,1,0,1,1,1,0,1,1,1,1,0,0,0,1$

Figure A.1: Temporal Parity with $n = 3$, $\tau = 0$

The *temporal density* task determines if a number of input bits n in a bit stream u have had a majority of ”1”s. The sliding window n can be delayed by a number of time steps τ. The formal definition for a bit stream u where $|u| = T$, a delay τ and a sliding window $n \geq 1$ is

$$B_n(t) = \begin{cases} 1, & \text{if } 2\sum_{i=0}^{n-1} u(t-\tau-i) > n \\ 0, & \text{otherwise} \end{cases} \tag{A.2}$$

where $\tau + n \leq t \leq T - \tau - n$.

Figure A.2 illustrates the correct input to output mapping for the temporal density task with a sliding window size of 3 and no delay.

Input: $0,0,1,0,1,1,1,0,0,1,0,1,0,0,1,0,1,1,0,0$
Output: $0,0,0,0,1,1,1,1,0,0,0,1,0,0,0,0,1,1,1,0$

Figure A.2: Temporal Density with $n = 3$, $\tau = 0$

In [46], it is speculated that the *separation capability* is more important than fading memory for the temporal parity task. This is due to the fact that the parity operator $\oplus$ is not linearly separable, making the task for large window sizes highly non-linear. The temporal parity task is therefore considered more difficult than the temporal density task, as it requires more memory.

A.2 NARMA10

The general *Auto-Regressive Moving Average* (ARMA) is a model used in statistical analysis of time series, and was introduced in [63]. In this general model, the dependence of X_t on past values is linear. If the dependence is non-linear, it is specified as *Non-linear Auto-Regressive Moving Average* (NARMA). The number associated describes the order of dependence in time steps. NARMA10 and NARMA30 are frequently used in RC literature to test the memory capacity and computational power of a reservoir.

The 10-th order NARMA function is defined as:

$$y(t) = \alpha y(t-1) + \beta y(t-1)\sum_{i=1}^{n} y(t-i) + \gamma x(t-n)x(t-1) + \delta \tag{A.3}$$

where $y(t)$ is the output at time step t, $x(t)$ is the input at time step t, $n = 10$, $\alpha = 0.3$, $\beta = 0.05$, $\gamma = 1.5$ and $\delta = 0.1$. The input $x(t)$ is sampled from a uniform distribution $[0, 0.5]$.

Input: $0.400, 0.076, 0.118, 0.322, 0.407, 0.287, 0.368, 0.313, 0.472, 0.109, 0.200, 0.163, 0.129,$
$0.196, 0.076, 0.169, 0.018, 0.485, 0.468, 0.067$
Output: $0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.166, 0.174, 0.184,$
$0.222, 0.295, 0.236, 0.279, 0.214, 0.526, 0.395$

Figure A.3: NARMA model of the 10th order for $0 \leq t < 20$

Figure A.3 illustrates an input to output mapping for the NARMA10 task. The first n values are normally truncated and is considered reservoir warm-up.

Appendix B

Experimental platform configuration options

B.1 Available Configuration Options

Table B.1: Configurable options

Setting	**Description**
task	”generateNew”,”matrix”,”narma”,”rerun”
trainingSamples	The total number of samples in the training sets
testSamples	The total number of samples in the test sets
minPowerCoefficient	Min power spectrum coefficient γ
dataset.type	*temporalParity*, *temporalDensity* or *narma10*
dataset.normalize	*true* if data should be scaled to $-1.0 \leq x \leq 1.0$
delay	Delay for temporal tasks
windowSize	Sliding window size for temporal tasks
functionSM	Directory for SM functions
functionMath	Directory for math functions
functionMath.whiteList	The math function set used
realnumbersLower	Lower bound for parameters $P_0...P_2$
realnumbersUpper	Upper bound for parameters $P_0...P_2$
todoSize	Size of the ToDo-list
startNodes	Number of nodes in the genotype at $t = 0$
levelsback	Max relative connection length
maxNodesMovedOrDuplicated	Limits the number nodes per mov/dup
smcgp.outputs	Number of output SMCGP nodes
smcgp.timeout	Timeout in ms per input
smcgp.maxGrowthFactor	Limits the size of the phenotypes
smcgp.useSimpleInputFunction	Only uses INP function
smcgp.functionMath	Enabled math functions
smcgp.functionSM	Enabled self-modifying operators

Continued on next page

Table B.1 – *Continued from previous page*

Setting	**Description**
phenotype.minFitness	Lower bound for when to store candidate
minPowerSpectrumStates	Min number of distinct harmonics
powerSpectrumRoundingThreshold	Rounds values to nearest delta
useNumericalConvergenceCheck	true or false
pointAttractorStepsBack	Num steps to check backwards
numericalConvergenceStepsBack	Num steps to check backwards
numericalConvergenceRoundingThreshold	Considers numbers equal within this num
minUniqueValues	Minimum number of unique values
minTopographicStates	Min number of states seen
maxTopographicStates	Max number of states seen
minInputNodePercentage	Enforce a minimum of input nodes in genotype
ea.maxEpochs	Evolutionary epochs
ea.stopCondition	Must currently be 64
ea.mutationRate	A number $0 < i \leq 1.0$
ea.initialPopulation	Starting population
ea.poolSize	Number of candidates per epoch
ea.crossover	Use crossover
ea.crossoverRate	A number $0 < i \leq 1.0$
ea.elitism	true if best candidate is passed on unmutated
ea.parentPool	Size of the parent selection pool
ea.parentSelectionScheme	”tournament”,”best”,”4+1”
ea.tournamentSize	Size of the tournament rounds
signal.sampleSize	Sample size of sine wave
signal.amplitude	Amplitude of sine wave
signal.phase	Phase shift of sine wave
signal.frequency	Frequency of sine wave
signal.overSamplingFactor	Multiply number of samples for accuracy
reservoir.epochs	Max epochs when training readout
reservoir.evaluateEveryNumEpochs	Print epoch stats every n epochs
numCachedTimeSteps	Number of time steps t in feedback
useInputExpansion	true if using inputexp, false if not
inputExpansionFactor	Integer $i > 0$ to scale input presence
readout.stopCondition	Accuracy threshold $0.0 \leq a \leq 1.0$
readout.batchSize	Training mini-batch size
readout.hiddenLayerOneNodeCount	Num nodes in hidden layer
readout.learningRate	Real number $0.0 < lr < 1.0$
readout.momentum	Momentum factor in Nesterov’s (NAG)
readout.quarterWayFitnessCutoff	Stop if accuracy not over this val at 1/4 epochs
readout.halfWayFitnessCutoff	Stop if accuracy not over this val at 1/2 epochs
readout.useFitnessTrainingCutoff	*true* if aborting if fitness is below above parameters
readout.useMultiLayerForRegression	*true* if regression should use multilayer readout

B.2 Configuration Parameters Common for all Experiments

The following tables of configuration parameters were fixed and remained unchanged across all experiments.

Table B.2 shows the EA search parameters that were common for all experiments.

Setting	Value
Max epochs	1000
Mutation Rate	0.22
Crossover	*true*
Crossover Rate	0.1
Population Size	20
Initial population size	40
Elitism	*true*
Parent pool size	8
Parent selection scheme	*tournament selection*
Tournament Size (k)	5
Max nodes moved/duplicated (per input)	5

Table B.2: Common evolutionary search parameters

Table B.3 shows the SMCGP parameters that were common for all experiments.

Setting	Value
Max growth factor	3.0
Todo-list size	1
SM params lower bound	-40.0
SM params upper bound	40.0
Timeout (per input)	500ms
Max nodes moved/duplicated (per input)	20
Enabled Math functions	Add, Constant, Cos, Divide, Max, Mean, Min, Modulo, Multiply, Pow, Sin, Square, Subtract, Tan, Tanh
Enabled SM operations	SMchc, SMchf, SMchp SMdel, SMdup, SMmov, SMshiftc

Table B.3: Common SMCGP parameters