Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

B.A.I Electronic Engineering Research Project

2022

Studying Chaotic Systems with Hardware Reservoir Computing

Submitted as part of EEU44EE2 – Electronic Engineering Project

"I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write."

Alexandr Goultiaev Tolstokorov

Project Supervisor: Harun Siljak

Contents

Lay Abstract

Dynamical and chaotical systems have been attributed to vast amounts of phenomena in most scientific and mathematic research fields. However, these systems are notorious for being difficult to predict with mathematical theory. Recently a new approach has emerged from the availability of large datasets, this approach uses machine learning techniques when no explicit model can be formulated. However, these machine learning techniques need hardware implementations for the eventual use in real-world engineering operations. I focus on Reservoir Computing as a hardware oriented subset of Artificial Neural Networks with the ability to model and learn these dynamic and chaotic systems. Chaotic systems have been observed to have the ability to synchronize, which opens a possibility for communication between two synchronized chaotic systems called chaotic communication. This research paper proposes a Low-Cost hardware implementation of a Reservoir Computer on reprogrammable logic electronic circuit boards to act as a chaotic system for the use in chaotic communications.

Abstract

This research paper proposes a hardware-oriented Reservoir Computer topology called Cyclic Reservoir that is able to achieve close to state-of-the-art performance for chaotic time series prediction. A hardware implementation targeting FPGAs using Stochastic Computing elements is then proposed that is able to substantially reduce the hardware resource demand of the network and avails of a high robustness to environmental noise. A chaotic communication channel using the hardware implementation of the Reservoir Computer is proposed that is more secure than analog chaotic circuit communication channels and plausibly more efficient in real-world use than other Artificial Neural Network based chaotic communication channels.

1. Motivation and Objectives

1.1. Motivation

The modern-day world produces an ever-increasing demand for intelligent machine learning systems to perform a vast number of tasks for our daily lives. This demand is met with the rise of Artificial Neural Networks, however for the technology to be truly ever-present in daily life hardware implementations of it must be developed. Reservoir Computing rises as a plausible solution for hardware implementations, and I am interested in their potential use in Chaotic Communication.

1.2. Objective

The Objectives set out at the beginning of the project are as follows:

I. Study Reservoir Computing.

II. Design a Reservoir Computer that is suitable for hardware implementations.

III. Implement the design and test its performance.

IV. Design an efficient hardware implementation of the Reservoir computer design.

V. Study Chaotic systems and construct Chua's chaotic circuit.

VI. Explore the possibility of using the FPGA RC for chaotic communication.

2. Introduction

2.1. Overview

This chapter introduces the concepts of Artificial Neural Networks, Chaotic systems, and their respective research fields. In particular, I will focus on Reservoir Computers as a standout branch of Recurrent Neural Networks which are an Artificial Neural Network, for their ability to deal with temporal signals inherited from RNNs and its easier training procedure. And inside the field of Chaotic systems and the study of chaos, I will present the concept of synchronization and how it can be used for communication. Lastly, I hope to then tie these two topics together with an attempt at a functional and easy-to-implement hardware Reservoir Computer to be used in a chaotic communication channel. The center of attention and therefore research in this paper will be Reservoir Computers, specifically their possible implementations in hardware and their use in real-time and real-world engineering tasks such as performing time-series forecasting.

2.2. Artificial Neural Networks

Artificial Neural Networks (ANNs) are a subset of machine learning that are inspired in their operation by the biological processes that happen in our brain when processing information. The motivation for developing such networks is their ability to solve problems that are difficult to solve deterministically with sequential algorithms and their "intelligence" stemming from how they adapt to solve the required task.

Consider tasks such as speech recognition or image recognition, these were difficult to solve by past machine learning algorithms, but Neural Networks can be built with relative ease to classify the data efficiently and accurately at high speeds.

Artificial Neural Networks (ANNs) are made of nodes and separated into layers, them being: an input layer, one or more hidden layers, and an output layer. Each node is an artificial neuron and has an associated weight and threshold for its activation.

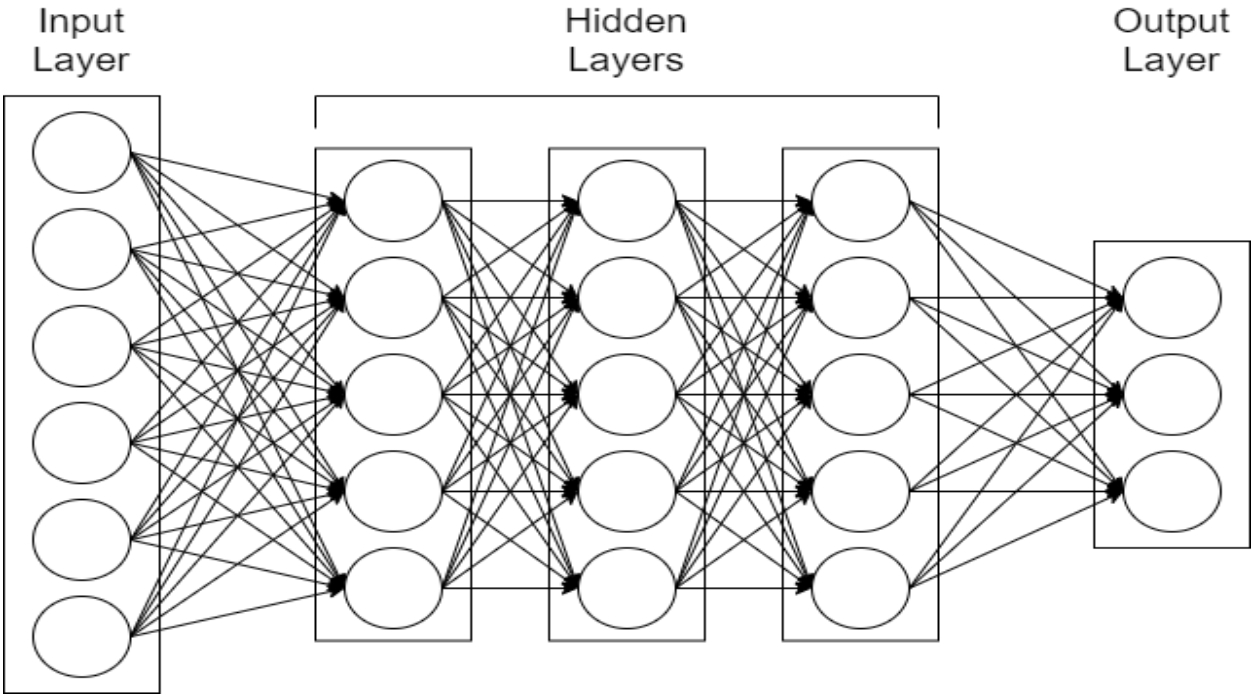


Fig 1. Artificial Neural Network diagram.

Neural Networks have the capability to learn a desired function by using data pairs which makes them a supervised learning algorithm. Training happens when a training data set is fed to the network and the weights of the neurons are dynamically changed to achieve the wanted output.

The field of research on Artificial Neural Networks does not only encompass computer science but is widely interdisciplinary and is showing prominent growth and potential in fields such as Artificial Intelligence, signal processing, modeling, and simulation. This growth sparks the need for eventual hardware designs and

implementations for the use of these Neural Networks in real-world engineering tasks.

There exist multiple archetypes of these Neural Networks with: Feed Forward Neural Networks (FFNNs) and Recurrent Neural Networks (RNNs) being some of the more prominent ones. Feed Forward Neural Networks have their neurons organized in layers and don't have any backward or lateral connections between the hidden layers as shown above in Fig 1.

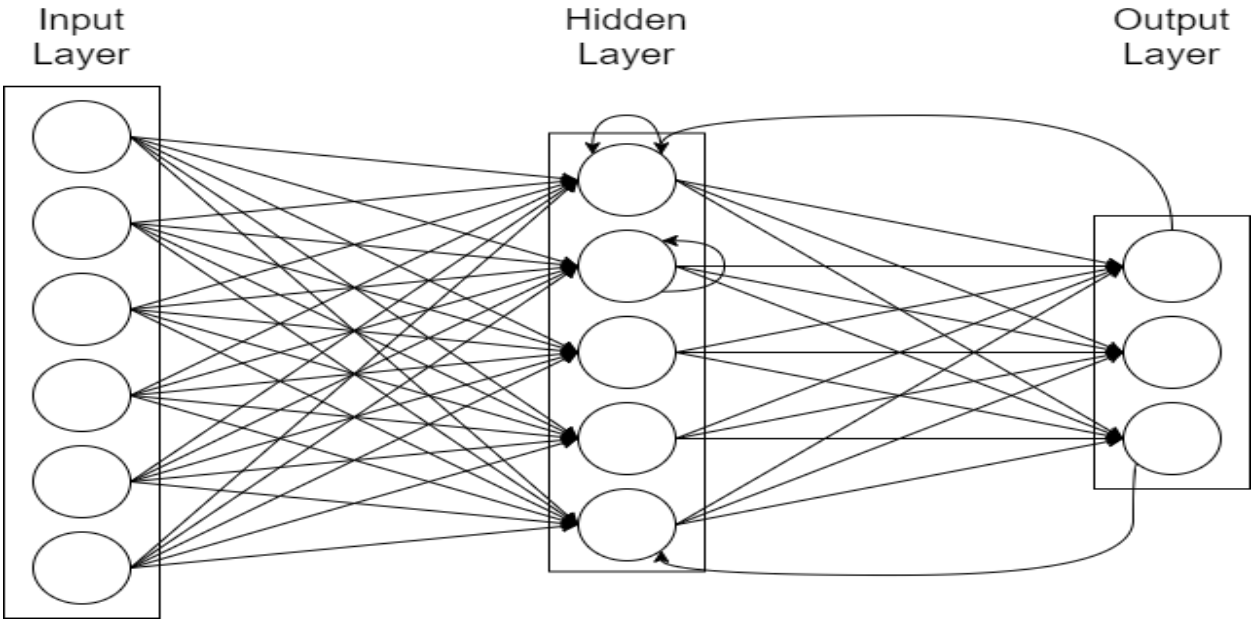


Fig 2. Recurrent Neural Network with one hidden layer diagram.

On the other hand, Recurrent Neural networks allow for feedback connections between neurons allowing the information to propagate both ways as shown in Fig 2.

This allows the network to have a "memory" as the state of a neuron do not only depend on the current value of the input signal but also the past states of the network.

Therefore, FFNNs are a static reactive function to the input signal and RNNs are a dynamic system with an evolving state. The major advantage then of RNNs over FFNNs is that they can learn temporal tasks as the nonlinear transformations of the input signal that happen inside the system capture useful contextual information when processing the signal.

Nevertheless, RNNs present a big limitation that is a result of the complexity of training them and the time it requires. Both RNN and FFNN perform training by using gradient descent. Gradient descent is a method where the weights of neurons in a network are changed according to the output error gradients so that output training error is minimized.

2.2.1. Reservoir Computing

Reservoir Computers (RC) is a recent strategy for implementing Recurrent Neural Networks (RNNs) that is characterized by an easier training procedure that can be performed via a simple linear regression.

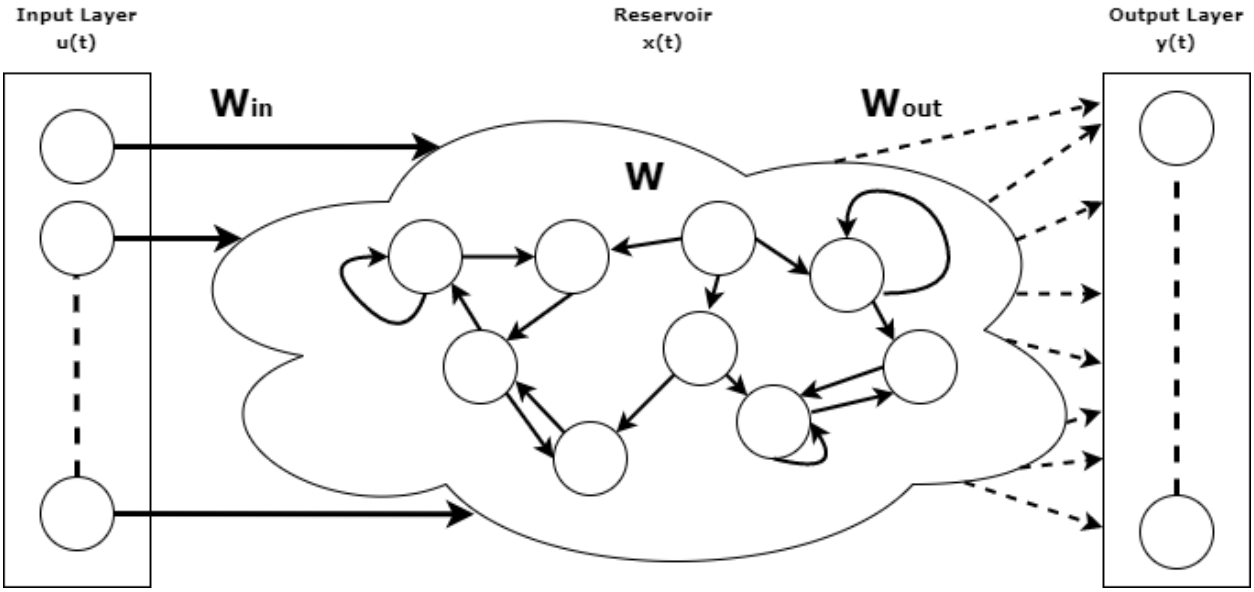


Fig 3. Reservoir Computer diagram.

The RC architecture consists of three parts: an input layer, the reservoir, and the output layer. The input layer feeds the reservoir with input signals: $\boldsymbol{u(t)} = (\boldsymbol{u_1(t)}, \dots, \boldsymbol{u_k(t)})$, the reservoir consists of several randomly interconnected neurons $(\boldsymbol{N})$ with states: $\boldsymbol{x(t)} = (\boldsymbol{x_1(t)}, \dots, \boldsymbol{x_N(t)})$ and internal weights $(\boldsymbol{W})$ and the output layer consists of a linear weighted sum of the node states. An important difference is that the connections are kept fixed between the nodes during the systems training and operation (Fig 3).

Mathematically the objective of the system is to perform some transformation on the input and therefore learn the function $\boldsymbol{y(t)} = \boldsymbol{y(u(t))}$ for a non-temporal task and $\boldsymbol{y(t)} = \boldsymbol{y(u(t), u(t-1), u(t-2), \dots)}$ for a temporal task, which minimizes the error between $\boldsymbol{y(t)}$ and $\boldsymbol{y_{target}(t)}$. There exist a variety of employed error equations, but the one I will use and focus on is the mean-squared-error (MSE, (1)).

$$MSE(\boldsymbol{y}, \boldsymbol{y_{target}}) = \frac{1}{\boldsymbol{n}} \sum_{i=1}^{n} (\boldsymbol{y} - \boldsymbol{y_{target}})^2 \qquad (\ 1\ )$$

This means that the system is mapping a low dimensional input $\boldsymbol{u(t)}$ into a high dimensional space (matrix of neuron states $\boldsymbol{X}$). To obtain the approximation of $\boldsymbol{y_{target}(t)}$ a linear weighted sum of the neuron states and their respective weights is used (2).

$$\boldsymbol{y(t)} = \sum_{i=1}^{N} \boldsymbol{W}_i \times \boldsymbol{x}_i(\boldsymbol{t}) \quad or \quad \boldsymbol{y(t)} = \boldsymbol{W_{out}x(t)} \qquad (2)$$

Where $\boldsymbol{x}_i(\boldsymbol{t})$ denotes the state of i^{th} node at time $\boldsymbol{t}$,and $\boldsymbol{W_{out}}$ is the matrix of trained weights assigned to the neurons. The training process then involves minimizing the error by adjusting the $\boldsymbol{W_{out}}$ matrix. Generally, the states $\boldsymbol{x}$ are initialized to 0 and then a training data set is fed to the system, an initial washout period is discarded for any transient errors and the reservoir states $\boldsymbol{x(t)}$ are collected and stored in a large matrix ($\boldsymbol{X}$,(3) of dimension $\boldsymbol{N} \times \boldsymbol{T_{train}}$, where $\boldsymbol{T_{train}}$ is the number of time steps in the training data set.

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1(1) & \cdots & \boldsymbol{x}_1(\boldsymbol{T_{train}}) \\ \vdots & \ddots & \vdots \\ \boldsymbol{x_N}(1) & \cdots & \boldsymbol{x_N}(\boldsymbol{T_{train}}) \end{pmatrix} \qquad (3)$$

Similarly, the desired outputs $\boldsymbol{y_{target}(t)}$ are also collected and stored in a matrix ($\boldsymbol{Y}_{target}$,(4)) of dimension $\boldsymbol{K} \times \boldsymbol{T_{train}}$, where $\boldsymbol{K}$ is the number of desired output signals at each data point.

$$\boldsymbol{Y}_{target} = \begin{pmatrix} \boldsymbol{y_{target,1}}(\boldsymbol{1}) & \cdots & \boldsymbol{y_{target,1}}(\boldsymbol{T_{train}}) \\ \vdots & \ddots & \vdots \\ \boldsymbol{y_{target,K}}(\boldsymbol{1}) & \cdots & \boldsymbol{y_{target,K}}(\boldsymbol{T_{train}}) \end{pmatrix} \qquad (4)$$

The training in terms of matrices $\boldsymbol{W_{out}}$, $\boldsymbol{Y}_{target}$ and $\boldsymbol{X}$ corresponds to a least squares regression (5) of the target outputs $\boldsymbol{Y}_{target}$ onto the predictors in $\boldsymbol{X}$.

$$\boldsymbol{W_{out}} = (\boldsymbol{X^T X})^{-1} \boldsymbol{X^T Y}_{target} \qquad (5)$$

Once the training data set is exhausted the system is deemed to be trained and the $\boldsymbol{W_{out}}$ matrix stops updating. Usually, the system is then fed a validation set before testing and benchmarking, this validation set is used to select the optimal values for some hyperparameters of the network such as: input weighting ($\boldsymbol{W_{in}}$), spectral radius and sparseness degree. Finally, the system is evaluated on a testing set and the performance is calculated.

However, the simplicity of the training procedure in RC doesn't include the hyperparameters which are difficult to optimize and assess their performance. Multiple strategies exist to optimize them, and it is a hot topic for research in RC and Echo State Networks (ESNs).

2.3. Chaotic Systems

Chaotic systems are systems that are highly sensitive to initial conditions, where the uncertainty of a forecast increases exponentially with elapse time. No general

agreement exists on a mathematical definition of chaos. However, a simple and widely used definition of chaotic systems is that a chaotic system will [1]:

1. *Be sensitive to initial conditions.*
2. *Be topologically transitive (for any two open sets, some points from one set will eventually hit the other set).*
3. *Have dense periodic orbits.*

Then a widely used test to determine if a deterministic dynamical system is chaotic is to calculate the Lyapunov exponents of the system. If the largest Lyapunov exponent is positive it indicates chaos: if $\lambda > 0$, then the nearby trajectories separate exponentially and if $\lambda < 0$, then nearby trajectories stay close to each other [2].

2.3.1. Chua Circuit

Chua's circuit is the simplest well documented electronic circuit that exhibits chaotic behavior.

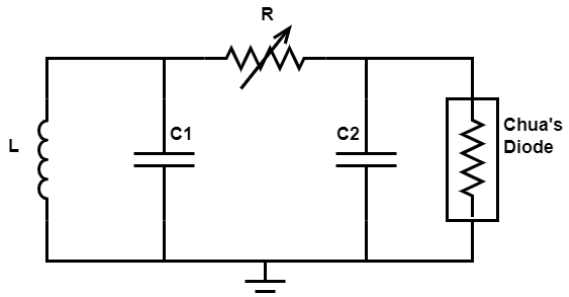


Fig 4. Chua's circuit diagram.

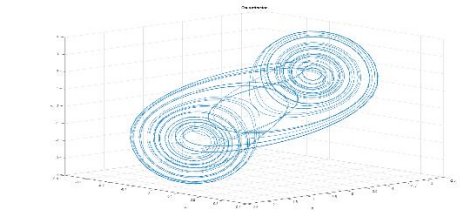
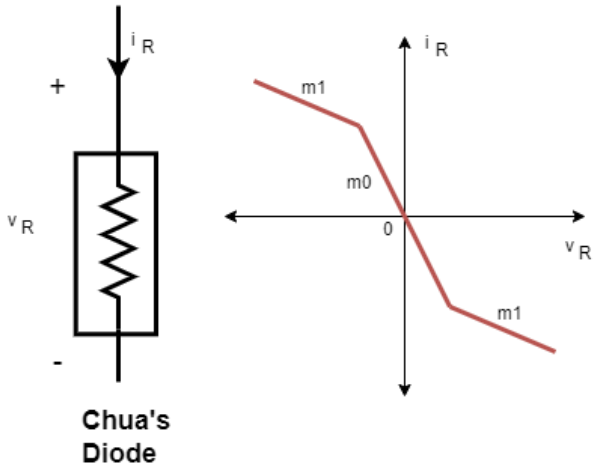


Fig 5. Chua's chaotic attractor.

The diagram of the Chua circuit is shown in Fig 4. It is made up of 5 circuit elements, the four on the left are standard off-the-shelf linear passive electrical components, namely an Inductor (*L*), a resistor (*R*) and two capacitors (*C1* and *C2*); and the one on the right is Chua's diode, a nonlinear element that is characterized by a nonlinear voltage function $i_R = g(v_R)$. Where i_R is the current through the diode and v_R the voltage across the diode. The function $g(\cdot)$ can have

many shapes, but the original [3] and the one we will focus on is a 3-segment piecewise-linear characteristic shown below in Fig 6.



$$g(v_R) = \begin{cases} v_R m_1 + m_1 - m_0 & , v_R \leq -1 \\ v_R m_0 & , -1 \leq v_R \leq 1 \\ v_R m_1 + m_0 - m_1 & , 1 \leq v_R \end{cases}$$

Fig 6. Current-Voltage characteristic of Chua's Diode.

Where m_1 is the slope of the upper and lower sections and m_0 is the slope of the middle part. The dynamics of a dimensionless Chua system are given by:

$$\begin{aligned} \dot{x} &= \alpha(y - h(x)) \\ \dot{y} &= x - y + z \\ \dot{z} &= -\beta y \end{aligned} \qquad (6)$$

We obtain these equations by rescaling the circuit variables v_{C_1}, v_{C_2} and i_L from Fig 4. The Chua equations consist of 3 dimensionless state variables (x, y and z), 2 dimensionless parameters α and β which are real numbers and a scalar function of x which is defined as a piecewise-linear function $h(x)$. [4]

$$h(x) \triangleq x + g(x) = m_1 x + \frac{1}{2}(m_0 - m_1)[|x+1| - |x-1|] \qquad (7)$$

If we are to plot x , y and z or v_{C_1}, v_{C_2} and i_L in case of a physical circuit we can obtain Chua's chaotic attractor also called the Double Scroll (Fig 5).

2.3.2. Chaotic Synchronization

Chaotic systems would appear to be inherently unsynchronous, however they have been observed to synchronize.

Consider two identical *d*-dimensional chaotic systems a and b described by:

$$x_t^a = F(x_{t-1}^a) \qquad (8)$$

$$x_t^b = F\left(x_{t-1}^b\right) \quad (9)$$

Which share a function F. If the initial conditions are different even by a slight margin the system will have exponential divergence due to its chaotic nature. They will still have the same attractor, but their motion will be uncorrelated over time.

In this example, synchronization can be achieved by a coupling between the two systems such that the trajectories of x_t^a and x_t^b become asymptotical with time ($x_t^a \approx x_t^b$, $\left\|x_t^a - x_t^b\right\| \to 0 \; as \; t \to \infty$).

$$x_t^a = F(x_{t-1}^a) + c^a(x_{t-1}^a - x_{t-1}^b) \quad (10)$$

$$x_t^b = F\left(x_{t-1}^b\right) + c^b(x_{t-1}^b - x_{t-1}^a) \quad (11)$$

Where c^a and c^b are the coupling constants. If $c^a \neq 0$ and $c^b \neq 0$, we say that there is a two-way coupling. If c^a is null and $c^b \neq 0$, we say that there is one way coupling from a to b, since b is influenced by a but a isn't influenced by b [5].

The systems in equations (10) and (11) coupled essentially form a 2*d*-dimensional dynamical system. If synchronization is achieved, $x_t^a = x_t^b$ the synchronized state of the system represents a *d*-dimensional invariant manifold where equations (10) and (11) reduce to equations(8) and (9).

2.3.3. Chaotic Communication

Certain chaotic systems possess a self-synchronization property [6]. A chaotic system is therefore self-synchronizing if it can be separated into subsystems: a drive system and a stable response subsystem that synchronize when linked with a common drive signal [7].

For example, the Lorenz system can be decomposed into two separate response subsystems that will each synchronize to the drive system when started with any initial condition. The Lorenz system is given by:

$$\begin{cases} \dot{x} &= \sigma(y - x) \\ \dot{y} &= \rho x - y - xz \\ \dot{z} &= xy - \beta z \end{cases} \quad (12)$$

Where σ, ρ and β are parameters. The Lorenz system can then be decomposed into two stable subsystems.

A stable (x_1, z_1) response subsystem:

$$\begin{aligned} \dot{x_1} &= \sigma(y - x_1) \\ \dot{z_1} &= x_1 y - \beta z_1 \end{aligned} \qquad (13)$$

And a second stable (y_2, z_2) response subsystem:

$$\begin{aligned} \dot{y_1} &= \rho x - y_2 - x z_2 \\ \dot{z_2} &= x y_2 - \beta z_2 \end{aligned} \qquad (14)$$

Equation (12) can be considered as the drive system since its dynamics are independent of the response subsystems. While equation (13) and equation (14) represent dynamical response systems which are driven by the drive signals $y(t)$ and $x(t)$ respectively.

This decomposition allows a communication channel between the transmitter which is the Lorenz system (12) and the receiver which are the two subsystems (13) and (14).

If we define the dynamical errors as $e = t - r$, where the transmitter variables are $t = (x, y, z)$ and the receiver variables are $r = (x_r, y_r, z_r)$. The synchronization in the Lorenz system is a result of a stable error dynamics between the transmitter and receiver [8].

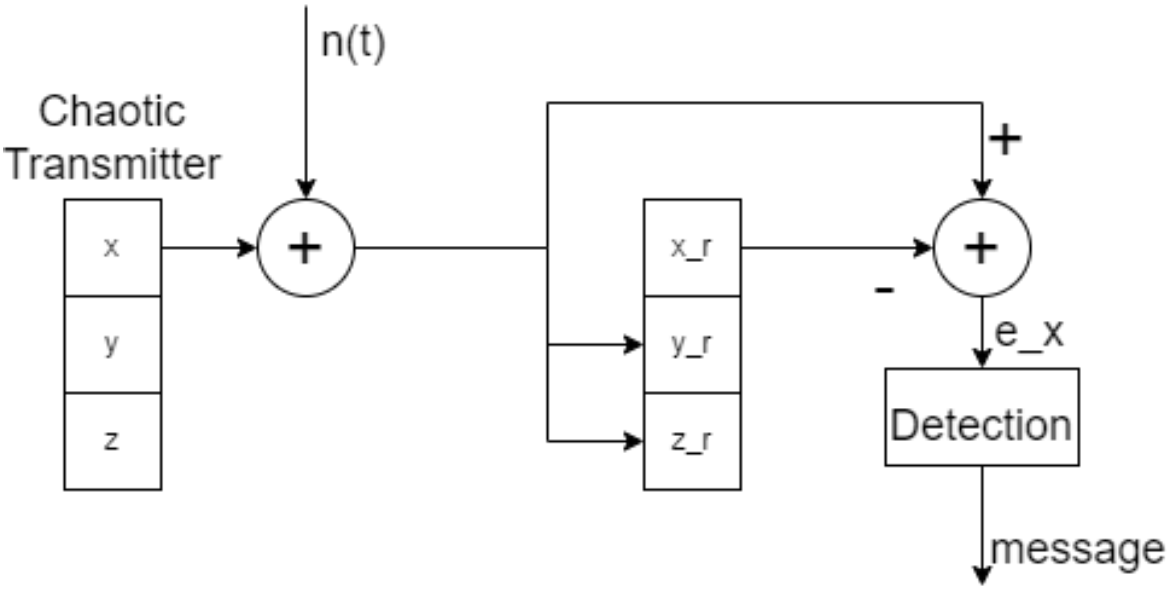


Fig 7. Chaotic Communication channel using Lorenz system diagram.

In this setup we can effectively communicate by applying an information signal $n(t)$ to the x signal in a Lorenz system and then subtract the corresponding state of x_r, essentially removing the chaos from the information signal. This leaves us with the error $e_x = x - x_r$, which in case of $n(t)$ being a binary bitsream would mean that if

we pass e_x through a lowpass filter we should recover the binary bitstream signal and thus the information communicated.

2.4. Field-Programmable-Gate-Arrays

A Field-Programmable-Gate-Array or FPGA for short is an integrated circuit that can be programmed by a user after it has been manufactured. Modern FPGAs avail of large amounts of configurable logic blocks connected via programmable interconnects, these blocks create an array of logic gates that allow the FPGA to be reprogrammable and customized to perform specific computations. Due to their programmable nature, FPGAs are a common resource used in a wide variety of fields and is usually used as a resource for some hardware acceleration to offload some repetitive and computationally intensive task.

FPGA programming therefore consists of specifying in a hardware description language (HDL) a hardware architecture that will execute the wanted algorithm.

3. Implementation

3.1. Reservoir Structure

Like any Artificial Neural Network, Reservoir Computers benefit greatly from the inherent parallelism in hardware implementations. Furthermore, Reservoir Computing with its simpler training procedure is even better suited for hardware implementations such as on a FPGA. Implementing a Reservoir Computer on a FPGA would allow to accelerate the network and have a real-world network that can process and output in real-time.

3.1.1. Limitations and Architectures

However, the conventional random structure of the reservoir in a Reservoir Computer isn't a good choice for a hardware implementation due to its geometry.

Thankfully a variety of reservoir topologies and architectures exist for Reservoir Computing, such as: Single Dynamic Node Reservoir, Time Delay Node Reservoir, Liquid State Reservoir, Cyclic Node Reservoir, etc.

The chosen architecture for the reservoir in this paper is the Cyclic Reservoir.

3.1.2. Cyclic Reservoir Structure

The neurons in a cyclic reservoir are arranged in a ring structure, with every neuron having a connection to both the neuron before and after it in the chain.

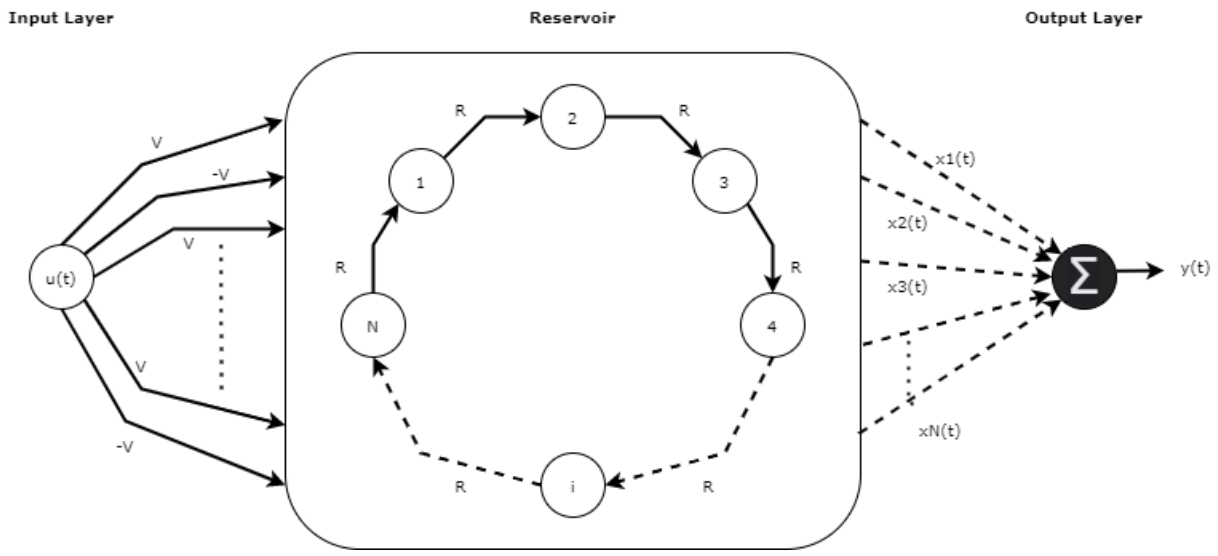


Fig 8. Cyclic Node Reservoir architecture.

In this design the node connections are weighted by the weight *r* and the input is fully connected to the reservoir and has a connection weight *v*. Parameters *r* and *v* must be scanned to find the optimum weight configuration. This architecture has been observed to only have a slightly worse performance than the classical topology [9]. The main advantage of such a reservoir in hardware implementations is its scaling simplicity as the number of connections within the reservoir is independent of the number of neurons.

3.2. Node Structure

3.2.1. Dual Input Node

The node structure chosen was the dual-input node due to the Cyclic Reservoir specifying that all neurons have a connection to the neuron before and after it and that the input layer is fully connected to the reservoir. This means that one input is from the input layer and one input is the output from the node before in the ring. In our case we are aiming to do chaotic time series prediction and will only use the chaotic time series as input, so our input layer is just one element, therefore the nodes end up with only two inputs.

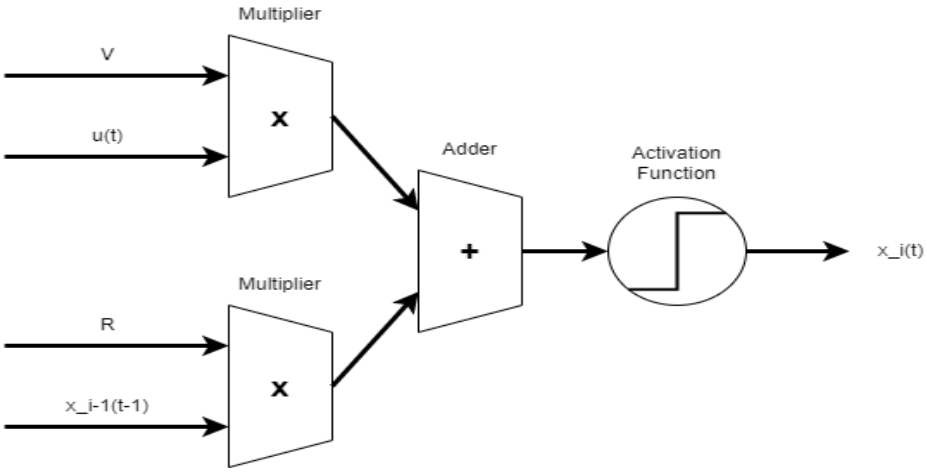


Fig 9. Dual-input node structure.

As shown above in Fig 9, the systems input $\boldsymbol{u}(t)$ is influenced by the parameter *v* and the previous node's output is influenced by the parameter *r*. The weighted inputs are then combined and passed through an activation function to output the node's state.

3.2.2. Activation Function

The transfer function of an analog discrete-time neuron is typically nonlinear in nature, and grounds the output to some value (usually between -1 and 1). The most common activation function is sigmoidal in shape.

3.2.3. Stochastic Computing

Even with the simpler Cyclic Reservoir topology, the multiplications that happen inside the nodes are very resource heavy for implementations on more limited hardware targets such as FPGAs. This is where a feasible alternative to implement complex computations is applied. This alternative is Stochastic Computing.

Stochastic computing is based on converting binary values inside the FPGA into probabilistic bit streams that can then be used in simplified methods of computation and then be reconverted into the binary values [10]. This approach is generally able to reduce the circuit area of computations compared to classical deterministic approaches [11].

Stochastic Computing's main drawback is long computation time, which usually grows exponentially with respect to precision, meaning that this method is not able to speed up the operation of an Artificial Neuronal Network compared to the conventional approaches. However, due to its substantial reduction in necessary hardware resources and higher error tolerance it is still an approach worth considering in resource limited or high environmental noise environments.

Stochastic Computing's higher error tolerance stems from stochastic circuitry ability to tolerate environmental errors that seriously impact conventional circuitry. A single bit-flip in a binary circuit can cause a very large error especially if it occurs in a high-significance bit, whereas even flipping a few bits in a long bitstream will have very little effect on the value in stochastic circuitry.

4. Results

4.1. Software Implementation

MATLAB was used to first build, train and test the chosen cyclic reservoir architecture for the Reservoir Computer. A Chua's circuit with chaotic behavior was also simulated in MATLAB using the same specifications as the breadboard circuit that has been built.

4.1.1. Cyclic Node Reservoir in MATLAB

The cyclic reservoir was constructed by arranging nodes in a ring structure as detailed in Fig 8. The nodes also followed the dual-input structure shown in Fig 9 and the activation function used was the tanh function which is commonly used in RC nodes.

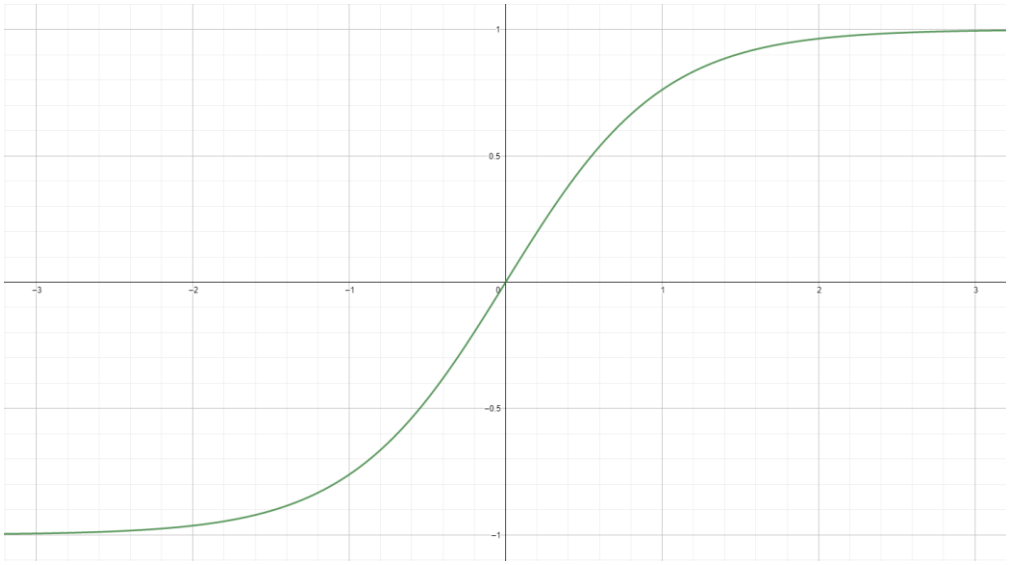


Fig 10. Tanh function.

To train the output weights of the nodes least squares regression (5) was used, specifically a MATLAB matrix approach [12].

```
function W_out = least_squares_regression(X, Y_target)
    K = (X.'*X)^-1;
    W_out = K * X.'*Y_target;

end
```

Fig 11. Least-squares-regression function code (MATLAB).

Where `W_out` is the resulting vector of node output weights, `X` is the matrix of node states (3) and `Y_target` is the vector of the wanted system outputs (4).

With this design I have two configuration parameters (*r* and *v*) which need to be optimized for the RC to preform optimally. Both parameters optimal values need to be found experimentally, therefore I scanned the reservoirs performance (Mean-squared-error of its forecast) for the range of values that *r* and *v* can take (0 to 1). To do this I simply needed to do a *for* loop inside of another *for* loop as shown below.

```
for y = 1:config_res
    for x = 1:config_res
          . . .
          . . .
    end
end
```

Fig 12. Parameter scan loop.

Where `config_res` is a value that dictates the resolution of my scan, parameter *r* is `x/config_res` inside the loop and similarly parameter *v* is `y/config_res`. Therefore, if `config_res` is 10 then the values that will be scanned for *r* and *v* will be from 0.1 to 1 in increments of 0.1.

The system is highly sensitive to the configuration parameters and small changes in them can severely alter the performance of the system, therefore being able to adjust the resolution of the scan is useful as some optimal values might land in-between points in a scan with low resolution like 10. The performance of an optimally configurated cyclic RC will be shown in section 4.3.2.

4.1.2. Chua simulation in MATLAB

While in most academic literature a dimensionless model is usually preferred, it is still useful to have a more realistic model. As this project goal is a real-world capable implementation it is even more justified to use a model that is as close to a real Chua circuit as possible.

Using the equations detailed in section 2.3.1, a MATLAB script was constructed that uses values for capacitance of *C1* and *C2*, the resistance of *R*, inductance of *L* and the slopes m_0 and m_1 of the Chua's diode. Furthermore, another MATLAB script was constructed that replaces Chua's diode with off-the-shelf electronic components for greater resemblance to what an actual circuit would behave like.

4.2. Hardware Implementation

The FPGA hardware implementation of the RC was designed and written in Verilog and compiled in Vivado. The Reservoir Computer was constructed as per the previous specifications in section 3 of a simple Cyclic Reservoir, 2-input Nodes and stochastic circuitry for computation. The training for the RC was decided to be offloaded to a computer as the training procedure is not well suited for such an implementation and would eliminate the possibility of having a Low-cost FPGA as the target of the implementation.

Therefore, the aimed operational procedure for this design would be to act as the reservoir only, as opposed to the whole system, meaning that the design's collected output would be a matrix of node states ($\boldsymbol{X}$,(3)) which then could be used to determine the output weights ($\boldsymbol{W_{out}}$,(5)) and obtain the systems final output ($\boldsymbol{y(t)}$,(2)) on the host PC. The design would be able to operate semi autonomously if the board or setup avails of some digital storage like an SD card where it can collect the node outputs for later use outside it or work in conjunction with a computer which would collect the node outputs and calculate the output in parallel to the FPGAs operation. The proposed design has a 16bit resolution for the node states and an 8-bit resolution for the weights.

4.2.1. Stochastic Circuitry

Using Stochastic Computation, we will be converting binary values inside the FPGA to probabilistic pulse streams when simplified computation is needed and then reconverting back. This will be done via two modules, namely Binary2Stochastic (B2P) which as the name implies is the circuit that will transform binary values into the stochastic pulse streams and Pulse2Binary (P2B) which will do the opposite conversion. These conversions happen with a period of $T_{EVAL} = N_c \times CLK$ therefore, the system as a whole follows this other clock, specifically the system is sampling the input every T_{EVAL} and also outputs every T_{EVAL}. The constant N_c dictates the evaluation time of the stochastic circuitry and therefore the system and is adjusted as necessary. However, the smaller the value of N_c the lower accuracy the system will have as the stochastic pulse streams have less resolution. Below is an example of how multiplication is done with stochastic pulse streams and the effect of T_{EVAL}.

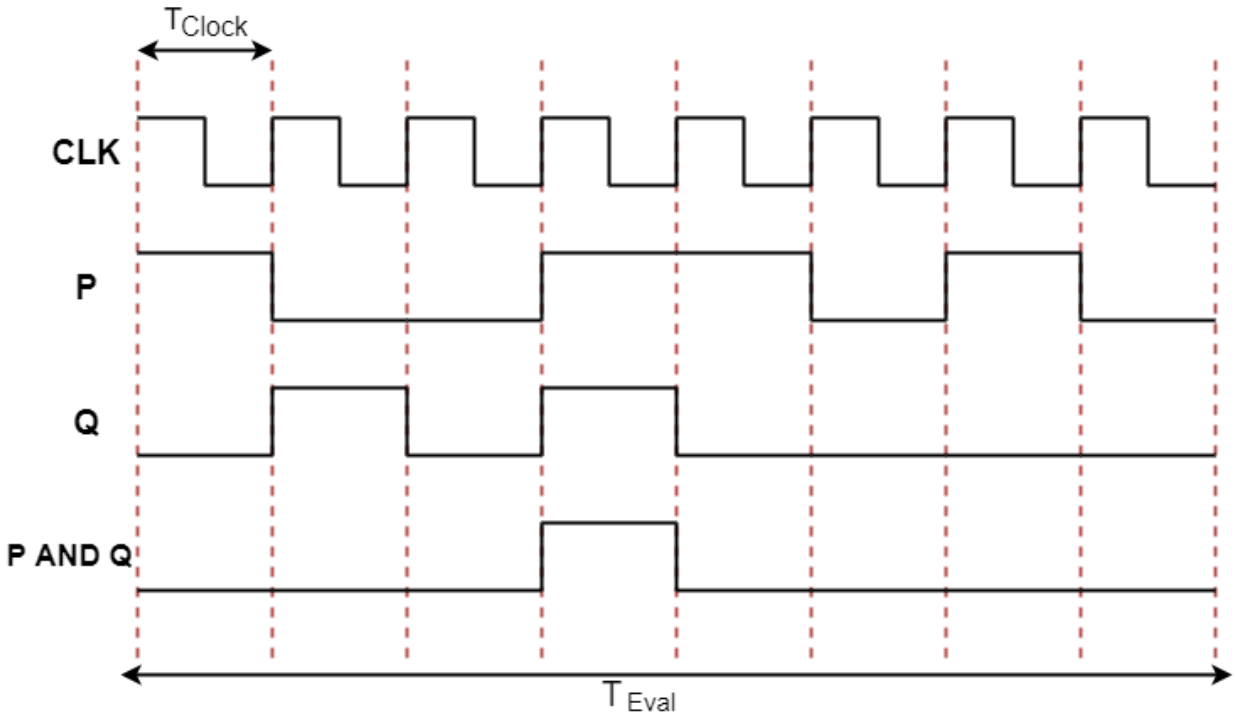


Fig 13. Stochastic pulse stream multiplication.

In Fig 13 we are using a single AND gate to multiply two numbers represented by two stochastic pulse streams p and q over $T_{EVAL} = 8 \times T_{CLK}$.

$$\begin{matrix} Unipolar\ coding \\ p \rightarrow [0,1] \\ Bipolar\ coding \\ p^* \rightarrow [-1,1] \end{matrix} \rightarrow p^* = 2p - 1 \qquad \begin{matrix} p = \frac{Number\ of\ high\ values}{Number\ of\ cycles} = \frac{4}{8} = \frac{1}{2} \\ q = \frac{2}{8} = \frac{1}{4} \\ p\ AND\ q = p \times q = \frac{1}{4} \times \frac{1}{2} = \frac{1}{8} \end{matrix} \quad (15)$$

As shown the multiplication is greatly reduced and only consumes one AND gate no matter how many bits are in a value with unipolar coding and an XNOR gate for Bipolar coding, this is a great improvement in resource usage and circuit area when compared to conventional bit by bit multiplication.

Binary to Stochastic conversion makes use of pseudorandom number generators which in our design will be a 16bit LFSR. The B2P module is really simple, it consists of a comparator circuit that compares the incoming binary value P to a pseudorandom number from the LFSR and outputs a high if the binary value is greater and 0 if its lesser.

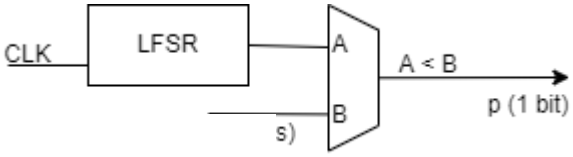


Fig 14. B2P module.

The output is a probabilistic stream of 1s and 0s that over the T_{EVAL} represents the necessary binary value.

Pulse to Binary conversion makes use of two counter, a register and a comparator circuits. The P2B module uses one counter to count the number of high values in the incoming stochastic pulse stream p and a second counter that counts up every clock cycle, which has a comparator attached to its count that outputs a high when the clock counter reaches N_c. This comparator's output then triggers a register that saves the first counters count to output its value and resets the counters. Essentially, we are counting how many 1s are in p every N_c and outputting that value.

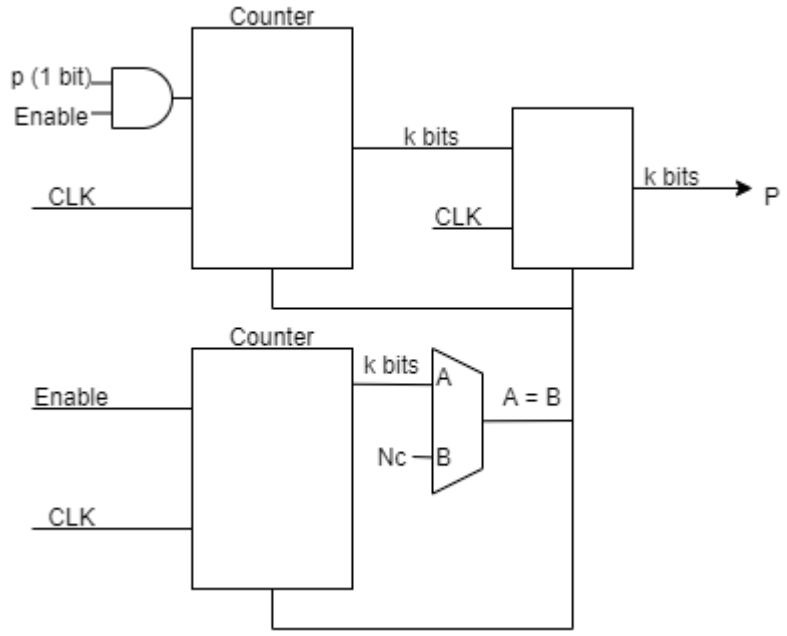


Fig 15. P2B module.

Something to consider is that a probabilistic error is always present during P2B conversion. The probability of obtaining an output equal to X is given by the binomial distribution below in (16).

$$P(X) = \binom{N_c}{X} p^X (1-p)^{N_c - X} \qquad (16)$$

The mean value of X is the expected exact conversion ($\overline{X} = pN_c$), and the standard derivation is $\sigma = \sqrt{N_c p(1-p)}$. This error is inversely proportionate to N_c as increasing the evaluation time T_{EVAL} will decrease it, however this is at the cost of the processing speed of the system [13] [14] [15].

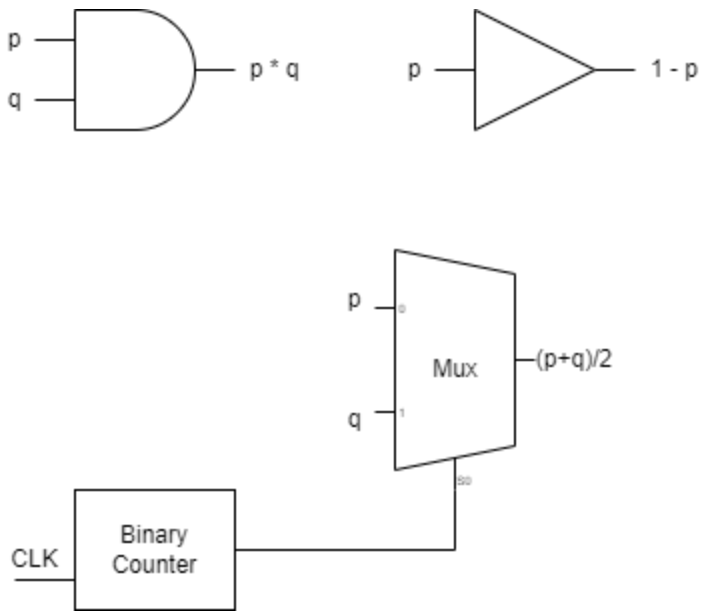


Fig 16. Arithmetic examples with stochastic pulse streams.

After converting the binary values into probabilistic pulse streams, we can easily perform arithmetic using basic logic gates and multiplexers as demonstrated in Fig 16. As shown earlier in Fig 13 multiplication can be done by an AND gate for unipolar coding and XNOR gate for bipolar coding, summation can be done with multiplexers that switch on a binary clock and negation can be done through NOT gates.

4.2.2. Reservoir and Node circuitry

The chosen dual-input node design from Fig 9 was modified to decrease the computational hardware requirements by using Stochastic Computations. Specifically, the incoming inputs to the node would be converted into stochastic pulse streams for their multiplication by their respective configuration weights and for their summation afterwards. Then the stochastic pulse streams would be reconverted to a binary value for the activation function and output.

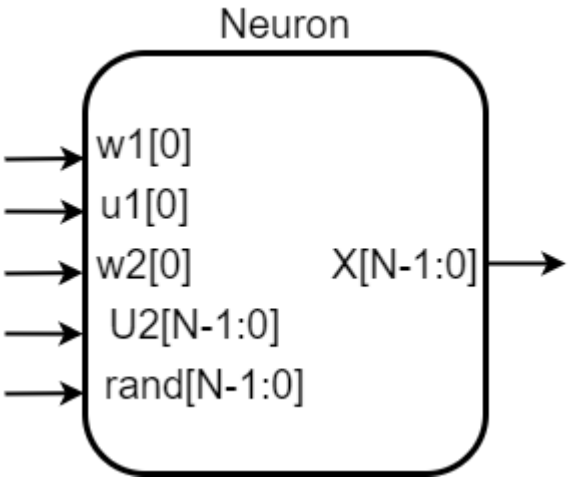


Fig 17. High level hardware neuron module.

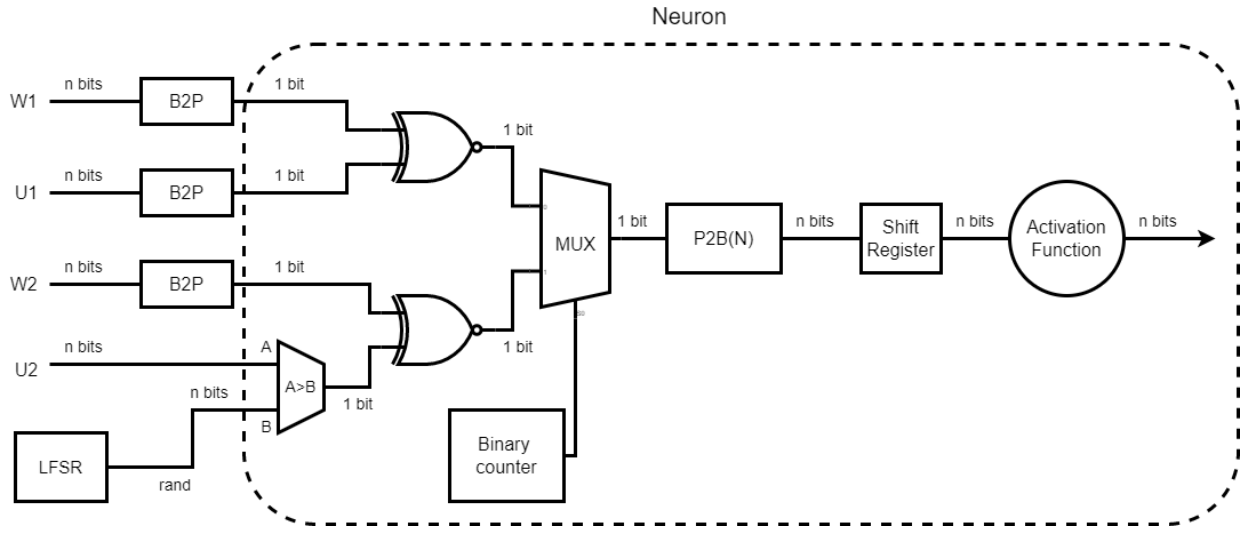


Fig 18. Low level hardware neuron module.

As shown in Fig 18 the neuron module has 2 changing *n*-bit (16-bit in our case) inputs *U1* and *U2*, which are the systems input $\boldsymbol{u}(\boldsymbol{t})$ and the previous neurons output $\boldsymbol{x}_{n-1}(t)$ respectively. Furthermore, the module has 2 static n-bit inputs, inputs *W1* and *W2* which are the configuration parameters *r* and *v*. The last input to the module is *rand* which is a pseudorandom number generated by a LFSR module outside.

The conversion from binary to stochastic pulse streams is performed outside the neuron for *U1*, *W1* and *W2*. Input *U2* is the output from another neuron and it's a direct wire between neurons so it's more convenient to convert it to a stochastic pulse stream inside the neuron using *rand* and a comparator, essentially a B2P module nestled inside the neuron. Once converted the 16-bit *U1* becomes a 1-bit stream *u1*, similarly *W1* and *W2* convert to 1-bit streams *w1* and *w2* (Fig 17). Inside the neuron multiplication is performed with two XNOR gates for bipolar coding and then fed into a multiplexer working on a binary clock.

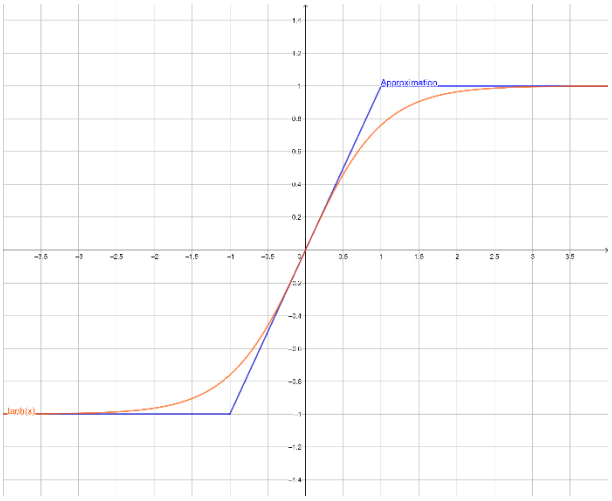
$$B2P(W1[15:0]) = w1[0] \quad B2P(W2[15:0]) = w2[0] \quad B2P(U1[15:0]) = u1[0] \quad B2P(U2[15:0]) = u2[0]$$

$$u1\ XNOR\ w1\ =\ B2P(U1[15:0] \times W1[15:0]) \quad u2\ XNOR\ w2\ =\ B2P(U2[15:0] \times W2[15:0])$$

$$MUX_{CLK}(u1\ XNOR\ w1\ , u2\ XNOR\ w2) = \frac{u1\ XNOR\ w1 + u2\ XNOR\ w2}{2} \qquad (17)$$

After the multiplexer we reconvert $\frac{u1\ XNOR\ w1+u2\ XNOR\ w2}{2}$ back to a binary value through the use of a P2B module, the obtained binary value will be $\frac{U1\times W1+U2\times W2}{2}$. Then we make use of a shift register that shifts the binary word one position to the left which has the effect of multiplying the binary value by two, meaning that the input to the activation function ends up being $U1 \times W1 + U2 \times W2$.

As for the sigmoidal activation function of the neuron, a 3-piecewise linear approximation of tanh was used shown below in Fig 19.



$$f(x) = \begin{cases} -1, & x < -1 \\ x, & -1 \leq x < 1 \\ 1, & x \geq 1 \end{cases}$$

Fig 19. 3-piece Tanh approximation.

The LFSR module can be shared by all neurons as the neurons only communicate with binary values instead of the stochastic pulse streams. Sharing a common LFSR module allows reducing the hardware resources for every neuron significantly. Furthermore, the B2P modules for the systems input $\boldsymbol{u(t)}$ and the two configuration parameters *r* and *v* can also be shared by all neurons since those values are the same for all the neurons. This more efficient structure that shares the pseudorandom number generators and some B2P modules can be seen below in Fig 20.

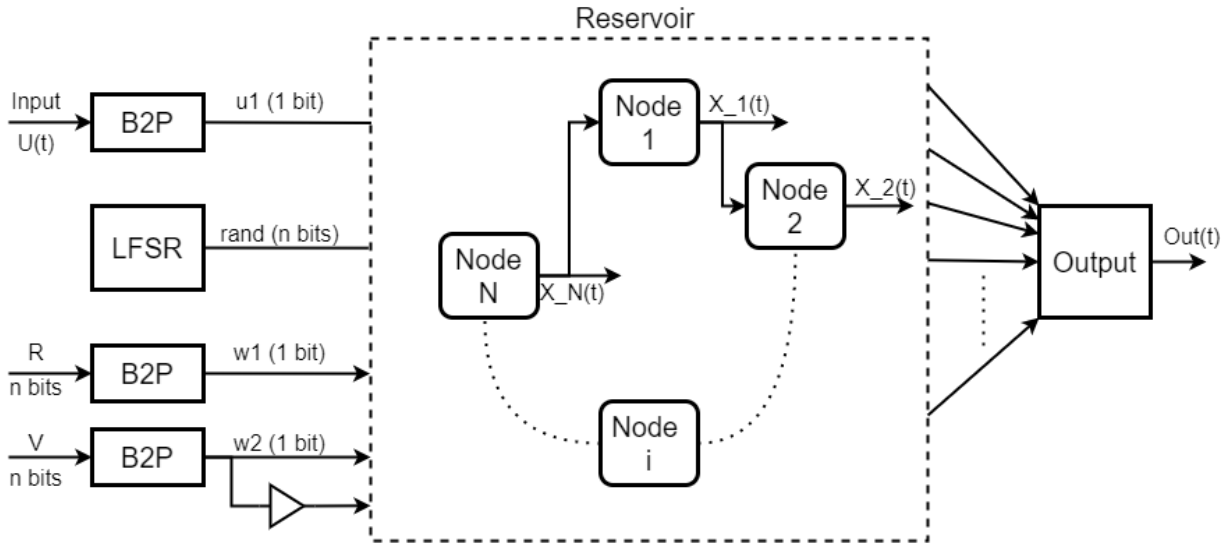


Fig 20. Hardware RC diagram sharing modules.

As shown in Fig 8 the reservoir structure in Fig 20 also assigns the *v parameter that is responsible for the weighting of the systems input* $\boldsymbol{u(t)}$ with the use of a NOT gate where $node_i$ is fed *v*, $node_{i+1}$ is fed NOT(*v*) and $node_{i+2}$ is fed *v* etc.

4.2.3. Chua's circuit on breadboard

To build and analyze Chua's circuit we must express its dynamics function (6) in terms of the components and their magnitudes.

$$\begin{aligned} \dot{v}_{C1} &= \frac{1}{R \times C_1}((v_{C2} - v_{C1}) - R \times g(v_{C1})) \\ \dot{v}_{C2} &= \frac{1}{R \times C_2}((v_{C1} - v_{C2}) + R \times i_L) \\ i_L &= -\frac{v_{C2}}{L} \end{aligned} \qquad (18)$$

$$(v_{C1}) = \begin{cases} m_0 v_{C1} + (m_0 - m_1)E_1 & , v_{C1} \le -E_1 \\ m_1 v_{C1} & , -E_1 \le v_{C1} \le E_1 \\ m_0 v_{C1} + (m_1 - m_0)E_1 & , E_1 \le v_{C1} \end{cases}$$

However, Chua's diode isn't readily available as it isn't manufactured so it must be replaced with off-the-shelf electronic components. There are different configurations to replace the element, I will use one that uses two op-amps and four resistors described in [16] as shown below in Fig 21.

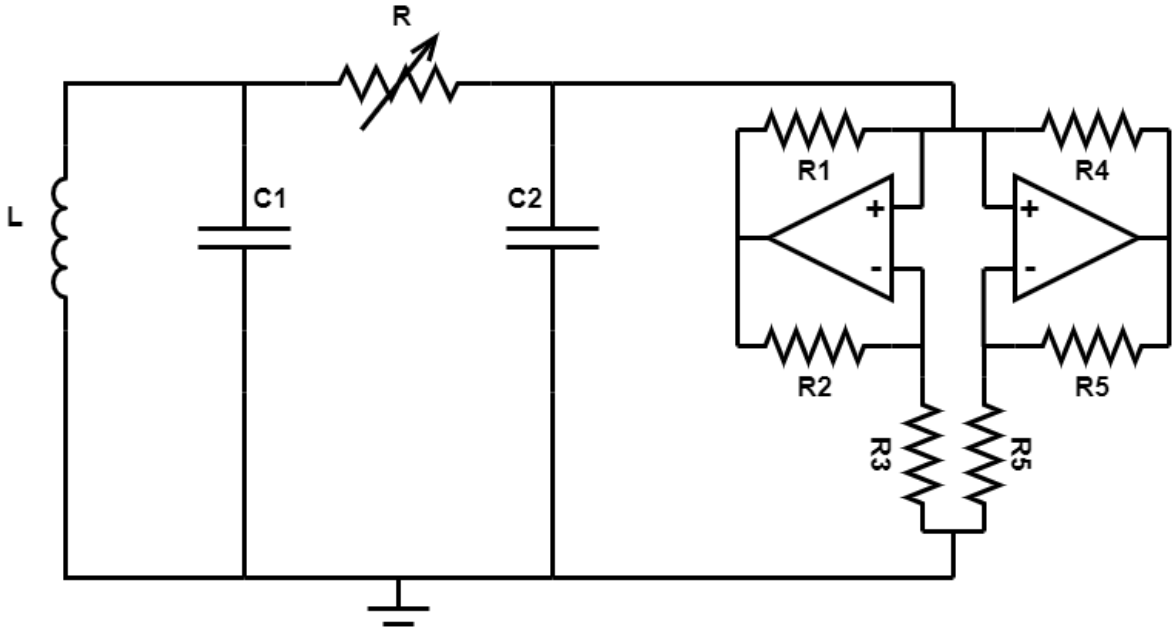


Fig 21. Chua's circuit realized with OP-Amps.

Component	Magnitude	Tolerance
Inductor L	18mH	5%
Capacitor C1	10nF	5%
Capacitor C2	100nF	5%
Resistor R1	220Ω	5%
Resistor R2	220Ω	5%
Resistor R3	2.2kΩ	5%
Resistor R4	22kΩ	5%
Resistor R5	22kΩ	5%
Resistor R6	3.3kΩ	5%
Potentiometer R	2kΩ	Not applicable
2 Batteries	9V	Not applicable

Table 1. Component list of Chua's circuit.

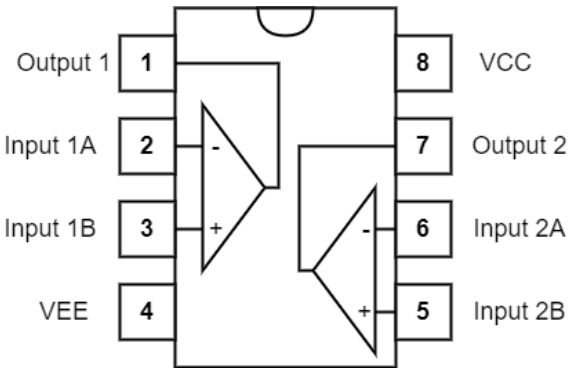


Fig 22. TL062 Pinout.

Furthermore, it's possible to rewire the replacement of the Chua diode in Fig 21 using an Op-Amp IC, namely TL062 (Fig 22). The circuit was then wired up on a breadboard with the components listed in Table 1. To display the double scroll an oscilloscope was connected to the circuit. The oscilloscopes X input was clipped to the leg of capacitor C1 and ground to measure the voltage across it (X=v_{C1}), the Y input was similarly set to measure the voltage across capacitor C2 (Y=v_{C2}).

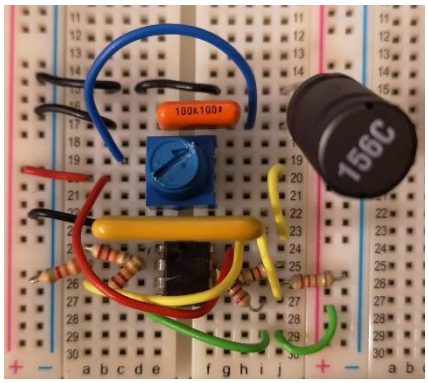


Fig 23. Photo of Chua's circuit on a breadboard.

Following that the Oscilloscope was set into XY mode, and the potentiometer slowly turned until the double scroll was visible on screen. This could be done for any of the three input combinations (X=v_{C1} , Y=v_{C2} and Z=i_L) to show any 2D perspective of the double scroll.

4.3. Performance Tests

The Constructed software Cyclic Reservoir Computer was scanned for the optimal choice of configuration parameters *r* and *v*. And then tested with the optimal configuration parameters on the following chaotic time series: Mackey-Glass, Lorenz and Chua's.

4.3.1. Software RC parameter scan

The results for the scan of a 20-neuron reservoir for optimal parameters on the Mackey-Glass series showed that the system has a very large spread of performances (MSE values).

v \ r	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0.1	0.0075	0.383	0.0133	0.3584	0.0055	2.1126	0.0101	0.0048	27.4992	0.2619
0.2	0.0955	0.0174	0.0585	0.0110	0.0960	0.2711	0.0598	1.3424	0.03010	0.0306
0.3	0.5237	12.0319	0.3991	0.0844	0.0574	8e-04	0.0054	0.1149	0.0017	0.0027
0.4	5.0907	0.0243	0.0273	1.7718	4.5e+04	7.4e-05	0.1727	0.0726	17.9478	0.5009
0.5	0.0232	0.0034	0.0078	0.0091	0.0031	0.0559	3.2830	0.1278	4.8e-04	0.0415
0.6	0.0047	0.3775	1.1280	0.3576	0.0654	0.0060	0.0112	3.3484	0.3325	0.0144
0.7	0.0038	0.0589	0.0193	0.1510	0.0015	3.6e-04	0.0279	0.0627	2.8e-04	0.4110
0.8	7.6e-05	0.0122	0.5759	9.6e-04	0.0131	0.0069	0.0275	2.6e-05	0.0242	0.0012
0.9	0.0309	0.1038	0.0104	0.0013	0.0487	0.0031	0.0013	0.0584	0.0049	0.1129
1	0.0202	0.7437	0.0087	0.0014	0.6202	0.0089	0.0596	0.0829	0.1491	81.5257

Table 2. Table of MSE values for parameter scan on Mackey-Glass and resolution 10.

As shown in Table 2 some values like for example $(r, v) = (0.5, 0.4)$ gives a very high MSE in comparison to the rest of the table, this means that if we were to plot the scan results in a mesh or colormap we would not be able to see any meaningful result as the spike from this high value would mean that all the other values are not discernable from each other like shown below in .

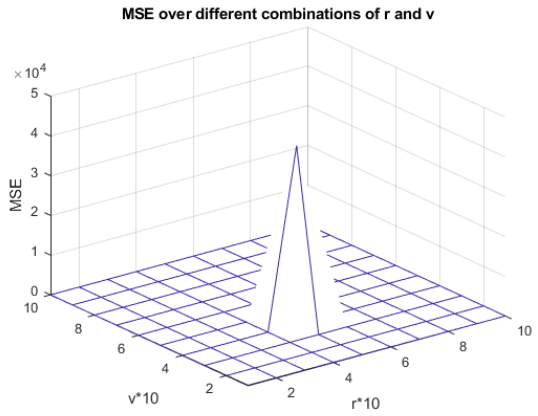


Fig 24. Parameter scan mesh for Mackey-Glass (N=20, config_res = 10).

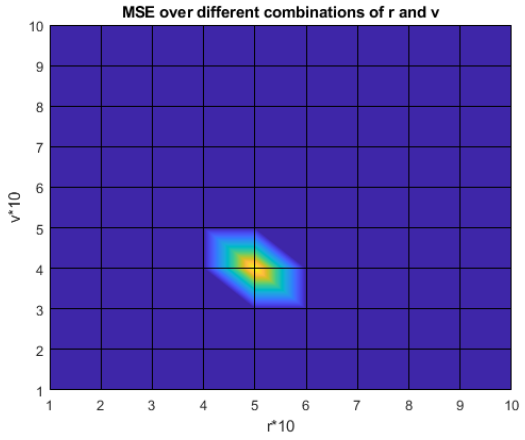


Fig 25. Parameter scan colormap for Mackey-Glass (N=20, config_res = 10).

To fix this issue a logarithmic scaling was applied on the MSE values (Z axis) and any value above 0.1 was cut to 0.1 as we don't care about them.

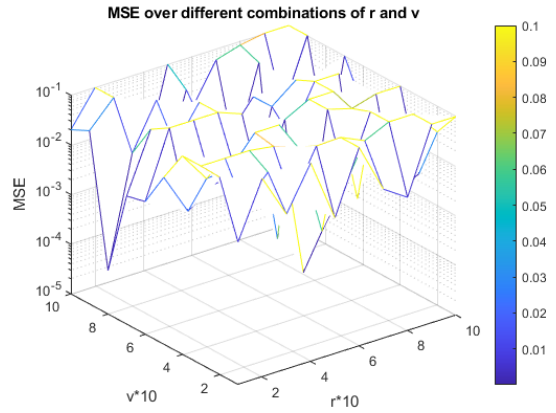


Fig 26. Parameter scan mesh for Mackey-Glass with logarithmic scaling and upper limit of 0.1. (N=20, config_res = 10).

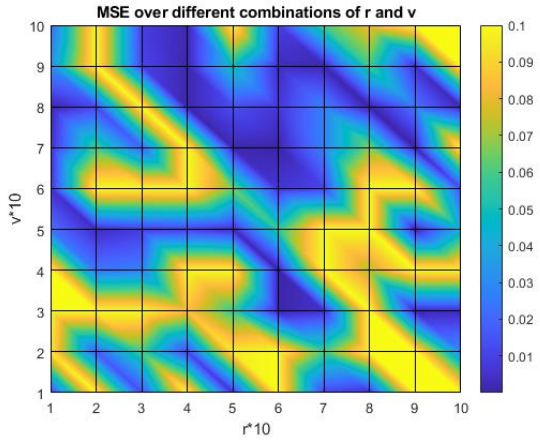


Fig 27. Parameter scan colormap for Mackey-Glass with upper limit of 0.1. (N=20, config_res = 10).

Following this, two reservoirs with 20 and 50 neurons were scanned with resolution 40 (*r* and *v* step size of 0.025) on each chaotic time series to find the optimal configuration of the reservoir for testing.

4.3.2. Software RC benchmarks

4.3.2.1. Software RC on Mackey-Glass delayed time series

The dataset used is a Mackey-Glass chaotic time series with delay=17 obtained from [17]. The system trains for 2000 steps and is tested on 2000 steps. First a Reservoir with 20 neurons was scanned for the optimal parameters with resolution 40.

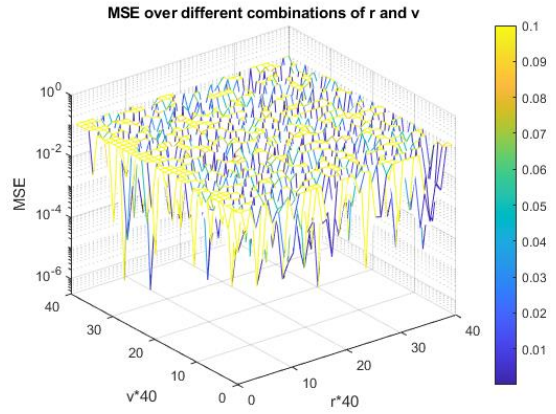


Fig 28. Parameter scan mesh for Mackey-Glass with logarithmic scaling and upper limit of 0.1. (N=20, config_res = 40).

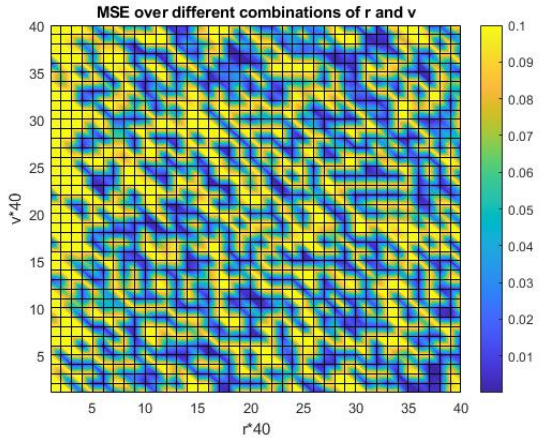


Fig 29. Parameter scan colormap for Mackey-Glass with upper limit of 0.1. (N=20, config_res = 40).

The best performance was found at $(r, v) = (0.85, 0.25)$ with an $MSE = 3.2424 \times 10^{-7}$.

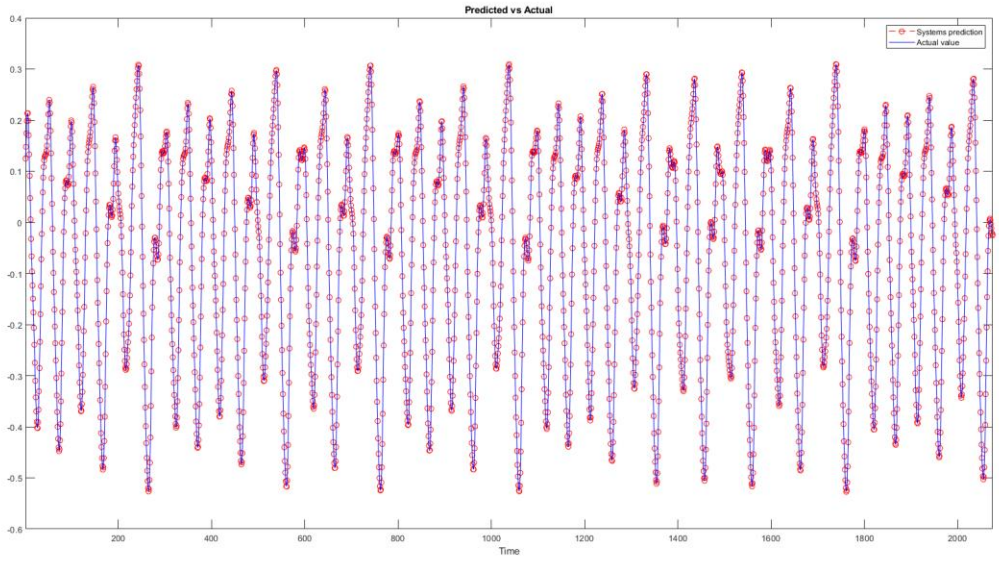


Fig 30. Predicted vs ground truth of RC on Mackey-Glass. (N=20)

Then this process was repeated for a Reservoir with 50 neurons:

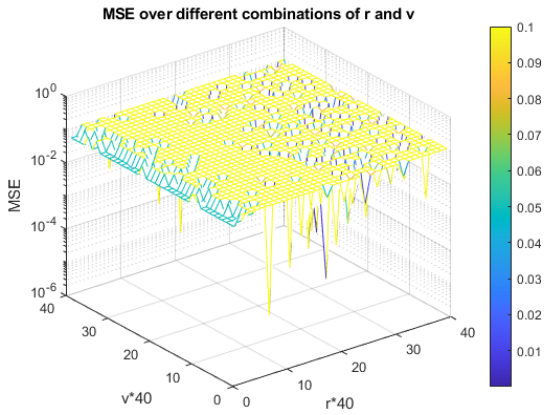


Fig 31. Parameter scan mesh for Mackey-Glass with logarithmic scaling and upper limit of 0.1. (N=50, config_res = 40).

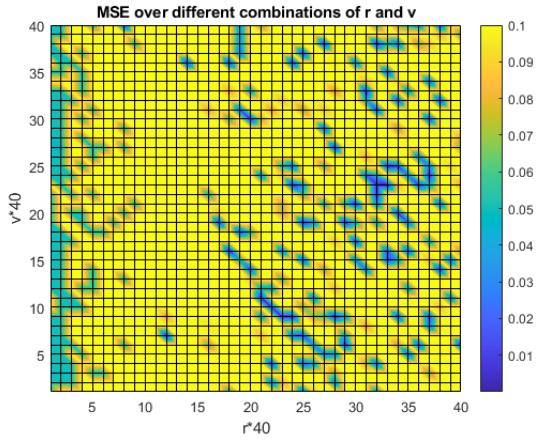


Fig 32. Parameter scan colormap for Mackey-Glass with upper limit of 0.1. (N=50, config_res = 40).

The best performance was found at $(r, v) = (0.825, 0.625)$ with an $MSE = 4.9837 \times 10^{-6}$.

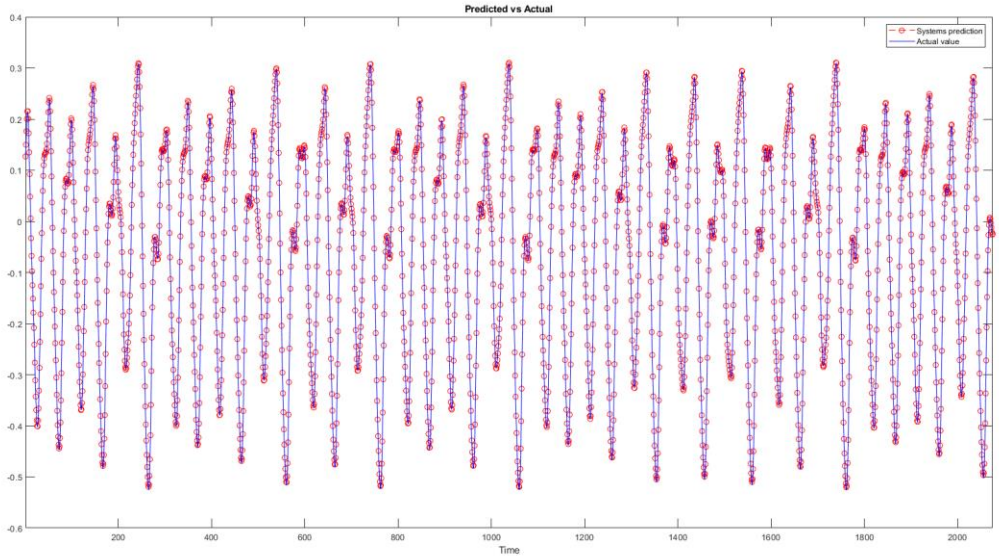


Fig 33. Predicted vs ground truth of RC on Mackey-Glass. (N=50)

4.3.2.2. Software RC on Chua's chaotic time series

The dataset used was a MATLAB simulation of the built Chua's circuit as shown in Fig 21. The system was fed the X (v_{C1}) signal of the circuit. The tests were done in the same way as in the section before but the training and testing were both 1400 steps. First a 20-neuron RC was tested:

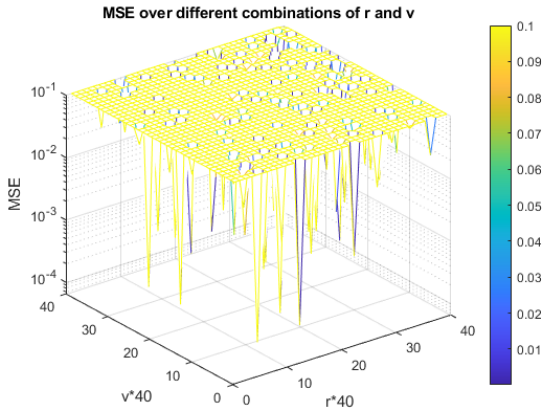


Fig 34. Parameter scan mesh for Chua with logarithmic scaling and upper limit of 0.1. (N=20, config_res = 40).

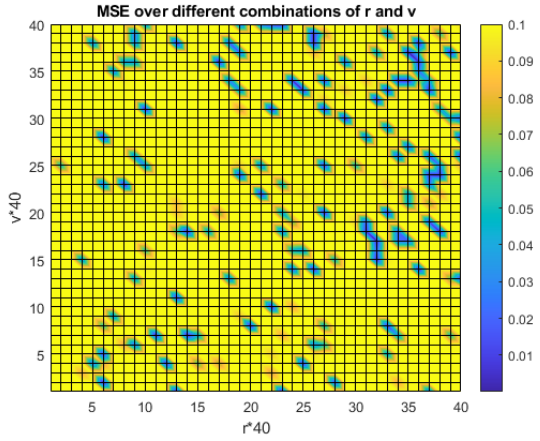


Fig 35. Parameter scan colormap for Chua with upper limit of 0.1. (N=20, config_res = 40).

The best performance was found at $(r, v) = (0.575, 1)$ with an $MSE = 6.5205 \times 10^{-5}$.

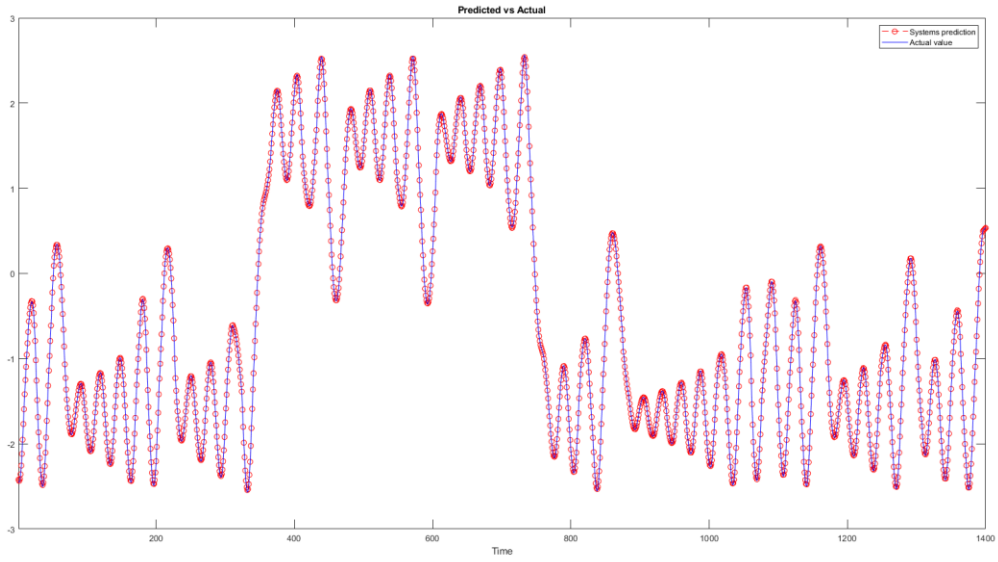


Fig 36. Predicted vs ground truth of RC on Chua. (N=20)

Secondly a 50-neuron RC:

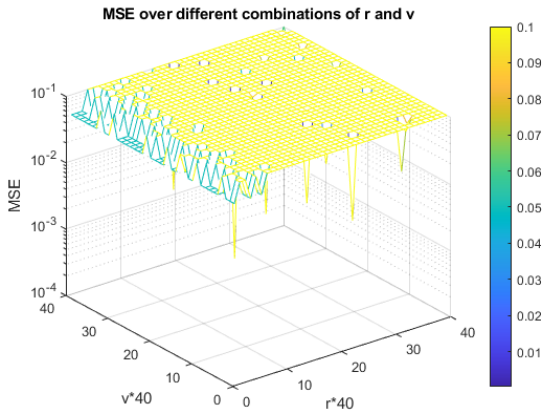


Fig 37. Parameter scan mesh for Chua with logarithmic scaling and upper limit of 0.1. (N=50, config_res = 40).

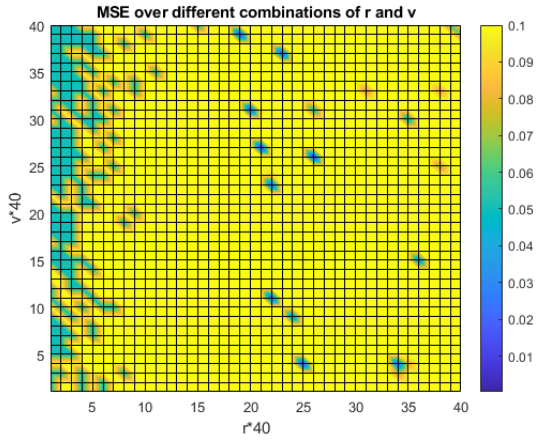


Fig 38. Parameter scan colormap for Chua with upper limit of 0.1. (N=50, config_res = 40).

The best performance was found at $(r, v) = (0.525, 0.675)$ with an $MSE = 2.8223 \times 10^{-4}$.

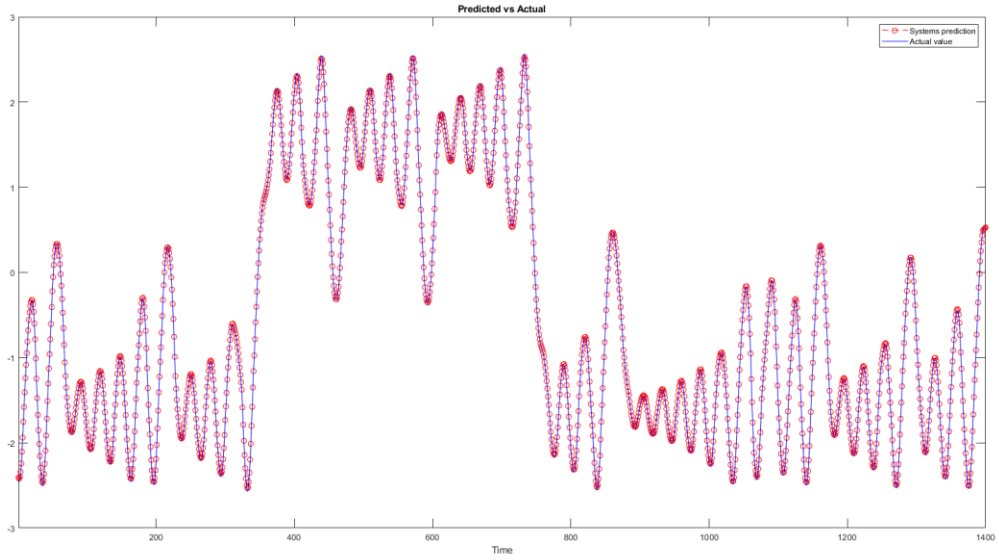


Fig 39. Predicted vs ground truth of RC on Chua. (N=50)

4.3.2.3. Software RC on Lorenz chaotic time series

The dataset used was a MATLAB simulation of a chaotic Lorenz system with the dynamics constants from equation (12) being set to ($\rho = 28$, $\sigma = 10$, $\beta = \frac{8}{3}$). The signal Y of the Lorenz system was used as input for the RC. The system was trained on 2000 steps and tested on 2550 steps. First the performance was tested with a 20-neuron RC:

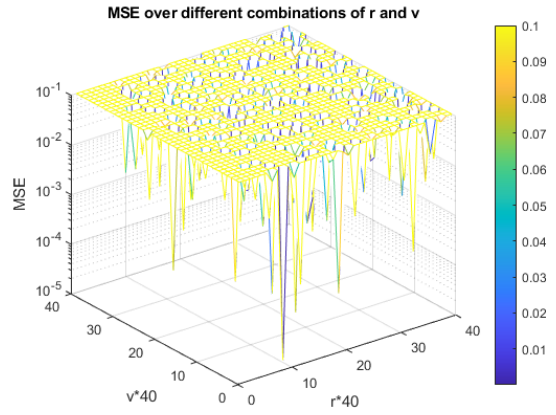


Fig 40. Parameter scan mesh for Lorenz with logarithmic scaling and upper limit of 0.1. (N=20, config_res = 40).

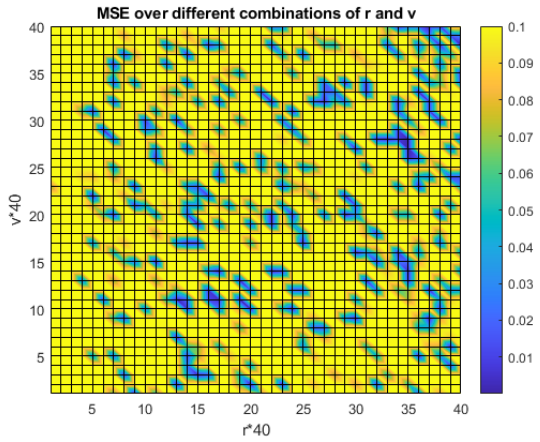


Fig 41. Parameter scan colormap for Lorenz with upper limit of 0.1. (N=20, config_res = 40).

The best performance was found at $(r, v) = (0.375, 0.025)$ with an $MSE = 1.3643 \times 10^{-5}$.

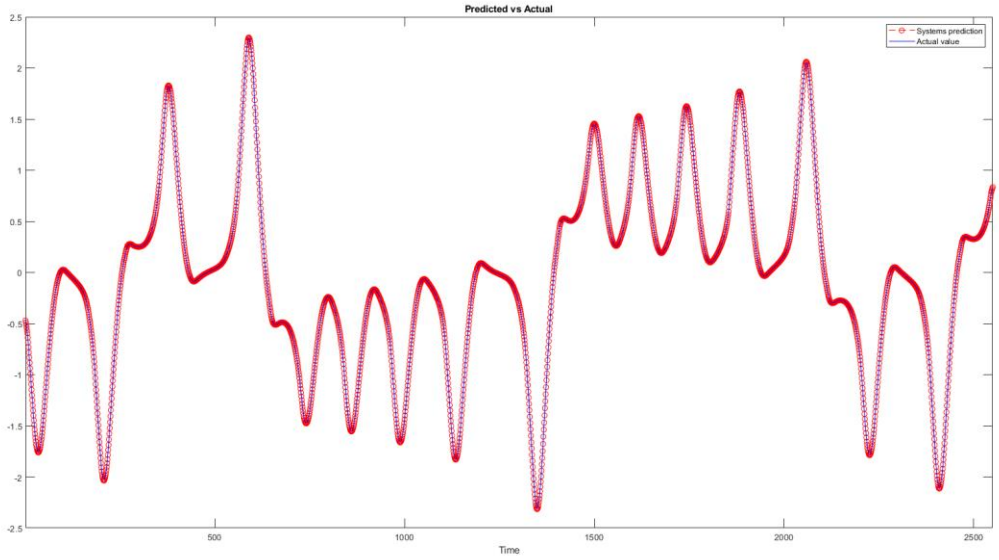


Fig 42. Predicted vs ground truth of RC on Lorenz. (N=20)

Lastly a 50-neuron RC was also tested:

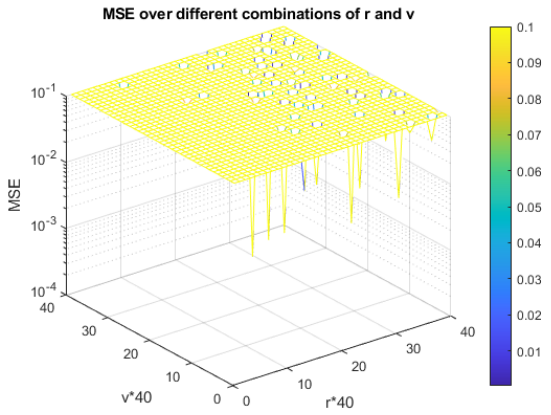


Fig 43. Parameter scan mesh for Lorenz with logarithmic scaling and upper limit of 0.1. (N=50, config_res = 40).

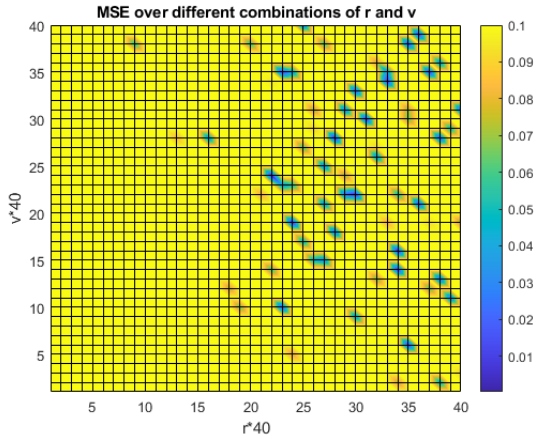


Fig 44. Parameter scan colormap for Lorenz with upper limit of 0.1. (N=50, config_res = 40).

The best performance was found at $(r, v) = (0.225, 0.025)$ with an $MSE = 3.1768 \times 10^{-4}$.

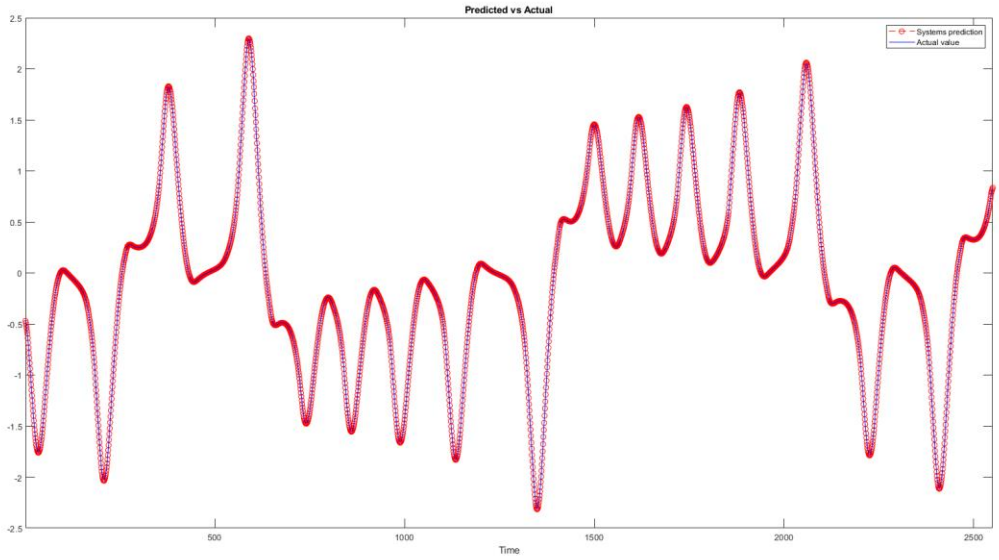


Fig 45. Predicted vs ground truth of RC on Lorenz. (N=50)

5. Discussion and Applications

5.1. Discussion of Results

The results of the performance benchmarks for my Cyclic Reservoir design demonstrate a significant spectrum of performances for the configuration parameter choices and an unexpected general decrease in performance with higher neuron count in the system. Nevertheless, the performance at optimal configuration of parameters *r* and *v* or close to optimal is quite good especially considering its simpler design compared to state-of-the-art chaotic time series forecasting solutions [18] [19] [20].

The first finding that was recorded after the benchmark performance tests is the apparent decrease in performance in the tests with a reservoir with 50 neurons compared to those with 20 neurons. This decrease in performance might however be simply due to the optimal configuration values of *r* and *v* falling in between the values we scan due to the system becoming even more sensitive to the magnitude of the configuration parameters. A test on the Mackey-Glass dataset performed in the same manner as above with the same design and 50 neurons was scanned for the optimal parameters, with the scan step size being set to 0.01 (`config_res` =100).

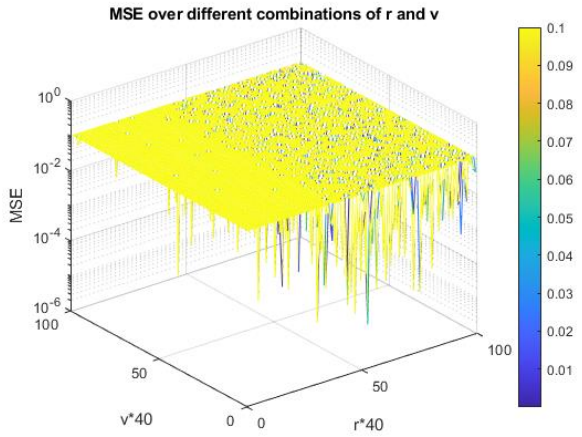


Fig 46. Parameter scan mesh for Mackey-Glass with logarithmic scaling and upper limit of 0.1. (N=50, config_res = 100).

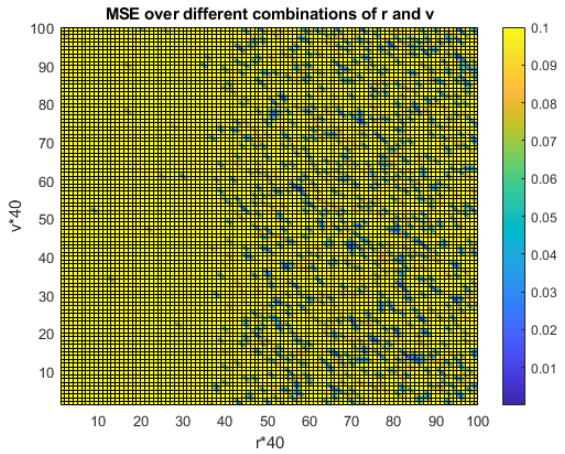


Fig 47. Parameter scan colormap for Mackey-Glass with upper limit of 0.1. (N=50, config_res = 100).

This test yielded a $MSE = 1.3299 \times 10^{-6}$ which while still smaller than the same test with 20 neurons, is an improvement on the lower resolution scan test for 50 neurons. I hypothesize that with a sufficiently increased scan resolution a more optimal configuration of parameters *r* and *v* can be found that yields a lower MSE value for a 50-neuron reservoir than any 20-neuron alternative for most cases. However, the performance obtained in both variants is suitable for its given task and with the added context of this Reservoir Computer being proposed as a low-cost implementation the 20-neuron design is very efficient and any hyper

optimization further to achieve even better results is not justified or warranted in the scope I am dealing with.

The second finding was recorded when using the Chua's chaotic time series to benchmark the RC's performance and is that the designs performance is subpar or even null when the input signal has a magnitude over the range [1, -1]. This was found as a result of Chua's X signal or the voltage across capacitor *C1* v_{C1} being in the range of ~[2.5, -2.5] and the systems performance being subpar for some configuration parameters and not even working for other values. This is most likely due to an overflow inside the system, specifically if the inputs to the node are high even after the weighting, when the activation function is used the result will be close to 1 (the upper limit of the tanh function). This means that the next node will also have high magnitude inputs and therefore quickly all the nodes in the cyclic reservoir are outputting their upper limit and the system breaks down. This was fixed for the performance benchmarks on Chua's chaotic series by rescaling the dataset by a factor of $1/10$ and could be done by a simple voltage transformer on the real circuit.

5.2. Applications

The spectrum for possible fields where Reservoir Computers could be used is truly vast and includes speech recognition [21] [22], robotics [23] [24], video and text processing [25] [26] and medicine [27] [28] among others. The broad range of uses of a RC can be anything like being a black-box tool for nonlinear system modelling, approximation and detection of patterns in temporal functions, filtering and classification and time-series prediction which was demonstrated in this project.

5.2.1 RC in Chaotic Communication

One particular field of interest for myself is the use of Reservoir Computing in the field of Chaotic Communications. Artificial Neural Networks in the context of Chaotic Communications have been researched almost from the inception of Chaotic Communications as a research field [29]. An example is that ANNs have been explored as an encryption and decryption tool for crypto-chaotic communications systems [30].

Consider Chua's chaotic system from equation (6), a chaotic communication based on this chaotic system will have only have 7 parameters: α , β , m_0 , m_1 the initial position of x , y and z. In cryptography these would be considered the keys and when the encoded message can be obtained without having access to all the keys the crypto channel is said to fail. Seven keys is a very small amount and therefore the channel could easily be decrypted by hackers using simple methods like brute-force. The advantage of using ANNs comes then when instead of using either an

analog or numerical Chua system, an ANN trained to recreate Chua's system ($x, y, z = \hat{x}, \hat{y}, \hat{z}$) and used as the transmitter-encryptor and receiver-decryptor in a chaotic communication channel.

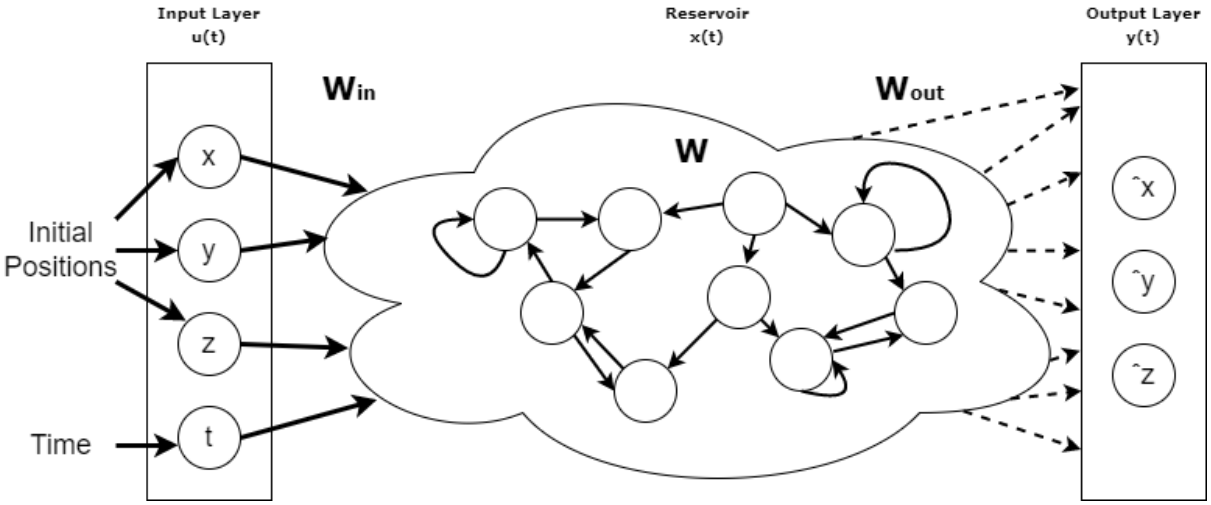


Fig 48. Reservoir Computer that acts as a Chua system.

A RC is perfectly fit for this role as the new keys now for the crypto chaotic channel become the exact Reservoir topology, the weight matrices of $\boldsymbol{W_{in}}$, $\boldsymbol{W}$ and $\boldsymbol{W_{out}}$, the exact number of neurons $\boldsymbol{N}$ and any biases included in the system. As these would be needed to produce the same chaotic dynamic to then obtain the decrypted message. Adapting Fig 7 we obtain a structure of a chaotic communication channel built with two RC's using the X signal from Chua's chaotic system as shown below.

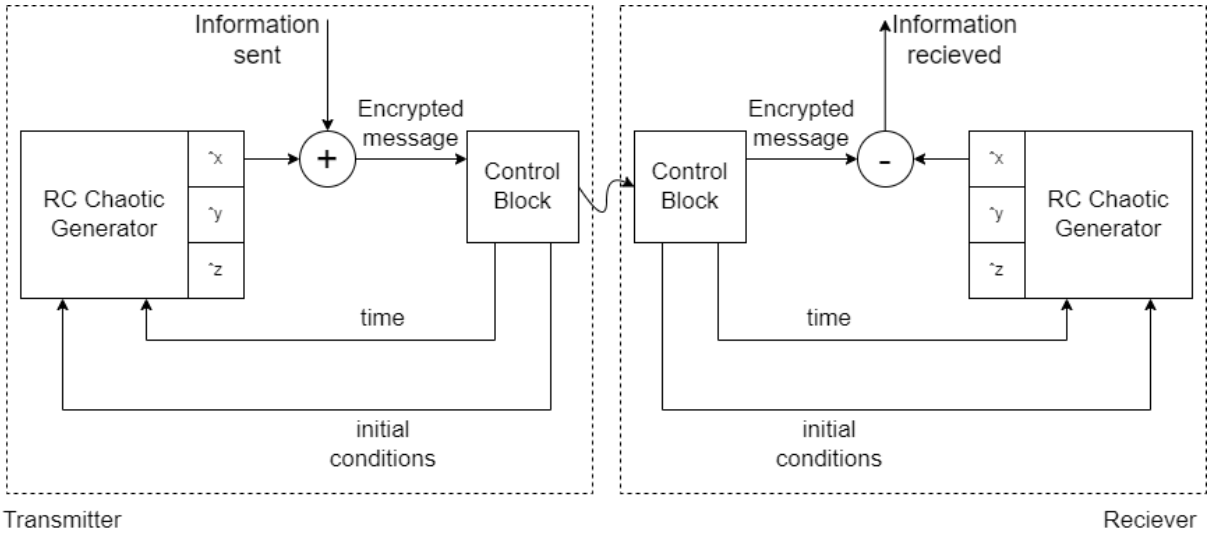


Fig 49. Crypto-Chaotic Communication setup with RCs on the X signal of Chua's system.

This setup might even be more efficient than similar implementations with other ANNs as the RCs topology and dynamics allow for an exponentially larger search list

of parameters that need to be matched in order to recreate the same chaotic signal.

The constructed Cyclic Reservoir Computer from Fig 8 has demonstrated to be able to learn and recreate signal X of Chua's chaotic system in Fig 36 and could be adapted to have as input the initial conditions and time signal and output Chua's X signal.

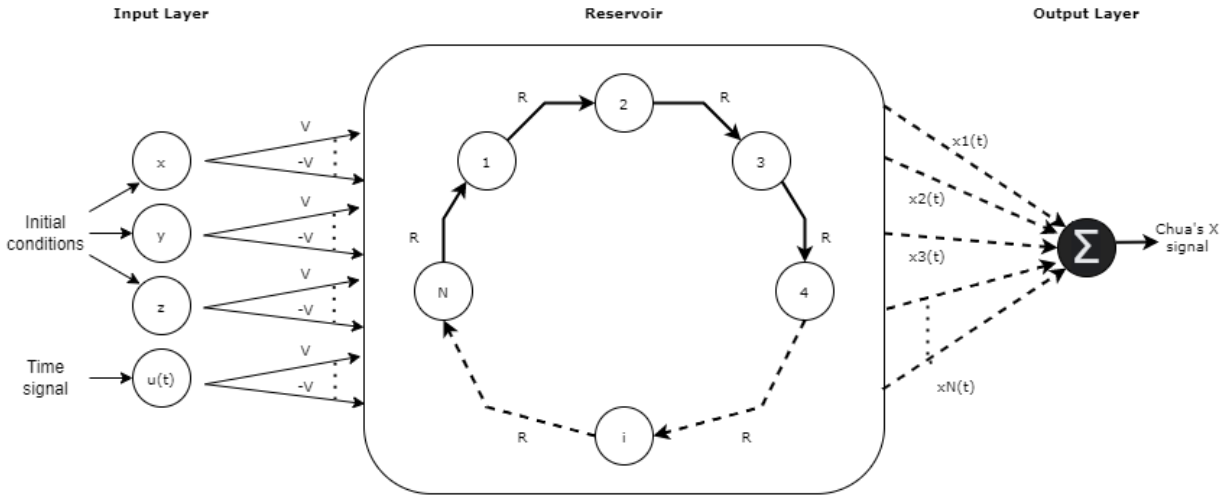


Fig 50. A Cyclic Reservoir acting as a Chua's chaotic signal X generator.

As we now have 4 inputs compared to 1 input in the design I tested, some adjustments to the nodes would have to be made. These adjustments however would be simple and would consist of adding the 3 extra inputs to create a 5-input node. The configuration parameters could be left as are, *r* for intraneuronal connections and *v* for input weighting or a new parameter could be introduced to weight each input separately.

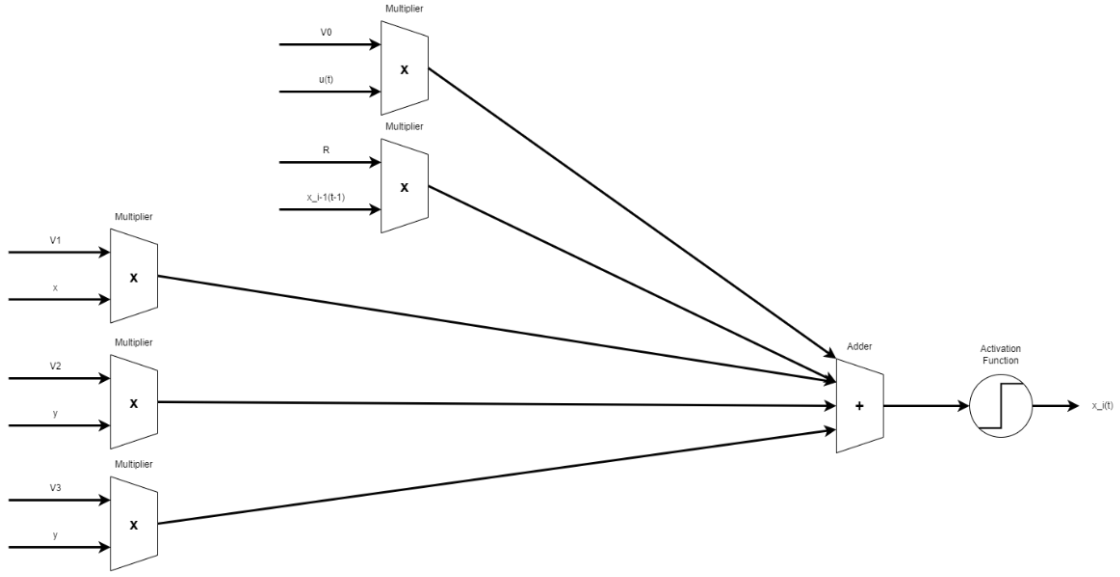


Fig 51. Adapted 5-input neuron.

If we consider the hardware implementation design of a Cyclic Reservoir that was proposed in this paper in section 4.2, together with this chaotic communication setup we could potentially realize a real-world chaotic communication channel using FPGAs with Cyclic RCs that act as generators of a chaotic system that synchronize and form the secure communication channel.

6. Conclusion

In this paper a hardware friendly reservoir topology in Reservoir Computing is tested and benchmarked for time-series recreation in MATLAB using different chaotic time series. The MATLAB tests are done with the Chua's chaotic time series (demonstrated only for the X signal), a Lorenz chaotic time series (demonstrated only for the Y signal) and an outside sourced Mackey-Glass with delay=17 chaotic time series. The tests show an ability of the design to recreate these chaotic time series with good accuracy and highlight some drawbacks of the proposed design, namely its oversensitivity to the configuration parameters, high range of achievable performances and necessity of scaling the input to the system to [-1,1].

Then a low-cost hardware implementation targeting FPGAs of said reservoir topology with Stochastic Computing elements is proposed. The proposed design is constructed in Verilog and on-paper achieves a substantial reduction in necessary hardware requirements and has a high environmental noise tolerance. However, some disadvantages of the design are also outlined, like the longer processing time and theoretically slightly lower accuracy.

Lastly a chaotic communication channel structure is proposed that would make use of the proposed hardware Reservoir Computers on FPGAs as a more secure alternative to analog chaotic circuit based chaotic communication channels and possibly even other implementations of the chaotic communication channel with other ANNs.

The logical next step and area of research would be constructing and exploring the viability of a complete communication package made up of two FPGAs with the proposed Reservoir Computer design and chaotic communication channel structure outlined in this paper.

Appendix

A. MATLAB code

A.1. Cyclic Reservoir Computer

A.1.1. Train and Test reservoirs

```
%collect a matrix of node outputs for training
for j = 1:trainLen
    node_out(j,1) = Activation_function(v*data(j+initLen,1)+r*node_out(j,N));
    for k = 2:N
        if rem(k,2)==0
            node_out(j,k) = Activation_function(-v*data(j+initLen,1)+r*node_out(j,k-1));
        end
        if rem(k,2)~=0
            node_out(j,k) = Activation_function(v*data(j+initLen,1)+r*node_out(j,k-1));
        end
    end
end
%If we are recreating the input then y_target is equal to system input
y_target = data(initLen:trainLen+initLen-1);
%call the least squares regression
W_out = least_squares_regression(node_out, y_target);
```

Fig 52. MATLAB code for the reservoirs training.

Where `trainLen` is the time steps we train the system for, `initLen` is the initial transient period we discard from the training process, `node_out` is a matrix of node outputs during training, `W_out` is the resulting vector of trained weights and `N` is the node count.

```
%we test the system
for z = 1:testLen
    node_x(z,1) = Activation_function(v*data(z+initLen+trainLen,1)+r*node_x(z,N));
    for m = 2:N
        if rem(m,2)==0
            node_x(z,m) = Activation_function(-v*data(z+initLen+trainLen,1)+r*node_x(z,m-1));
        end
        if rem(m,2)~=0
            node_x(z,m) = Activation_function(v*data(z+initLen+trainLen,1)+r*node_x(z,m-1));
        end
    end
end
%we calculate the output of the system and the MSE
for n = 1:testLen
    for c = 1:N
        y_out(n) =  y_out(n) + (node_x(n,c)*W_out(c,1));
    end
end
y_target_final = data(trainLen+initLen+1:testLen+trainLen+initLen);
MSE = mean((y_out-y_target_final).^2);
```

Fig 53. MATLAB code for the reservoirs testing.

Where `testLen` is the time steps we test the system for, `node_x` is a matrix of node outputs during testing, `y_out` is a vector over the systems output over testing,

`y_target_final` is the input to the system during testing and `MSE` is the mean-squared-error.

A.1.2. Least Squares Regression function

```
function W_out = least_squares_regression(X, Y_target)
    K = inv(X.'*X);
    W_out = K * X.'*Y_target;
end
```

Fig 54. MATLAB code for least-squares-regression.

A.2. Code for Lorenz and Chua systems

A.2.1. Lorenz system

```
function [x,y,z] = lorenz_system(rho, sigma, beta)
    eps = 0.000001; %ode solver precision
    T = [0 25]; %time interval
    initV = [0 1 1.05]; %initial point

    options = odeset('RelTol',eps,'AbsTol',[eps eps eps/10]);
    [T,X] = ode45(@(T,X) right_hand_side(T, X, rho, sigma, beta), T, initV, options);

    x = X(:,1);
    y = X(:,2);
    z = X(:,3);
end

function dx = right_hand_side(T, X, rho, sigma, beta)
    %Evaluates the right hand side equations of the Lorenz system
    dx = zeros(3,1);
    dx(1) = sigma*(X(2) - X(1));
    dx(2) = X(1)*(rho - X(3)) - X(2);
    dx(3) = X(1)*X(2) - beta*X(3);
end
```

Fig 55. MATLAB script for Lorenz system.

A.2.2. Chua system

```
function out = chua_function(t,p)
    % t = Time, p = Position
    C1  = 10*10^(-9); %10nF
    C2  = 100*10^(-9); %100nF
    R = 1800; %1.5kOhm
    G = 1/R;
    L = 18*10^(-3); %18mH
    m0 = -1.2;
    m1 = -0.7;
    x = p(1);
    y = p(2);
    z = p(3);
    h = m1*x+0.5*(m0-m1)*(abs(x+1)-abs(x-1));
    xdot = (1/C1)*(G*(y-x-h));
    ydot = (1/C2)*(G*(x-y)+z);
    zdot  = -(1/L)*y;
    out = [xdot ydot zdot]';
end
```

Fig 56. MATLAB script for Chua's system.

B. Verilog code

B.1. Stochastic Circuitry

B.1.1. B2P module

```
//Binary to pulse stochastic converter using a comparator and random number generator
module B2P(
    input wire [15:0]binary_in,
    input wire clk,
    output wire pulsed_out
);

wire [15:0]lfsr_out; //output from LFSR module
wire lfsr_status; //flag check if LFSR is done
wire lt, gt, eq; //flag checks of comparator outputs (less than, greater than and equal)

// calling the LFSR module to get our pseudo-random number
LFSR random_n(clk, 1, {16{1'b0}} , lfsr_out, lfsr_status);

//calling the comparator module to compare the input number and the LFSR's output
comp_16bit comp(binary_in, lfsr_out, lt, gt, eq);

//if the binary input is greater than the pseudo-random number then the output is 1, otherwise 0
assign pulsed_out = gt ? 1'b1:
                    lt ? 1'b0:
                    eq ? 1'b0:
                         1'b0;
endmodule
```

Fig 57. Verilog code for B2P module.

Where `LFSR` is the LFSR module below and `comp_16bit` is a 16-bit comparator module.

B.1.2. P2B module

```
//Stochastic pulse back to binary value converter module
module P2B(
    input wire clk, enable, p_input, //p_input is the pulse stream to convert to binary
    input wire [15:0]Nc,
    output wire [15:0]b_output //16bit binary output
    );

wire count_enable; //enable pin for counter
wire count_reset; // reset pin for counter
wire [15:0]m1, m2; //two 16bit internal connections
wire lt, gt, eq; //less than, greater than and equal boolean outputs for comparator

// enable signal for counter is the output of enable for module AND the pulse stream
assign count_enable = p_input & enable;

//one counter to count the number of high pulses in p_input
up_counter_16 counter1(clk, count_reset, count_enable, m1);

//second counter to count up while module is enabled
up_counter_16 counter2(clk, count_reset, enable, m2);

//comparator module call to check if second counter has counted to Nc
comp_16bit check(Nc, m2, lt, gt, eq);

//if second counter reaches Nc reset all counters
assign count_reset = eq;

// if counters are reset a register passes on the accumulated number of high values in the
stochastic pulse stream as the converted binary value
register_16bits hold(count_reset, clk, m1, b_output);
endmodule
```

Fig 58. Verilog code for P2B module.

Where `Nc` is the evaluation constant (2^{16}), `up_counter_16` is a 16-bit counter module that counts only up and `register_16bits` is a 16-bit register module to hold a value.

B.1.3. LFSR module

```
//LFSR module to create a pseudo-random 16bit number given a seed
module LFSR(
    input wire clk,
    input [15:0]seed, //seed for the LFSR to preload
    output [15:0]lfsr_16bits,
    output lfsr_done //flag that LFSR is done generating a pseudorandom number
    );
reg [16:0]r_LFSR = 0; //initialize the LFSR register value to 0
reg r_XNOR;

always@(posedge clk)
    begin
        if(enable == 1'b1)
            begin
                r_LFSR <= seed;
            end
end
always@(*)
    begin
        case(16)
            3: begin
                r_XNOR = r_LFSR[3] ^~ r_LFSR[2];
            end
            4: begin
                r_XNOR = r_LFSR[4] ^~ r_LFSR[1];
            end
            5: begin
                r_XNOR = r_LFSR[5] ^~ r_LFSR[3];
            end
            6: begin
                r_XNOR = r_LFSR[6] ^~ r_LFSR[5];
            end
            7: begin
                r_XNOR = r_LFSR[7] ^~ r_LFSR[6];
            end
            8: begin
                r_XNOR = r_LFSR[8] ^~ r_LFSR[6] ^~ r_LFSR[5] ^~ r_LFSR[4];
            end
            9: begin
                r_XNOR = r_LFSR[9] ^~ r_LFSR[5];
            . . . . .
            . . . . .
            . . . . .
            31: begin
                r_XNOR = r_LFSR[31] ^~ r_LFSR[28];
            end
            32: begin
                r_XNOR = r_LFSR[32] ^~ r_LFSR[22] ^~ r_LFSR[2] ^~ r_LFSR[1];
            end

        endcase
end
assign lfsr_16bits = r_LFSR[16:1];
assign lfsr_done = (r_LFSR[16:1] == seed) ? 1'b1 : 1'b0;
endmodule
```

Fig 59. Verilog code of LFSR module.

B.2. Reservoir and Nodes

B.2.1. Node module

```
//two input node that does operations with stochastic bit streams and then convert back to binary
for the rest of the computations
module Node(
    input wire clk, Nc,
    input wire w1, //configuration parameters V's stochastic bit stream
    input wire i1, //reservoir input stochastic bitstream
    input wire w2, //configuration parameters R's stochastic bit stream
    input wire [15:0]rand, //pseudo-random number from LFSR
    input wire [15:0]i2, //output from another neuron
    output wire [15:0]x_i //output from this neuron
    );
wire x_anterior; // stochastic bitstream for i2
wire lt, gt, eq; //Boolean outputs of comparator
wire x1, x2, y; //internal wires
wire [15: 0] p2b_out; //output of P2B module
wire [16: 0] shift_out; //output from shiftin binary word
wire [15: 0] node_o; //node output

comp_16bit agtb(i2, rand, lt, gt, eq); //calling comparator for B2P for i2
assign x_anterior = gt ? 1'b1: //B2P operation for i2
                    lt ? 1'b0:
                    eq ? 1'b0:
                         1'b0;
//XNOR of w1 and i1 stochastic bitstreams (x1)
assign x1 = ~(w1 ^ i1);
//XNOR of w2 and x_anterior stochastic bitstreams (x1)
assign x2 = ~(w2 ^ x_anterior);
//Multiplexer of x1 and x2 with output y
mux_2 add(clk, x1, x2, y);
//Stochastic bitstream back to binary conversion of y
P2B p2b_1(clk, 1'b1, y, Nc, p2b_out);
//multiply the output of the P2B by 2 by shifting the bits
assign shift_out = p2b_out << 1;
//calling the activation function tahn to get the final output of the neuron
Activation_function sig(shift_out, node_o);
assign x_i = node_o;
endmodule
```

Fig 60. Verilog code for Node module.

Where `mux_2` is a multiplexer module running on a binary clock, and `Activation_function` is the activation function module described below.

B.2.2. Reservoir module

```
module Reservoir(
    input wire [15:0]u_t, //input to reservoir
    input clk, Nc, //clk and evaluation parameter
    input wire [15:0]R, V, seed, //r and v are parameters and seed is the seed for the LFSR
    output wire [15:0]y_out[0:N-1] // array of outputs of the neurons
    parameter N = 20;//number of neurons
    );
wire i1, ri, vi;//these are the stochastic bitstreams from u_t, R and V respectively
wire [15:0]out_x[0:N-1];//array of the neuron outputs
wire [15:0] rand; // pseudo random output from LFSR
wire rand_done;// status on LFSR

//calling the LFSR module to create a pseudo-random 16 bit number
LFSR lfsr_1(clk, 1'b1, seed, rand, rand_done);
//Binary to pulse stochastic conversion of input to reservoir to be used inside neurons
B2P input_b2p(u_t, clk, i1);
//Binary to pulse stochastic conversion of R to reservoir to be used inside neurons
B2P r_b2p(R, clk, ri);
//Binary to pulse stochastic conversion of V to reservoir to be used inside neurons
B2P v_b2p(V, clk, vi);
//initialization of first node in the cyclic reservoir
Node n0(clk, Nc, vi, i1, ri, rand, out_x[19], out_x[0]);
//assigning each neurons connections in the cyclic reservoir
genvar i;
generate
    for( i = 1; i < N; i = i + 2) //assigning NOT V to every second neuron
        begin
            Node ni(clk, Nc, vi, i1, ri, rand, out_x[i-1], out_x[i]);
        end
    for( i = 2; i <N; i = i + 2)
        begin
            Node ni(clk, Nc, ~vi, i1, ri, rand, out_x[i-1], out_x[i]);
        end
endgenerate
. . .
. . . //assign all neuron outputs to the reservoir outputs
. . .
//out_x = y_out
endmodule
```

Fig 61. Verilog code and pseudocode for the Reservoir module.

B.2.3. Activation function

```
module Activation_function(//3piecewise linear approximation of the tanh function
    input wire [16:0] x, //input to activation function
    output wire [15:0] f //output of activation function and node output
);
//if x > 16b'1111111111111111, f= 16b'1111111111111111
if( $signed(x) > $signed({1'b0, 16'h7FFF}))
        begin
            f = 16'h7FFF;
        end
//if x < 16b'0000000000000000, f= 16b'0000000000000000
else if( $signed(x) < $signed({1'b1, 16'h8000}))
        begin
            f = $signed(16'h8000);
         end
//else f = x
else
        begin
            f = $signed({x [16], x [14:0]});
         end
endmodule
```

Fig 62. Verilog code of 3-piece approximation of tanh.

Bibliography

[1] R. L. Devaney, An Introduction to Chaotic Dynamical Systems, 1989.

[2] T. S. P. a. L. O. Chua, Practical Numerical Algorithms for Chaotic Systems, 1989.

[3] L. O. Chua, "The Genesis of Chua's circuit," 1992.

[4] L. O. Chua, M. Komuro and T. Matsumoto, "The double scroll family," *IEEE Transactions on Circuits and Systems Ii: Analog and Digital Processing,* 1986.

[5] P. Verzelli, "Learning Dynamical Systems Using Dynamical Systems - The Reservoir Computing Approach," Universita della Svizzera Italiana, 2022.

[6] L. M. Pecora and T. L. Caroll, "Synchronization in chaotic system," *Physical Review Letters,* vol. 64, no. 10.1063/14917383, p. 821, 1990.

[7] L. M. Pecora and T. L. Caroll, "Driving systems with chaotic signals," *Phys. Rev. A,* vol. 44, no. 4, pp. 2374-2383, 1991.

[8] M. K. Cuomo and V. A. Oppenheim, "Circuit Implementation of Synchronized Chaos with Applications to Communications," *Physical Review Letters,* vol. 71, no. 1, pp. 65-68, 1993.

[9] A. Rodan and P. Tino, "Minimum complexity echo state network," *IEEE Transactions on Neural Networks,* vol. 22, no. 1, pp. 131-144, 2011.

[10] S. L. Toral , J. M. Quero and L. G. Franquelo, "Stochastic pulse coded arithmetic," *2000 IEEE International Symposium on Circuits and Systems (ISCAS),* vol. 1, pp. 599-602, 2000.

[11] A. Morr, V. Canals, A. Oliver, L. M. Alomar and L. J. Rosello, "Ultra-Fast Data-Mining Hardware Architecture Based on Stochastic Computing," *PLoS ONE 10,* 2015.

[12] S. H. Brown, "Multiple Linear Regression Analysis: A Matrix Approach with," *Alabama Journal of Mathematics,* no. 34, pp. 1-3, 2009.

[13] M. L. Alomar Barcelo, "Methodologies for hardware implementation of reservoir computing systems," Universitat de les Illes Balears, 2017.

[14] M. L. Alomar, V. Canals, N. Perez-Mora, V. Martinez-Moll and J. L. Rosello, "FPGA-Based Stochastic Echo State Networks for Time-Series Forecasting," *Comput Intell Neurosci,* 2016.

[15] V. J. Canals Guinand, "Implementacion en hardware de sistemas de alta fiabilidiad basados en metodologias estocasticas," Universitat de les Illes Balears, 2012.

[16] M. P. Kennedy, "Robust OP Amp Realization of Chua's Circuit," *Frequenz,* vol. 46, no. 3-4, pp. 66-80, 1992.

[17] M. Lukosevicius, "A Practical Guide to Applying Echo State Networks," in *Neural Networks: Tricks of the Trade*, 2012.

[18] M. Lukosevicius, H. Jeager and B. Schrauwen, "Reservoir Computing Trends," *KI - Kunstliche Intelligenz,* vol. 23, pp. 365-371, 2012.

[19] Q. Li and R.-C. Lin, "A New Approach for Chaotic Time Series Prediction Using Recurrent Neural Network," *Mathematical Problems in Engineering,* 2016.

[20] R. Chandra, S. Goyal and R. Gupta, "Evaluation of deep learning models for multi-step ahead time series prediction," 2021.

[21] A. Jalalvand, F. Triefenbach, K. Demuynck and J.-P. Martens, "Robust continuous digit recognition using reservoir computing," *COMPUTER SPEECH AND LANGUAGE,* vol. 30, no. 1, pp. 135-158, 2015.

[22] Y. Zhong, J. Tang, X. Li, B. Gao, H. Qian and H. Wu, "Dynamic memristor-based reservoir computing for high-efficiency temporal signal processing," *Nature Communications,* vol. 12, no. 1, p. 408, 2021.

[23] T. Li, K. Nakajima, M. Cianchetti, C. Laschi and R. Pfeifer, "Behavior switching using reservoir computing for a soft robotic arm," in *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 4918-4924.

[24] E. A. Antonelo, B. Schrauwen, X. Dutoit, D. Stroobandt and M. Nuttin, "Event Detection and Localization in Mobile Robot Navigation Using Reservoir Computing," in *Artificial Neural Networks - ICANN 2007*, 2007, pp. 660-669.

[25] A. Jalalvand, B. Vandersmissen, W. De Neve and E. Mannens, "Radar Signal Processing for Human Identification by Means of Reservoir Computing Networks," in *2019 IEEE Radar Conference (RadarConf)*, 2019, pp. 1-6.

[26] M. H. Tong, A. D. Bickett, E. M. Christiansen and G. W. Cottrell, "Learning grammatical structure with Echo State Networks," *Nerual Networks: the official journal of the International Neuural Network Society,* vol. 20, pp. 424-432, 2007.

[27] P. Buteneers, D. Verstraeten, B. Van Nieuwenhyuse, D. Stroobandt, R. Raedt, K. Vonck, P. Boon and B. Schrauwen, "Real-time detection of epileptic

seizures in animal models using reservoir computing," *Epilepsy Research,* vol. 103, no. 2-3, pp. 124-134, 2013.

[28] M. Cucchi, C. Gruener, L. Petrauskas, P. Steiner, H. Tseng, A. Fischer, B. Penkovsky, C. Matthus, P. Birkholz, H. Kleemann and K. Leo , "Reservoir computing with biocompatible organic electrochemical networks for brain-inspired biosignal classification," *Science Advances,* vol. 7, no. 34, p. eabh0693, 2021.

[29] V. Milanovic and M. E. Zaghloul, "Synchronization of chaotic nueral networks and applications to communications," *International Journal of Bifurcation and Chaos,* vol. 6, no. 12b, pp. 2571-2585, 1996.

[30] I. Dalkiran and K. Danisman, "Artificial neural network based chaotic generator for cryptology," *Turkish Journal of Electrical Engineering and Computer Sciences,* vol. 18, 2010.