Name: Youwei Li
Matriculation number: S1923864

MEng Honours Project Phase 1 Report

Cell-search and interference cancellation in 5G cellular networks

3 August 2023

Mission Statement

Project Definition

The objective of this project is to employ reservoir computing to mitigate the impact of multi-path interference in wireless channels and address the nonlinear effects in transceiver hardware. This project will commence by conducting an in-depth study of reservoir computing. Subsequently, a MATLAB code will be developed to implement reservoir computing and simulate wireless communication scenarios. Through this simulation, our primary objective is to carefully observe and comprehensively analyze the effectiveness of reservoir computing in mitigating the challenges posed by multi-path interference and non-linearity within the wireless communication process.

Main Tasks

- Study reference[1].
- Understand reservoir computing.
- Implement MATLAB code for reservoir computing.
- Implement MATLAB code for Non-linearity.
- Implement MATLAB code for wireless communication.
- Perform MIMO simulation on MATLAB.
- record and analyze simulation results.
- Practical implementation or checking with real data.
- Compare reservoir computing with other techniques.

Background Knowledge

- MATLAB programming.
- Wireless communication-related knowledge.
- Machine learning knowledge.
- Linear algebra.

Abstract

This report examines the use of Reservoir Computing (RC) for predicting non-linearity and multi-path impact in wireless scenarios. Non-linearity is introduced through hardware imperfections causing distortion in the signal, while multi-path effects arise due to signal reflections, diffraction, and scatterings in the transmission environment. We propose a novel RC, with new introduced parameters: Memory strength and Memory node number. We apply this RC model, trained on data set that simulate real-world wireless conditions by MATLAB. The model exhibits superior performance in predicting non-linearity and multi-path impacts, surpassing traditional RC methods in accuracy. Importantly, the predicted non-linearity and multi-path effects from the RC model are subtracted from the received signal, enabling us to isolate the correct signal. This effectively mitigates the impacts of non-linear distortions and multi-path propagation, resulting in a clean, accurate representation of the original signal. The findings highlight the potential of RC in enhancing signal clarity and reliability in wireless communications.

Declaration of Originality

I declare that this thesis is my
original work except that
the MATLAB file fun_codedPayloadGen.m in Appendix C3
and the MATLAB file generate_transmitted_signal.m in Appendix C4
are done by my project supervisor Tharm Ratnarajah, and Haifeng Luo.

....

Statement of Achievement

In the current phase of our work, we have constructed an innovative reservoir computing model. Through comprehensive MATLAB programming for memory capacity tasks, NARMA10 tests, and simulations, we have met all the pre-set objectives and goals. Benchmarking results affirm the enhanced performance of our proposed reservoir computing model over reservoir computing in Ref[1].

Moreover, the simulations representing wireless communication between a single transmitter and receiver underline the potential applicability of our novel reservoir computing approach in the realm of digital communication. This innovative proposition brings a new perspective and opens fresh possibilities for advancing the efficiency and effectiveness of digital communication systems.

Contents

List of Symbols

#	Pound sign; used with the meaning “number of”.
θ	Angle between the radial line and the z-plane.

Glossary

BPTT Backpropagation Through Time.

CMC Cross Memory Capacity.

ESN Echo State Network.

ESNs Echo State Networks.

FNN Feed-Forward Neutral Network.

LMC Linear Memory Capacity.

MC Memory Capacity.

MIMO Multiple Input and Multiple Output.

MNN Memory Node number.

MS Memory Strength.

N The number of non-linear nodes.

PA Power amplifier.

PAs Power amplifiers.

PH Parallel Hammerstein.

QMC Quadratic Memory Capacity.

RC Reservoir Computing.

RCs Reservoir Computings.

RNN Recurrent Neural Network.

RNNs	Recurrent Neural Networks.
TDL	Tapped delay line.
TDRC	Time-Delay Reservoir Computing.

Chapter 1

Introduction

1.1 Project overview

1.1.1 Project background

The field of wireless communication faces significant challenges due to the inherent environmental factors affecting signal quality, most notably the impacts of multi-path propagation and hardware non-linearity effects. Multi-path propagation, caused by signal reflection, diffraction, or scattering off objects in the environment, can result in delay spread, phase shifts, and path loss, causing destructive interference and signal fading(**Figure 1.1** simply shows this phenomenon). Non-linearity effects, especially prevalent in amplifier circuits, lead to distortions like harmonic and inter-modulation distortions and spectral regrowth, which degrade the signal quality. Our project aims to explore the application of a novel computational framework, reservoir computing, to mitigate these challenges.

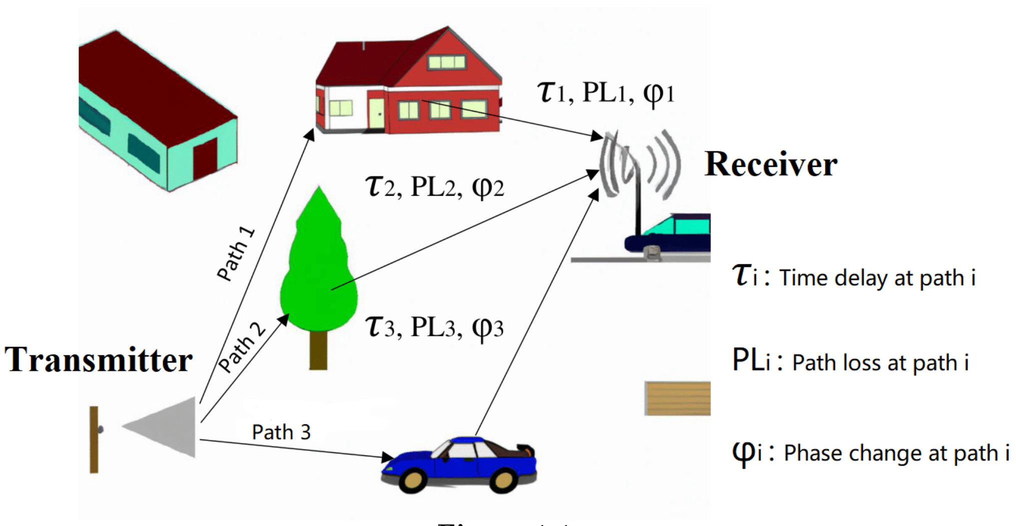
τ_1, PL1, φ_1
Receiver
τ_2, PL2, φ_2
Path 1
τ_i : Time delay at path i
τ_3, PL3, φ_3
Transmitter
Path 2
PLi : Path loss at path i
Path 3
φ_i : Phase change at path i

Figure 1.1

1.1.2 Introduction to reservoir computing

Reservoir computing is a computational framework suited for temporal/sequential data processing.[2] It is derived from several recurrent neural network models, including echo state networks and liquid state machines.[2] A reservoir computing system consists of a reservoir for mapping inputs into a high-dimensional space and a readout for pattern analysis from the high-dimensional states in the reservoir.[2] The reservoir is fixed and only the readout is trained with a simple method such as linear regression and classification.[2] Reservoir Computing (RC) brings several benefits:

1. **Efficiency:** Because the training is only performed on the output (readout) layer rather than the entire network, this makes the training process much less computationally intensive and faster.

2. **Simplicity:** Traditional recurrent neural network (RNN) training involves the difficulty of adjusting weights throughout the entire network, which can lead to complex, hard-to-optimize systems. In contrast, with RC, only the readout weights should be adjusted. This makes the optimization process much simpler since it is often a linear problem.

3. **Stability:** The reservoir in RC is randomly initialized and remains static during training, avoiding issues such as vanishing and exploding gradients that often occur in traditional RNNs. This means the training process is more stable and less prone to these specific types of problems.

Figure 1.2 gives an example of the structure of reservoir computing.

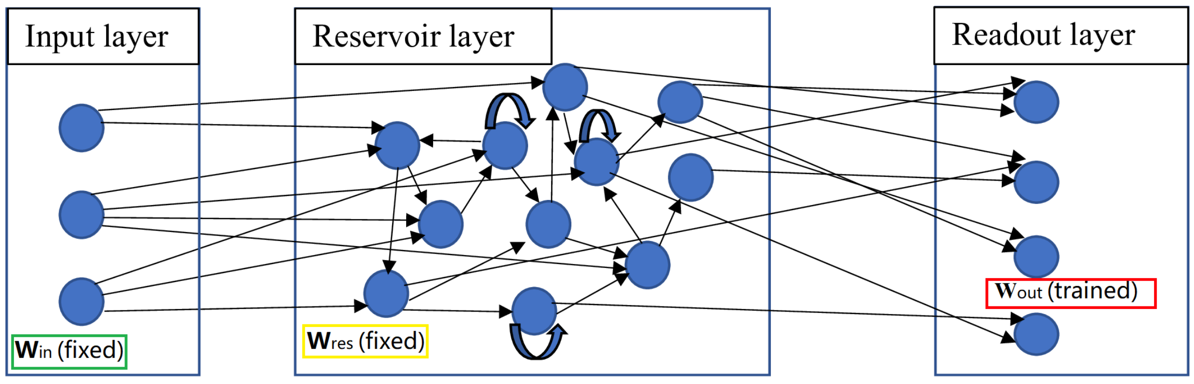

Figure 1.2

1.2 Summary

This project explores the use of reservoir computing to address key challenges in wireless communication. These challenges stem from environmental factors leading to multi-path propagation and hardware non-linearity effects, both of which degrade signal quality. Reservoir computing offers an efficient, simple, and stable approach to mapping input signals to a higher dimension, thereby simplifying the process of neural network training. By applying this computational framework, we aim to mitigate the detrimental effects of multi-path propagation and hardware non-linearity, with the goal of enhancing signal quality in wireless communication systems.

Chapter 2

Background

2.1 Fading

Fading is a key aspect of wireless communication systems. It refers to the fluctuation or decrease in signal strength over time and space. This phenomenon can have significant adverse effects on the performance and reliability of wireless communications.

Fading can occur due to several reasons:

- **Multipath Propagation:** When a signal is transmitted, it often doesn't travel along a single path from the transmitter to the receiver. Instead, it bounces off various objects in the environment (like buildings, trees, etc.) and reaches the receiver along multiple paths. Each of these paths can have different lengths and thus different travel times, causing the signals to interfere with each other at the receiver. This interference can either increase (constructive interference) or decrease (destructive interference) the signal strength, leading to fluctuations in the received signal strength.
- **Shadowing:** Sometimes, large obstacles (like buildings or hills) can block the direct communication path for the transmitter and receiver. This will significantly impair the signal's strength, a phenomenon known as shadowing.
- **Doppler Shift:** If either the transmitter or the receiver is moving, the relative motion can cause a change in the frequency of the received signal, known as the Doppler shift. This can lead to variations in signal strength over time.

The effects of fading can be quite significant. It can lead to a drop in the data rate because the receiver might need to request retransmissions if the signal strength drops too low. It can also lead to a decrease in the coverage area of a wireless network. Areas, where the signal strength is too low due to fading, might not have reliable connectivity. In a cellular network, if the signal strength falls below a certain threshold, it can cause a call to be dropped.

One of the common models used to represent fading in wireless communication is the Rayleigh fading model.

2.2 Rayleigh channel

In the multi-path environment, the received signal at any instant is the sum of L (a large number) of signals arriving via different paths. Each signal has a random phase and amplitude. The central limit theorem postulates that when a significant number of random variables, which are both independent and follow the same probability distribution, are added together, their total distribution is close to a normal (or Gaussian) distribution. Therefore, the signal's amplitude is subjected to Gaussian distribution.

The Rayleigh distribution for amplitude is given by:

$$P_R(r) = \begin{cases} \frac{r}{\sigma^2} \cdot e^{\frac{-r^2}{2\sigma^2}} & r \geq 0 \\ 0 & r < 0 \end{cases} \tag{2.1}$$

Where:

- r is signal amplitude.
- σ^2 is variance.

The phase of the received signal is distributed uniformly in the range from 0 to 2π, and it has no dependence on the amplitude.

2.3 Tapped delay line

A tapped delay line(TDL) is a model used in the digital communication field to represent the signal that undergoes a multi-path channel. It is often employed when designing and testing digital communication systems, particularly wireless systems.

The tapped delay line model represents the multi-path phenomenon by breaking the channel down into a series of discrete "taps", each of which represents a path that the signal can take. Each tap has two properties: delay (how much time the signal takes to travel along that path) and gain (how much the signal is attenuated or amplified along that path). **Figure 2.1** simply shows TDL model.

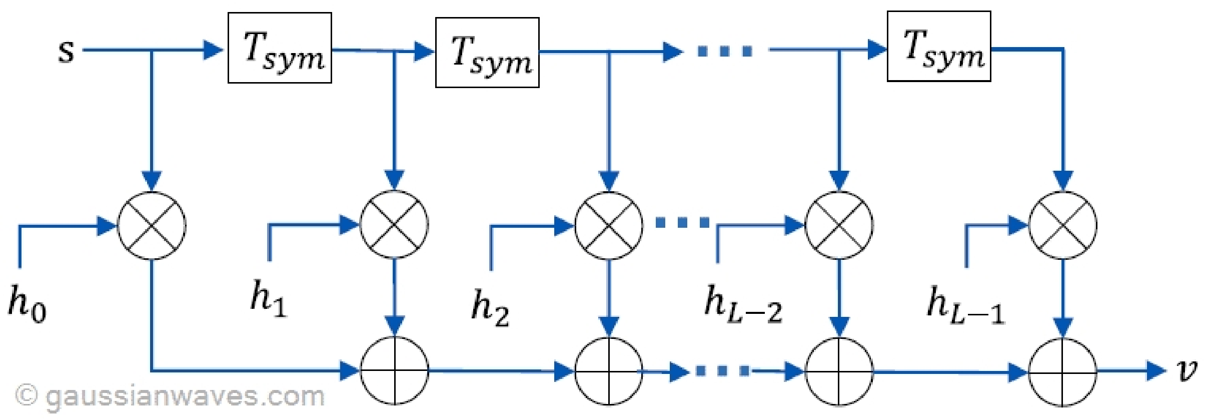

Figure 2.1: TDL model[3]

These taps are spaced at regular intervals. Channel's maximum excess delay determines the spacing and the maximum excess delay refers to the time gap between the arrival of the first signal and the final signal that surpasses a specified threshold.

The signal at the output of the tapped delay line is a summation of the signals at each tap. Each signal is delayed and scaled by the corresponding tap delay and gain, respectively. This output signal represents the signal that is received at the receiver, incorporating the effects of multipath propagation. The mathematics expression for TDL is given by:

$$y(t) = \sum_{i=0}^{N} h_i \cdot x(t - \tau_i) \tag{2.2}$$

where:

- $h_i$is the complex gain of the i_{th} tap.
- τ_i is the delay of the i_{th} tap.
- N is the number of taps

The tapped delay line model is a simplification of the actual channel behavior. Real-world channels have a continuous distribution of delays, not discrete taps. However, the tapped delay line model is a useful tool because it simplifies the complexity of multipath channels to a manageable level. One must remember that the model is just an approximation of real-world behavior and may not perfectly represent all aspects of a given channel. There are some common TDL models used for simulation, which are TDL-A, TDL-B, TDL-C, TDL-D, and so on.

2.4 Parallel Hammerstein Model

Power amplifiers(PAs) often introduce distortion due to their inherent nonlinear characteristics, particularly when they are operated near their saturation levels to maximize power efficiency. This is where the Parallel Hammerstein (PH) model can come in handy. The PH model, which is a parallel structure of multiple Hammerstein models, is well suited for modeling the memory effects and nonlinear behavior of PAs.

The Hammerstein model consists of a sequence that begins with a static nonlinear system and is then followed by a linear dynamic system. The static nonlinearity is used to capture the nonlinear behavior of the PA, while the linear dynamic system is used to capture the memory effects.

By arranging several of these Hammerstein models in parallel, the PH model can more accurately capture the complex nonlinear behavior of the PA. Each branch in the PH model can represent a different aspect of the PA's nonlinear characteristics.

By modeling and understanding the non-linearity introduced by the power amplifier using a PH model, we can design a digital predistorter. A predistorter is a device that introduces distortion that is the inverse of the PA's distortion, effectively canceling out the PA's nonlinear effects and thereby improving the overall performance of the communication system. Ref[4] has shown PH model's superior performance in mitigating PA's impairments. The mathematical expression for PH model is given by:

$$X_{PA}(n) = \sum_{\substack{p=1 \\ p\ odd}}^{P} \sum_{m=0}^{K-1} h_p(m)\Psi_p(X_{tx}(n-m)) \tag{2.3}$$

where:

- P highest order of non-linearity considered.
- K-1 highest memory length considered.
- $\Psi_p(x)$ is p_{th} order basis non-linear function and $\Psi_p(x) = |x|^{p-1} \cdot x$.
- X_{PA} is PA output at time sequence n.
- $X_{tx}(n-m)$ is m_{th} preceding memory of transmitted signal.

Chapter 3

Literature review

The structure known as RC, mentioned in Reference [1], will be referred to as Time-Delay Reservoir Computing (TDRC) in this chapter. This chapter will explain where the TDRC idea comes from (its origin), describe the structure of TDRC, explain how TDRC works (the underlying principle), introduce the use of technology grid search to optimize hyperparameters for TDRC, and also provide some references for basic knowledge of machine learning.

3.1 Origin of TDRC

TDRC has emerged as an impactful adaptation of conventional Reservoir Computing models, such as Echo State Networks(ESNs). This development simplifies the structure of ESNs and also saves the computation cost of ESNs. It's important to note that these traditional reservoir computing models are derived from Recurrent Neural Networks (RNNs).

3.1.1 Recurrent Neural Networks

Recurrent Neural Network (RNN) is a unique category of artificial neural networks(ANN) designed specifically for dealing with time-sequential data. In contrast to feedforward neural networks(also known as artificial neural networks), RNNs have a form of 'memory,' which is represented by hidden state vectors. These vectors capture information from previous inputs and use it to influence the processing of current and future inputs.

In more precise terms, an RNN computes the hidden state at time t, denoted as h_t, by using both the input vector at time t, denoted as x_t, and preceding hidden state, h_{t-1}. This operation is expressed by the following equation:

$$h_t = \sigma(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \tag{3.1}$$

The network then computes the output y_t based on the hidden state h_t:

$$y_t = \sigma(W_{hy}h_t + b_y) \tag{3.2}$$

In these equations, W_{hh}, W_{xh}, and W_{hy} are the weight matrices that the network learns during training. The vectors b_h and b_y are bias terms, also learned during training. The function σ is a non-linear activation function, commonly tanh or ReLU in the context of RNNs.

Figure 3.1 shows the structure of RNN and Feed-forward Neutral Network. The main difference is that each hidden state in RNN receives feedback from the previous state of itself, which enable each hidden state to memorize the previous state. Therefore, RNNs can learn temporal dependence. However, Feed-Forward Neutral Network(FNN) does not have any feedback, which means FNN is not capable of memorizing previous states.

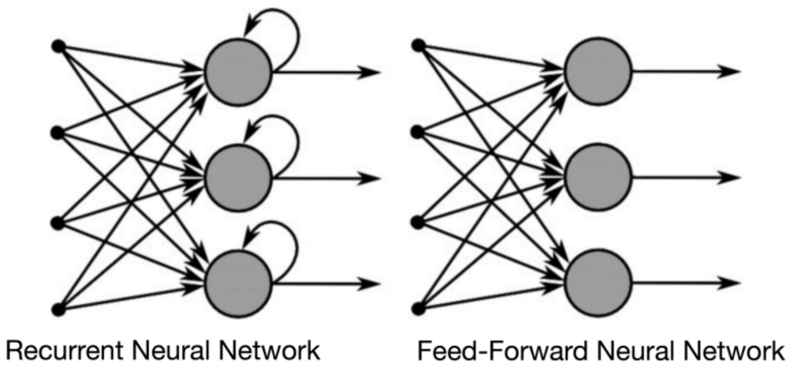

Figure 3.1: RNN vs FNN[5]

Training Process

The training of RNNs employs a technique known as backpropagation through time (BPTT). To apply BPTT, we first 'unroll' the network through time, which means treating each time step as a separate layer in a deep neural network. Then, we apply the standard backpropagation algorithm to compute gradients and update the weights. The adjustments for weights are related to the loss function.

The loss function L, which quantifies the discrepancy between the neutral network's predicted output and the true output, is defined as the sum of the losses at each time step for a sequence of length T:

$$L = \sum_{t=1}^{T} L_t(y_t, \hat{y}_t) \tag{3.3}$$

In BPTT, the objective involves calculating the gradients of the loss concerning the weights, followed by utilizing these gradients to modify the weights. The gradients are calculated using the chain rule of calculus:

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial W} \tag{3.4}$$

Once the gradients are calculated, the weights are updated using a method like stochastic gradient descent:

$$W_{t+1} = W_t - \alpha \frac{\partial L}{\partial W} \tag{3.5}$$

- W_{t+1} : Updated W_t.
- α : Learning rate.
- W_t : Weights after t updates.

However, BPTT has a couple of challenges. The first is the problem of "exploding gradients," where the gradients become too large and cause the learning process to become unstable. This can be mitigated by applying a technique known as "gradient clipping," where the gradients are rescaled if they exceed a certain threshold.

The second challenge is the problem of "vanishing gradients," where the gradients become too small and the parameters of the network stop learning. This is a more difficult problem to address and it is one of the reasons why other types of RNNs, such as ESNs and Long Short-Term Memory (LSTM) networks, are developed.

Applications

RNNs have wide-ranging applications across numerous domains. In natural language processing, they are used for tasks like sentiment analysis, text generation, and machine translation. In speech recognition, RNNs help in transcribing spoken language into written text. They are also used for time-series prediction tasks, such as stock price prediction or weather forecasting.

3.1.2 Echo State Network

ESNs represent a category of RNN, employing a concept known as Reservoir Computing. In an ESN, the recurrent layer, often referred to as the reservoir, is initialized randomly and remains untrained, while solely the readout layer, or output layer, is subjected to training.

The recurrent layer is typically large and sparse, and its purpose is to provide a high-dimensional, non-linear transformation of the inputs. The connection between reservoir states are randomized and when the corresponding weight is set to 0, 2 reservoir states are disconnected, and vice versa. Due to the recurrent nature of this layer, it has a "memory" and can therefore process temporal sequences.

The new reservoir state x(n+1) is determined by the present input u(n+1), x(n), and the preceding output y(n). Each of these terms is multiplied by their respective weight matrices (W^{in}, W^{res}, W^{back}), and the results are summed and passed through an activation function $f(\cdot)$(which is also called non-linear function), typically the hyperbolic tangent function.

ESN architecture is shown as follows:

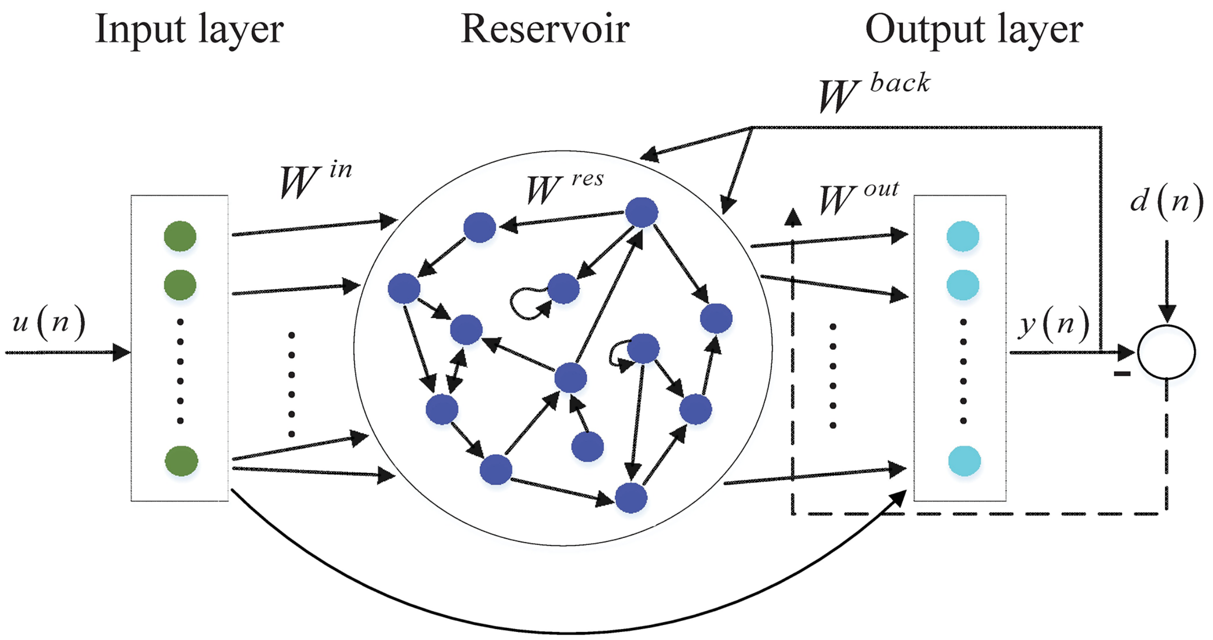

Figure 3.2: ESN architecture[6]

Reservoir state's update is given by:

$$x(n+1) = f(W^{in}u(n+1) + W^{res}x(n) + W^{back}y(n) + v(n)) \tag{3.6}$$

The readout layer is given by:

$$y(n+1) = f^{out}(W^{out}[u(n+1)|x(n+1)]) \tag{3.7}$$

where f^{out} denotes the activation function of the output neuron, [u(n + 1)|x(n + 1)] denotes the concatenation vectors of the input and internal activation vectors and W^{out} is the output weight matrix that has been trained.[6] v(n) represents noise. Equation(3.6) and equation(3.7) are from Ref[6].

Training process

In contrast to RNN, ESN solely trains the weights corresponding to the outputs. The weights for both inputs and reservoir are initialized randomly and remain fixed during training. The purpose of training is to find optimum weights for yielding the desired output. Linear regression is a common method for training ESN, which can be solved in a straightforward and computationally efficient manner. Because only the output weights are trained, and they are trained using linear regression rather than gradient-based optimization, ESNs avoid the issues with vanishing and exploding gradients that plague traditional RNNs. This makes ESN training simpler and more efficient than RNN training, especially for long sequences.

Linear regression is given by:

$$W^{out} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \tag{3.8}$$

Where X is all training data and Y is all target data. The detailed derivation for equation(3.8) is shown in **Appendix A**.

3.1.3 Time-Delay Reservoir Computing

It has been demonstrated in Ref[7], [8] that even a simplistic interconnection structure of the reservoir layer can yield outstanding results. Therefore, a simple architecture can be developed, where each reservoir state x_i is solely reliant on the state of its neighboring node x_{i-1} and masked current input.

In TDRC, a pre-processing step is carried out before the input reaches the reservoir nodes. This procedure, referred to as "masking," elevates the input to a higher dimensional space through multiplication by a mask array. Take for instance a mask array [0.2, 0.8, -0.5, 1, -1, -0.01, 0.3]. Given an input value of 2 at time n, the resulting masked input would be [0.4, 1.6, -1, 2, -2, -0.02, 0.6], which is a scalar multiplication between the input and the mask array. In this instance, the input's dimension transitions from 1×1 to 1×7. The mask array, which is randomly generated, remains unvaried across all inputs. The size of this array must match the number of reservoir nodes, ensuring each node receives a masked input. An illustrative example of a masking process involving five inputs with a mask array size of four is depicted in **Figure 3.3**.

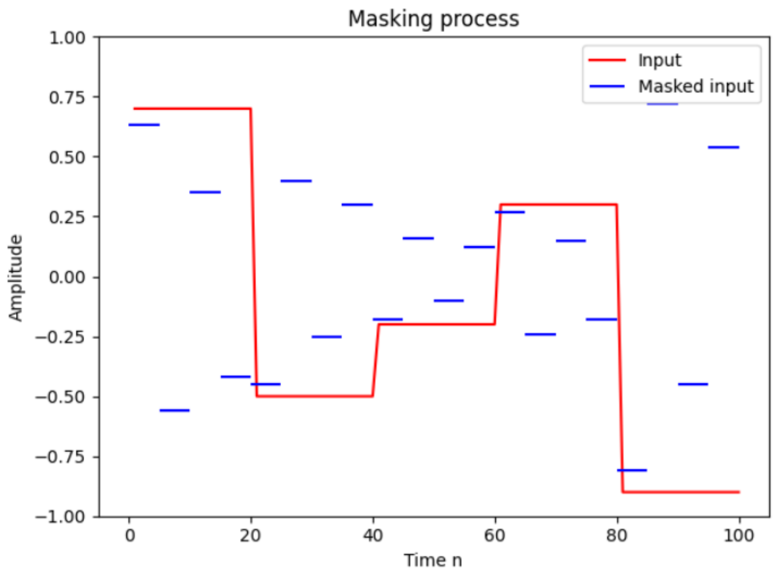

Figure 3.3: Masking

Post the masking phase, the reservoir layer undergoes an update. In order to maintain memory, delayed feedback is needed. The update process of the reservoir state can be described as follows:

$$x_i(n) = \begin{cases} F_{NL}(\alpha \cdot x_{i-1}(n-1) + \beta \cdot m_i \cdot u(n)) + v_i(n) & 2 \leq i \leq N \\ F_{NL}(\alpha \cdot x_{N+i-1}(n-2) + \beta \cdot m_i \cdot u(n)) + v_i(n) & i = 1 \end{cases} \tag{3.9}$$

Equation(3.9) is from Ref[1]. N represents the number of reservoir nodes, while $X_i(n)$ is the state of the i_{th} reservoir at time n. m_i is corresponding mask value, $u(n)$ denotes the input at time n and $v_i(n)$ is noise. The variables α and β represent the feedback gain and the input gain respectively. F_{NL} refers to a non-linear function. The sum of input gain and feedback gain should not exceed 1, otherwise, the reservoir will be unstable because the reservoir states will keep increasing as time go through and may reach infinity. **Figure 4.5** gives an example of TDRC with N=4 and because its structure looks like a delay loop, it is called Time Delay Reservoir Computing.

Finally, at the readout layer, output weight matrices multiply all reservoir states to get output. The readout layer is given by:

$$y(n) = W^{out}X(n) \tag{3.10}$$

X(n) is all reservoir states at time sequence n. The training process is the same as Echo State Networks except that only reservoir states are trained.

In summary, TDRC simplifies Echo State Networks, while remaining performance. The computational and training cost is greatly reduced and It is easier to practically implement.

3.2 Principle behind TDRC

The fundamental principle of TDRC involves approximating intricate temporal dependencies by using linear algebraic operations on a large number of different temporal dependencies. As an example, let us consider the approximation of the following temporal dependency:

$$u(t)^2 + u(t-2) \cdot u(t-3)^3 - u(t-4)^4 \tag{3.11}$$

Assume that we have a vast assortment of different temporal dependencies, each more complex than the one illustrated above. A sample of these dependencies is expressed by the following equations:

$$\begin{aligned}
0.2u(t) - 0.3u(t)^2 + 0.1u(t-1)u(t-2)^2 - 0.54u(t-4)^2 + 0.23u(t-4)^4 + 0.91u(t-2)u(t-3)^3 \quad &\text{(E1)}\\
u(t) + 0.6u(t)^2 + 0.5u(t-1)u(t-2)^2 + 0.1u(t-4)^2 - 0.3u(t-4)^4 + 0.24u(t-2)u(t-3)^3 \quad &\text{(E2)}\\
-0.3u(t) - 0.8u(t)^2 - 0.32u(t-1)u(t-2)^2 - 0.54u(t-4)^2 + 0.1u(t-4)^4 - 0.94u(t-2)u(t-3)^3 \quad &\text{(E3)}\\
\vdots \quad &\text{(Ei)}\\
-u(t) + 0.24u(t)^2 + 0.2u(t-1)u(t-2)^2 - 0.14u(t-4)^2 - 0.91u(t-4)^4 + 0.34u(t-2)u(t-3)^3 \quad &\text{(En)}
\end{aligned}$$

Given a sufficient number of complex temporal dependencies, Equation (3.11) can be accurately approximated through a series of additions and subtractions amongst these equations (a_1·E1 + a_2·E2 - a_3·E3 + ... - a_n·En). Although some residual terms may persist, they are usually negligible and have minimal impact on the approximation. Below is an example of such an approximation:

$$0.0007u(t) + 0.998u(t)^2 + 0.0005u(t-1)u(t-2)^2 + 0.00015u(t-4)^2 - 0.999u(t-4)^4 + 0.994u(t-2)u(t-3)^3 \tag{3.12}$$

The approximation relies significantly on the richness of dynamics in the reservoir layer. If the dynamics of the Reservoir layer are poor, the temporal dependencies involved are relatively simple, having only two terms—for instance, $a \cdot u(t)^2 u(t-2)$ and $b \cdot u(t-1)u(t-2)$. In this case, it becomes infeasible to approximate complex equations like (3.11).

Factors such as masking, non-linear function, and delayed feedback significantly influence the complexity of dynamics in TDRC. Subsequent subsections will elucidate how these elements contribute to enhancing the richness of dynamics.

3.2.1 Effect of masking

Masking increases the diversity of reservoir node states by multiplying each input with a vector of the random coefficient to get many different masked inputs. Consider a scenario where we have a mask array consisting of the elements [-0.6, 0.7, 0.1, -0.9]. In this context, we also utilize a basic non-linear function denoted as x^2. We further assume that α and β are set at 0.5 and 0.3 respectively. Based on these parameters, and according to equation (3.9), we can determine all the states within the reservoir as follows:

$$\begin{cases} X_1(n) = 0.25X_4(n-1)^2 - 0.18 \cdot X_4(n-1) \cdot u(n) + 0.0324u(n)^2 \\ X_2(n) = 0.25X_1(n-1)^2 + 0.21 \cdot X_1(n-1) \cdot u(n) + 0.0441u(n)^2 \\ X_3(n) = 0.25X_2(n-1)^2 + 0.03 \cdot X_2(n-1) \cdot u(n) + 0.0009u(n)^2 \\ X_4(n) = 0.25X_3(n-1)^2 - 0.27 \cdot X_3(n-1) \cdot u(n) + 0.0729u(n)^2 \end{cases} \tag{3.12}$$

Each of the preceding equations is distinct, facilitating a more straightforward approximation process. Conversely, if all the equations were identical, an approximation would become unfeasible. This is because any operations of addition or subtraction would merely yield a scaled or amplified form of the original equation, without generating a unique result. Consider a scenario where we aim to approximate the expression $u(n)u(n-1)^2$. We are provided with five identical equations of the form $u(n)^2 + u(n)u(n-1) - u(n)u(n-1)^2 + u(n-2)^2$. The central issue arises from the unique time-step dependency in the term $u(n)u(n-1)^2$, which contrasts with the terms in the proposed approximating equations. Due to this, the method of repetitively adding or subtracting the same equation fails to yield an accurate approximation of the original expression. This is primarily because these approximating equations do not capture the intricate dependencies between the different time steps. For a more precise approximation, it's necessary to employ a diverse set of equations, ones capable of accurately capturing the relationship embedded in the original expression. Relying on identical equations for approximation leads to a simple linear combination, which does not offer the requisite diversity to model the intricate interactions between $u(n)$ and $u(n-1)$ that are present in the original expression.

Hence, the effect of masking is to increase the diversity of reservoir states and provide a basis for better approximation. Without masking, the approximation becomes infeasible.

3.2.2 Effect of non-linear function

The non-linear function operates on two inputs: the memory trace of the previous state from an adjacent node and the current masked input. This function generates outputs with intricate temporal dependencies, the complexity of which is governed by the nature of the non-linear function. As an illustration, let's consider an instance where the preceding state of the node, $X_1(n-1)$, is given by $0.25u(n-2)^2 - 0.4u(n-1)u(n-2) + 0.16u(n-1)^2$, the masked input is 0.8u(n), the non-linear function is x^2, and the parameters α and β are assigned values of 0.5 and 0.3 respectively. Under these conditions, the state of the node, $X_2(n)$, will be calculated as follows:

$$\begin{aligned}X_2(n) = &\ 0.015625u(n-2)^4 - 0.05u(n-2)^3u(n-1) + 0.008u(n-2)^2u(n-1)^2\\ &- 0.01u(n-2)^2u(n-1)^2 + 0.0032u(n-2)u(n-1)^3 - 0.00064u(n-1)^4\\ &- 0.03u(n-2)^2u(n) + 0.0096u(n-2)u(n-1)u(n) - 0.00192u(n-1)^2u(n)\\ &+ 0.0576u(n)^2\end{aligned}$$

Upon inspection, it's evident that $X_2(n)$ is characterized by numerous complex temporal dependencies. This suggests that TDRC can capture and approximate all the temporal dependencies present in $X_2(n)$, provided there's adequate diversity among the reservoir nodes. We have also explored alternatives to the quadratic function, x^2, specifically, in trigonometric functions. The appeal of trigonometric functions lies in their ability to be expanded via the Taylor series, thereby incorporating a non-linearity into our approximations. The Taylor series expansions for the cosine, $\cos(x)$, and sine, $\sin(x)$, functions are presented below:

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \tag{3.13}$$

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \tag{3.14}$$

Equation (3.13) and (3.14) shows high non-linearity of trigonometric functions, which means the trigonometric function can provide much more complex temporal dependencies than a simple quadratic function.

3.2.3 Effect of delayed feedback

Delayed feedback plays a crucial role in maintaining a record of past states. Without delayed feedback, dependencies would only exist in relation to the current time, and temporal dependencies would not be preserved. To illustrate this, let's consider an attempt to approximate $u(n-1)u(n-2)u(n)^2 - u(n)u(n-3)$ in the absence of delayed feedback. The state of each reservoir node would take the form $a_1u(n) + a_2u(n)^2 + a_3u(n)^3 + ... + a_ku(n)^k$. Under these conditions, reconstructing $u(n-1)u(n-2)u(n)^2 - u(n)u(n-3)$ would be impossible, as the reservoir nodes lack any information concerning previous inputs.

With the progression of time, it becomes necessary to gradually forget certain past memories. Given that TDRC possesses a finite memory capacity, it cannot retain all past information indefinitely. This memory-decay process is governed by the parameter β. As per equation (3.9), each node accepts delayed feedback multiplied by β. Since β is less than 1, the influence of previous memories steadily decays and approaches zero over time. For instance, if β is set to 0.3 and the delayed feedback is $X_1(n)$, the influence of this feedback would be scaled down to at least $3.49 \cdot 10^{-11}X_1(n)$ after 20-time sequences, essentially rendering it negligible. However, what if a particular task necessitates a greater memory capacity than what TDRC can offer? Elevating β to 1 would enable the retention of all past memories, but this would inevitably lead to overflow, as node states would continually increase towards infinity over time. To mitigate this issue, a novel method has been devised in this project to adjust the memory capacity in accordance with the task's requirements. This method will be elaborated upon in Chapter 4, titled 'Methodology'.

3.3 Grid search

In machine learning, building a good predictive model isn't just about selecting the right algorithm or variables. We also have to tune hyperparameters, which are parameters not learned from the data but set prior to the learning process. Determining the optimal values for these hyperparameters can be tricky. To handle this challenge, we use a technique called grid search.

Grid search is a traditional method used to tune the hyperparameters of a model. It works by searching exhaustively through a predefined set of hyperparameters. Below are the steps involved in the grid search process:

1. **Define a set of hyperparameters:** This step involves manually specifying a subset of the model's hyperparameter space that will be searched.
2. **Set up the parameter grid:** The grid consists of different combinations of hyperparameters. For instance, if we have two hyperparameters, each with two possible values, our grid will have four possible combinations of hyperparameters to try out.
3. **Choose a suitable performance metric:** The performance of a model for each combination of hyperparameters is assessed using a chosen performance metric. For classification problems, this could be accuracy, precision, recall, or F1 score, and common metrics for regression are mean squared error(MSE), mean absolute error(MAE), and R-squared(R^2).
4. **Apply K-fold Cross-Validation (CV) for each combination:** Every combination of parameters is employed for model training, and the corresponding performance is assessed through K-fold cross-validation (CV). The training dataset gets partitioned into K distinct segments or folds. The training of the model utilizes K-1 folds, while the leftover fold serves the validation purpose. This procedure is executed K times, designating each fold as the validation set once. The performance metric is derived from the mean of the values tallied throughout the iterations.
5. **Find the best hyperparameters:** Upon evaluating all possible combinations, the option delivering the optimal performance, as judged by the selected metric, is chosen.

Grid search is a straightforward and effective method for hyperparameter tuning, offering a systematic approach to exploring the possible configurations of a model. However, it can be computationally demanding, especially when dealing with a large number of different hyperparameters. **Appendix B** shows the setting of the grid search for this project.

3.4 References for basic knowledge of machine learning

This section provides references for some basic knowledge of machine learning to help better understand reservoir computing. The references are listed below:

- Artificial neural networks: Ref[9], Ref[10], Ref[11].
- Regression: Ref[12], Ref[13], Ref[14].
- Recurrent neutral networks: Ref[15], Ref[16], Ref[17].
- Echo state network: Ref[18], Ref[19].
- Reservoir computing: Ref[20], Ref[21].

Chapter 4

Methodology

The structure of reservoir computing used in this report is based on Ref [1] and some changes are made here to improve the performance. The architecture of reservoir computing used in this report will be presented by three layers(input layer, reservoir layer, and readout layer), and the pseudocode of RC will also be presented at the end of this Chapter.

4.1 Input layer

The input signal, denoted as u(n) remains steady for a duration, T. This duration, T, is divided into N sub-intervals, each of a length θ. Each interval has a corresponding mask value, represented as m_i. Then using the input signal u(n) multiply each m_i to get N masked input signals(m_iu(n)). Because in this project, we are dealing with the complex number input signal, each mask value is set to be a complex number rather than a real number to enrich the dynamics of reservoir computing. In this project, the mask value is randomly picked from uniform distribution and the amplitude of the mask value is set to be less or equal to 1. Therefore, both real part and imaginary part of each mask value are randomly chosen from uniform distribution $U(\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$, which is represented as $\Re(m_i) \sim U(\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$, $\Im(m_i) \sim U(\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$. Here we use a mask array m, where m is $[m_1, m_2, ..., m_N]$ with length N, to store all mask values. Therefore, all masked input signals can be represented as scalar multiplication between input signal u(n) and mask array m, which is $u(n) \cdot m$. In summary, the primary function of the input layer is to mask the input.

4.2 Reservoir layer

To achieve useful computation on time dependent input signals, a good reservoir should be able to compute a large number of different functions of its inputs.[22] That is, the reservoir should be sufficiently high-dimensional and its responses should not only depend on present inputs but also on inputs up to some finite time in the past.[22] To achieve this, the reservoir should have some degree of nonlinearity in its dynamics and a “fading memory”, meaning that it will gradually forget previous inputs as new inputs come in.[22] The demands for memory fading vary across different tasks; while some necessitate a high degree of memory fading, others might only require a minimal level. Therefore, we need to tune the memory fading to best fit the task. In this design, there are 2 important parameters to adjust the degree of fading memory, which are Memory node number(**MNN**) and Memory strength(**MS**), respectively.

4.2.1 Memory node number

MNN determines how many nodes have memory. The node with memory means this node can memory n previous states of itself. Such a node needs to be able to retain at least two of its prior states because When node1 connects to node2, node2 will receive instant delayed feedback from node1 with information of 1 preceding memory of node1. In this case, node1 is only capable of maintaining its current state for a short duration and node1 doesn't possess storage for its prior states. This design incorporates two types of nodes. The first type is a non-linear node, which produces an output based on applying a non-linear function to the sum of all its inputs. The expression for the non-linear node is $X_i = F_{NL}(\sum_{k=1}^{n} I_k)$, where X_i is the state of a non-linear node and F_{NL} is non-linear function and I_k is one of input and n is total input. The other node is the delay node, which only stores the previous state of a specific node. The expression for the delay node is $DN_i(t) = X_i(t-n), n \geq 1$, where $DN_i(t)$ is the state of delay node DN_i at time sequence t and $X_i(t-n)$ is the state of non-linear node X_i at n previous time sequence(s) to t. To confer memory upon a non-linear node, it is linked in series with one or more delay nodes. These delay nodes essentially serve as shift registers, providing a temporary repository for previous states. **Figure 4.1** shows the combination of Non-linear nodes and delay nodes.

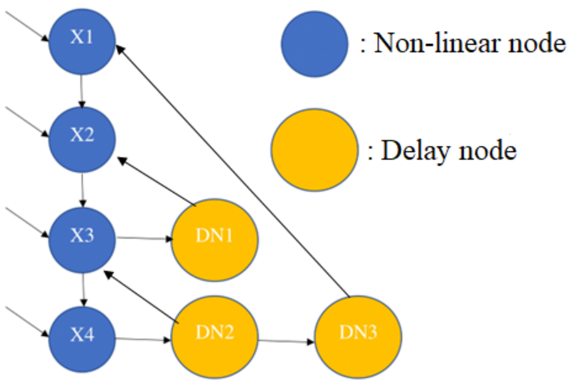

Figure 4.1

In **Figure 4.1**, DN1 retain the state X3(t-1) and DN1 enable X2 to receive the state X3(t-2); DN2 retain the state X4(t-1) and DN2 enable X3 to receive the state X4(t-2); DN3 retain the state X4(t-2) and DN3 enable X1 to receive the state X4(t-3). The MNN for this figure is 2 because only X3 and X4 have memory capability.

4.2.2 Memory strength

Memory strength determines how fast to forget past input. Higher MS indicates a slower speed of forgetting the past input, and vice versa. Because reservoir computing will gradually forget past input, the information about the previous state need to be feedback to the current state to enhance memory. Certain tasks necessitate robust memory capabilities, whereby a large number of preceding states must feedback to the current state. Conversely, other tasks may require a weaker memory, where only a few past states are needed to provide feedback to the current state. Memory strength is the difference between the current time sequence and the time sequence of the earliest memory. For example, if the earliest memory stored in the delay node is $X_i(t-n)$, the memory strength is n. **Figure 4.2** shows an example of reservoir with MNN=2 and MS=3.

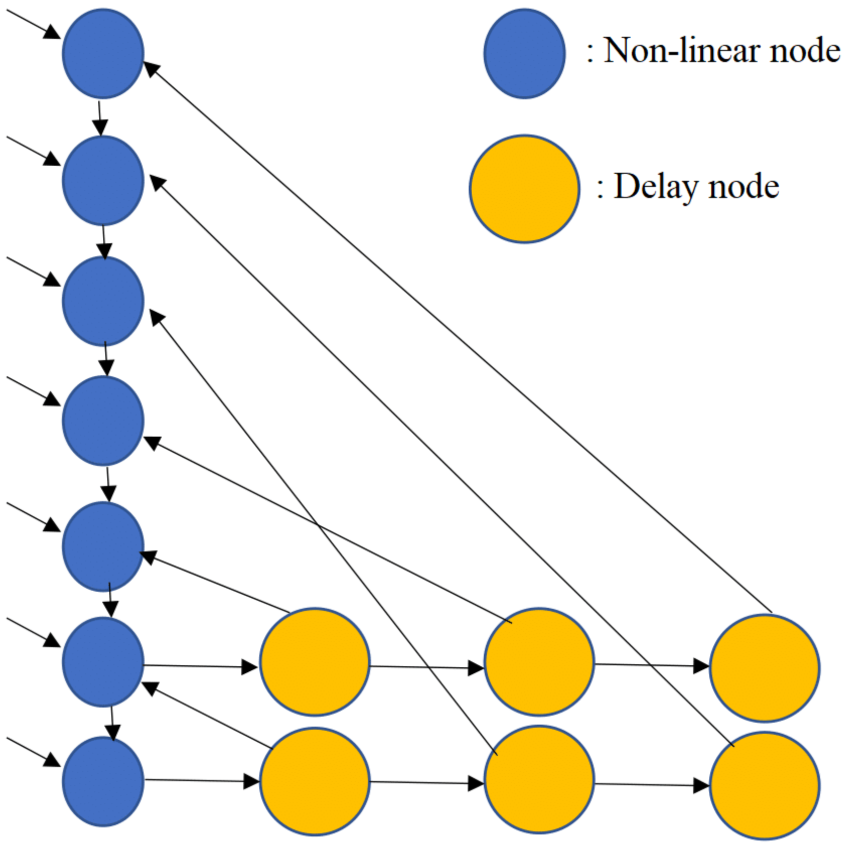

Figure 4.2

4.2.3 numerical model

A numerical model has been developed for the architecture of the reservoir. Equation (4.1) shows the expression for all non-linear node states.

$$X_i(n) = \begin{cases} F_{NL}(\alpha \cdot X_{i-MS\cdot MNN}(n-1) + \beta \cdot m_i \cdot u(n)), & MS \cdot MNN < i \leq N \\ F_{NL}(\alpha \cdot X_{N-(MNN-((i-1)\%MNN)-1)}(n-2-\lfloor \frac{i-1}{MNN} \rfloor) + \beta \cdot m_i \cdot u(n)), & 1 \leq i \leq MS \cdot MNN \end{cases} \tag{4.1}$$

F_{NL} is a non-linear function. F_{NL} can be tuned to adapt to the non-linearity requirement of the task. N is the number of non-linear nodes. The development of equation (4.1) is based on Equation (5) in Ref [22]. The main difference is that equation (4.1) brings the parameter Memory strength, which can tune the fading memory. If MS is set to 1, equation (4.1) is the same as equation (5) of Ref[22] except for the absence of phase term. If both MS and MNN are set to 1, equation (4.1) is the same as equation (4) in Ref [1], except that there is noise term $v_i(n)$ in equation (4) of Ref[1]. **Figure 4.3** shows the numerical model (4.1) with N = 10, MS = 2 and MNN =2; **Figure 4.4** shows the numerical model (4.1) with N = 4, MS = 1 and MNN =2; **Figure 4.5** shows the numerical model (4.1) with N = 4, MS = 1 and MNN =1.

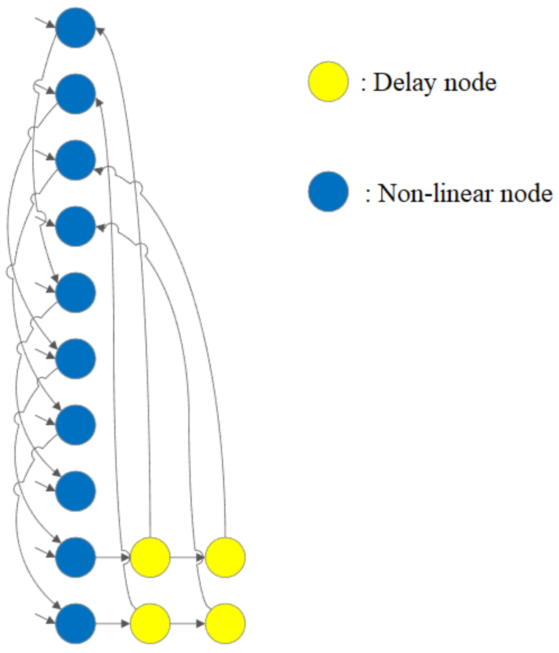

Figure 4.3

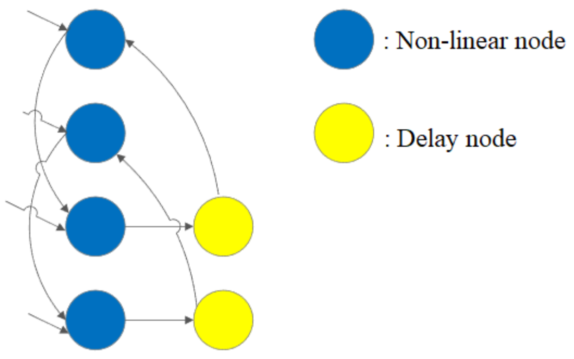

Figure 4.4

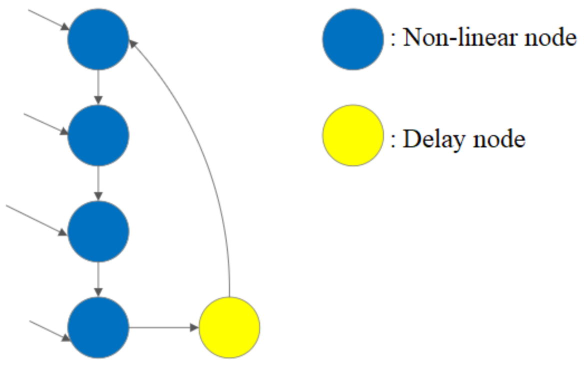

Figure 4.5

4.3 Readout layer

The readout layer is the final layer of Reservoir computing and plays a crucial role in generating the network's output. Let x(n) be a vector with dimension N $\times$ 1, which stores all non-linear nodes' states at time sequence n. The readout layer will take x(n) as input and output y(n) with dimension 1 $\times$ 1. The general formula used for the readout layer's operation is:

$$y(n) = W_{out} \times x(n) \tag{4.2}$$

W_{out} represents the output weights connecting the reservoir to the readout layer with the dimension of 1 $\times$ N. W_{out} is initially unknown, so training is needed to find the optimum W_{out}. To perform training, a bunch of time-sequential data needs to be collected. The data used to train are all non-linear node states in the reservoir layer at a range of time series, represented as X. Equation (4.3) shows the training data between time sequence t_0 and t_n.

$$X = [x(t_0)^\top \;\; x(t_0+1)^\top \;\; x(t_0+2)^\top \;\; x(t_0+3)^\top \;\; ... \;\; x(t_n)^\top]^\top \tag{4.3}$$

Training also needs target data. This target data, denoted as Y, spans the time sequence from t_0 to t_n, as shown in equation (4.4).

$$Y = [y(t_0) \;\; y(t_0+1) \;\; y(t_0+2) \;\; y(t_0+3) \;\; ... \;\; y(t_n)]^\top \tag{4.4}$$

Upon acquiring the training data, we employ ridge regression to execute the training process. Equation (4.5) shows this process.

$$W_{out} = (X^T X + \lambda I)^{-1} X^T Y \tag{4.5}$$

- λ: The ridge penalty (also known as regularization). A larger value of λ increases the amount of regularization and helps to prevent overfitting but decreases the performance.
- I: The identity matrix of appropriate dimensions (in this case, it would be the number of features by the number of features).
- $(X^T X + \lambda I)^{-1}$: This is the inverse of the matrix resulting from the addition of the product of X^T and X and the product of λ and I.

4.4 Pseudocode

Algorithm 1: Reservoir Class

```
/* Define Properties */
1 node_num, $F_{NL}$, $\alpha$, $\beta$, M (mask array), W, MS, MNN
/* Abbreviations */
2 Delay_nodes abbreviated as DN, node_states abbreviated as NS
/* Constructor */
3 Function Initialize(node_num, $F_{NL}$, $\alpha$, $\beta$, MS, MNN):
4     Set node_num, $F_{NL}$, $\alpha$, $\beta$, MS, MNN
5     Set M to random uniform values
6     Initialize W to zeros
7 end
/* Transform */
8 Function transform(input X):
9     Initialize NS, previous_states, DN with appropriate sizes
10    for each element i in X do
11        for each node j do
12            if node j has delay nodes feedback then
13                NS[i, j] = $F_{NL}$($\alpha\cdot$ DN[((j-1) % MNN)+1,((j-1) / MNN)+1] +$\beta\cdot$ M[j] $\cdot$ X[i])
14                if node j is within the first layer then
15                    Update the first layer of DN with the last MNN nodes from previous_states
16                end
17            else
18                NS[i, j] = $F_{NL}$($\alpha\cdot$ previous_states[j-MS$\cdot$MNN] +$\beta\cdot$ M[j] $\cdot$ X[i])
19            end
20        end
21        Update previous_states array with the current state of Non-linear nodes
22        for each layer L from last to second do
23            for each node m do
24                Shift state from the (L-1)th layer to the Lth layer in DN
25            end
26        end
27    end
28    Return NS
29 end
/* Fit */
30 Function fit(input X, output y, ridge regression penalty lambda):
31    if lambda not provided then
32        Set lambda to 0
33    end
34    Transform X to obtain NS (Node States)
35    Perform Ridge Regression on NS to obtain weights W
36 end
/* Predict */
37 Function predict(input X):
38    Transform X to obtain NS
39    Compute predicted output y by multiplying NS with weights W
40    Return predicted output y
41 end
```

Chapter 5

Experiment

5.1 Experiment Setup

For this experiment, we employed a 5G NR FR2 (New Radio Frequency Range 2) system to generate and transmit signals. All signal data used for simulation and benchmarks are generated by the function *generate_transmitted_signal*(). The setup focused on signal generation using the function *generate_transmitted_signal*(). The function incorporated various hard-coded parameters to produce signals, including those related to the transmission, subcarriers, OFDM symbols, bandwidth, subcarrier spacing, and power of the base station and user equipment. The parameters used for the transmission are as follows:

- Number of Transmit Antennas (numTxAnt): We used a single transmit antenna for our setup.
- Number of Receive Antennas (numRxAnt): A single receive antenna was utilized.
- Number of Data Subcarriers (numSubcarriers): The setup used 1024 data subcarriers.
- Number of OFDM Symbols (numSymbols): We used 8 OFDM symbols for our transmission.
- Bandwidth (bandWidth): The bandwidth for our setup was 20 MHz.
- Subcarrier Spacing (carrierSpace): The subcarrier spacing was kept at 15 KHz.
- Transmit Power of the Base Station (txPowBs_dBm): The transmit power of the base station was set to 23 dBm.
- Transmit Power of the User Equipment (txPowUe_dBm): The transmit power of the user equipment was set to 0 dBm.

The *generate_transmitted_signal*() function was implemented to generate a signal that adheres to these parameters. This signal served as our basic experimental unit for the transmission. The generation of signals using the function is dependent on these parameters.

5.2 Benchmarks

There are 2 general benchmarks for reservoir computing, which are Memory capacities and NARMA10 system, respectively. These 2 benchmarks measure both memory capability and non-linearity of the reservoir computing. In order to avoid the randomness of the experimental results, each benchmark is conducted 30 times and the experiment result statistics are averaged. The measuring steps are as follows :

1. Use function *generate_transmitted_signal*() 2 times to generate 2 data sets with the dimension of 9216 $\times$ 1, and mark these 2 data sets as X_{train} and X_{test} respectively.
2. Put these 2 data sets to the test model to generate target data set Y_{train} and Y_{test}, respectively.
3. Create a set of RC with different configurations.
4. Train each RC with the input X_{train} and corresponding target Y_{train}.
5. Use each RC to predict output, marked as $Y_{predict}$, with X_{test} as input.
6. Use the metric of each benchmark to perform measurements with $Y_{predict}$ and Y_{test} as input.

5.2.1 Memory capacities

The memory capacities characterize in a simple way how a reservoir processes information.[1] MC ranges from $-\infty$ to 1. The closer MC is to 1, the better RC fits the given MC task, and vice versa. The MC task is the reconstruction of the function y_k[n] with u[n] as input. Equation(5.1) shows the MC expression.

$$C[y_k] = 1 - NMSE = 1 - \sum_{i=1}^{n} \left(\frac{y_k[i] - \hat{y_k}[i]}{y_k[i] - \overline{y}} \right)^2 \tag{5.1}$$

- C[y_k] : MC for MC task y_k.
- y_k[i] : Output of y_k for i_{th} input.
- $\hat{y_k}[i]$: Prediction of RC for i_{th} input.
- $\overline{y}$: Mean of all output.
- n: The number of all input.
- NMSE: Normalized mean square error.

When faced with an exceedingly complex Memory Capacity (MC) task, the MC value may be significantly negative. Both zero and negative values of MC indicate a significant failure of the current model, irrespective of the degree of negativity. An excessively negative MC can distort the overall observed trend. To mitigate this problem, we implement a normalization procedure for MC. This normalization specifically involves resetting all negative MC values to 0. Through this process, the MC is effectively bounded within the range of [0,1]. The MC expression with normalization is shown in equation(5.2).

$$C[y_k] = max\left(0, 1 - \sum_{i=1}^{n}\left(\frac{y_k[i] - \hat{y_k}[i]}{y_k[i] - \overline{y}}\right)^2\right) \tag{5.2}$$

In this part, We will explore how parameters, non-linear nodes number(N), MNN, and MS, influence the performance of RC. Three kinds of MC tasks will be used in this section, which are linear memory capacity task(LMC task), quadratic memory capacity task(QMC task), and cross memory capacity task(CMC task). Parameters : α, β, F_{NL} are set to 0.5, 0.3, $\frac{sinh(x)}{1+e^{-x}}$ respectively.

Linear memory capacity

Linear memory capacity is introduced by Jaeger in Ref [23]. It quantifies a system's ability to recall and reconstruct a sequence of past inputs. In essence, it measures the system's "linear memory"-its capacity to remember past inputs. The higher the linear memory capacity is for the tasks requiring longer memory, the longer the sequence of past inputs the system can accurately remember. This is crucial for tasks where the output depends directly on past inputs. Mathematically, the LMC task is shown in equation(5.3).

$$y_k(n) = u(n-k) \tag{5.3}$$

In this test, k ranges from 1 to 100. **Figure 5.1** shows the test result over all k values.

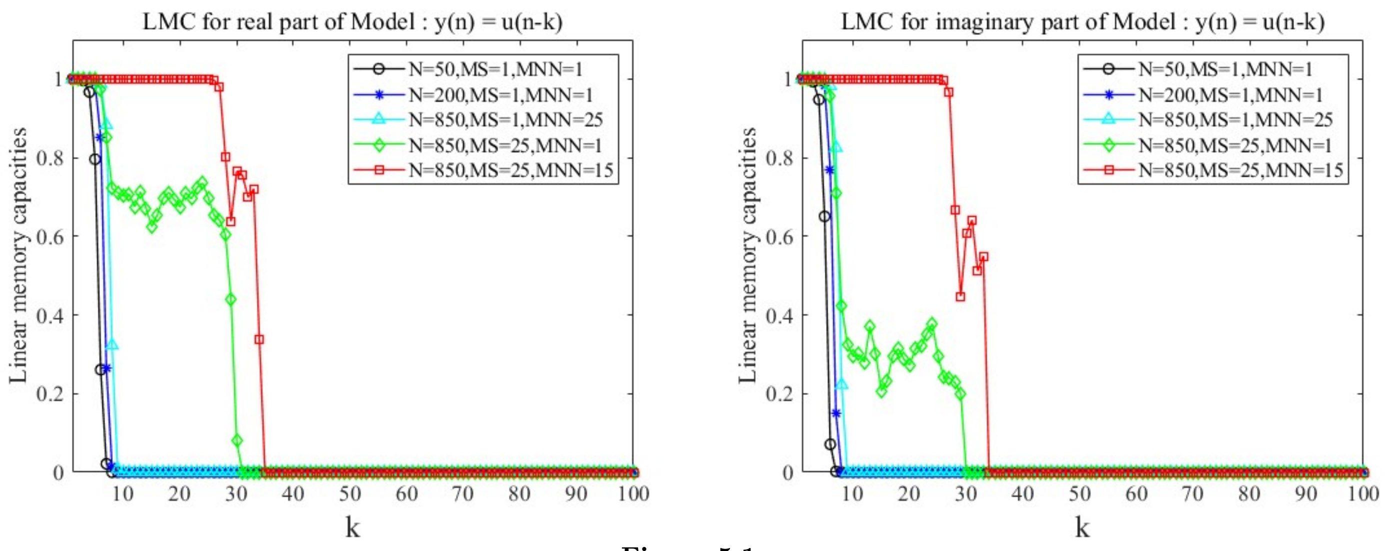

Figure 5.1

The test reveals that under the configuration of N=50, MS=1, and MNN=1, the RC has the capacity to reconstruct up to 4 previous inputs. Beyond this, LMC keeps significantly decreasing and it drops to zero for k>=8. When the parameter N is increased to 200, the RC is able to reconstruct 5 previous inputs effectively. As N is further increased to 850 and MNN is adjusted to 25, the RC is able to construct slightly more previous input. When MS is raised to 25 and MNN is reduced to 1, LMC can reconstruct 6 previous inputs and although LMC drops to zero when k is increased to 30, the range of LMC for values of 'k' between 7 and 30 does not satisfy the criteria for effective reconstruction. Under the configuration of N=850, MS=25, and MNN=15, the RC achieves optimal performance, flawlessly reconstructing 27 previous inputs. Hence, to effectively reconstruct past input, it is essential that both the MS and the MNN are suitably adjusted and optimized. The parameter N, however, exhibits only a slight impact in this particular test.

Quadratic memory capacity

Non-linear memory capacities are introduced in Ref[24]. QMC and CMC are non-linear memory capacities, which is an extension of LMC. The main difference is that non-linear memory capacities are to reconstruct the non-linear function of past input rather than the original past input. QMC quantifies the ability of a system to remember and reproduce squared versions of past inputs. This is particularly useful in tasks where the relationship between inputs and outputs is complex and non-linear. The QMC task is given by:

$$y_k(n) = u(n-k)^2 \tag{5.4}$$

The test result is shown in **Figure 5.2**

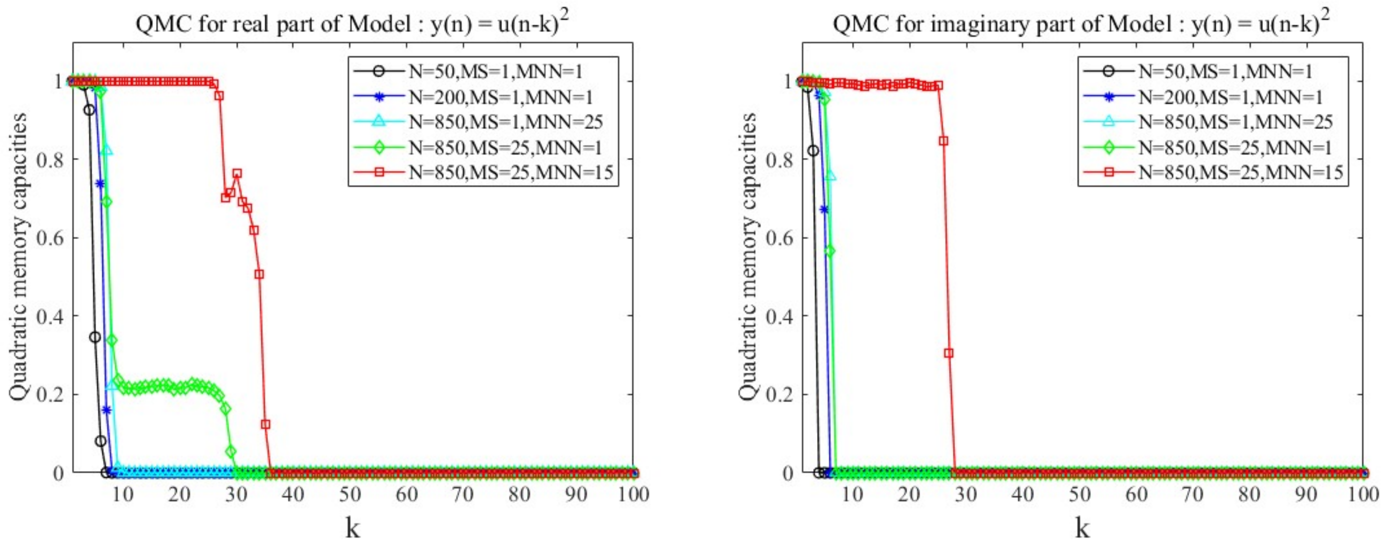

Figure 5.2

As shown in **Figure 5.2**, if either MNN or MS is 1, RC can only at most reconstruct the square of 3∼6 previous inputs. However, when MS=25 and MNN=15, RC can perfectly reconstruct the square of 27 previous inputs. Therefore, a single increase in MS or MNN, or N, will only have a slight impact on QMC and an increase in both MS and MNN will have a significant increase in QMC.

Cross memory capacity

The Cross Memory Capacity (CMC) is another measure of a system's non-linear memory. It quantifies the ability of the system to remember and reproduce the product of two different past inputs. This capacity is essential in tasks where the output depends on the interaction between different past inputs. The CMC task is given by:

$$y_k(n) = u(n-k)u(n-k') \tag{5.5}$$

In this test, k' is fixed to 1 and the test result is given by :

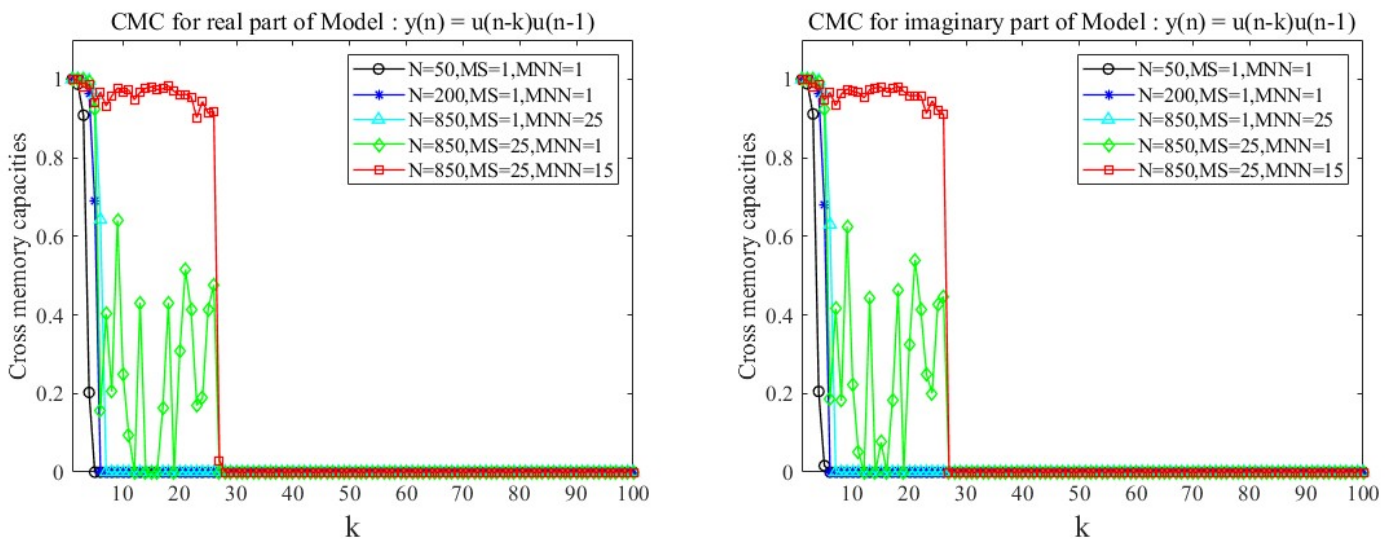

Figure 5.3

As depicted in **Figure 5.3**, it is noticeable that the configurations "N=50, MS=1, MNN=1", "N=200, MS=1, MNN=1", and "N=850, MS=1, MNN=25" exhibit similar patterns to those observed in Figure 4.2. The RC, when configured with N=850, MS=1, and MNN=25, can only effectively reconstruct CMC tasks when k is less than or equal to 5. In the range of k=10 to k=27, the CMC oscillates between 0 and 0.64. The configuration "N=850, MS=25, MNN=15" is capable of reconstructing CMC tasks when k is less than or equal to 26, but the performance in CMC tasks is slightly worse than QMC tasks and LMC tasks.

5.2.2 NARMA10

The emulation task of a nonlinear autoregressive moving average model, i.e., the NARMA10 task, has been widely used as a benchmark task for recurrent neural networks, especially in reservoir computing.[25] The NARMA10 task is frequently used because it provides a challenging benchmark for testing and evaluating RNNs and other types of machine learning algorithms designed to handle time-series data. The NARMA10 task measures :

- **Nonlinearity:** The NARMA10 task is nonlinear, which makes it more complex and challenging than simple linear autoregressive tasks. This can be useful for assessing how well an algorithm handles nonlinear relationships.
- **Memory Requirements:** The NARMA10 task requires a system to 'remember' the last 10 time steps of inputs and outputs. This feature allows researchers to measure a model's ability to process and store information over time. It's a good test of a model's 'memory' capabilities.
- **Temporal Dependencies:** The outputs in the NARMA10 task are dependent on the previous inputs and outputs, creating complex temporal dependencies. It's a good way to assess how well a model can reconstruct this complex temporal dependency.

The NARMA10 expression is as follows:

$$y(n) = 0.3 \cdot y(n-1) + 0.05 \cdot y(n-1) \sum_{i=1}^{10} y(n-i) + 1.5 \cdot u(n-1) \cdot u(n-10) + 0.1 \tag{5.6}$$

In this section, Mean Squared Error(MSE) is used to measure RC's performance in the NARMA10 task. The formula for calculating MSE is:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y(i) - \hat{y}(i))^2 \tag{5.7}$$

- n: The number of all output.
- y(i): The NARMA10 system output for i_{th} input.
- $\hat{y}$(i): RC's prediction for i_{th} input.

The Mean Squared Error (MSE) is invariably greater than or equal to zero, and a value of zero signifies a perfect fit with the data. In practice, this is usually not achievable, and so the goal is often to minimize the MSE to as low as possible. The benefit of squaring the residuals (differences) is that it penalizes larger errors more heavily than smaller ones, meaning models are more heavily penalized for making predictions that are further from the true value, which can facilitate observing the overall trend. This section aims to investigate the influence of variations in Mean Node Number MNN, MS, and N on the performance of RC in tackling the NARMA10 task. Furthermore, we will conduct a prediction visualization to observe how well RC fits the NARMA10 task. Parameters : α, β, F_{NL} are still set to 0.5, 0.3, $\frac{sinh(x)}{1+e^{-x}}$ respectively.

NARMA10 task with variation in N

In this test, we set the MNN and MS to 1, while varying the N within the range of 50 to 850. The goal is to evaluate the impact of the N on the performance of RC when applied to the NARMA10 task. The findings from this test are presented in the following:

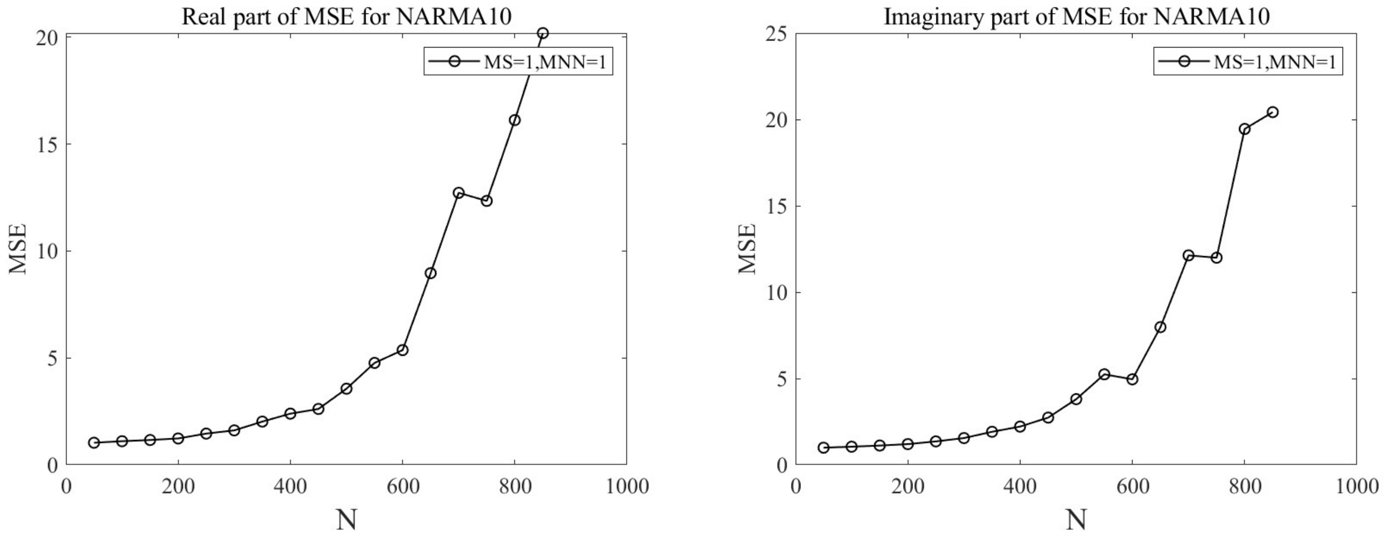

Figure 5.4

From our observations, there appears to be a consistent trend: as the N increases, the MSE correspondingly escalates. This implies a degradation in the performance of Reservoir Computing (RC) in the context of the NARMA10 task. Therefore, it can be concluded that merely expanding the N, while keeping the MS and MNN constant, leads to an elevation in MSE, thereby negatively impacting the performance of RC in the NARMA10 task.

NARMA10 task with variation in MNN

In this particular test, we fixed the MS at 25 and the N at 850, while the MNN varies from 1 to 15. Our objective is to find the influence of altering the MNN on the performance of RC within the framework of the NARMA10 task. The outcomes of this investigation are detailed in **Figure 5.5**.

As illustrated in **Figure 5.5**, we observe a notable trend: as the MNN increases, the MSE correspondingly decreases, gradually approaching zero. However, an interesting observation is that, when MNN exceeds 14, a slight but discernible upward trend in MSE begins to emerge, despite this increase being relatively minor. Hence, it can be inferred that an optimal value for the MNN exists, and it's crucial to note that a higher MNN doesn't necessarily correspond to a lower MSE.

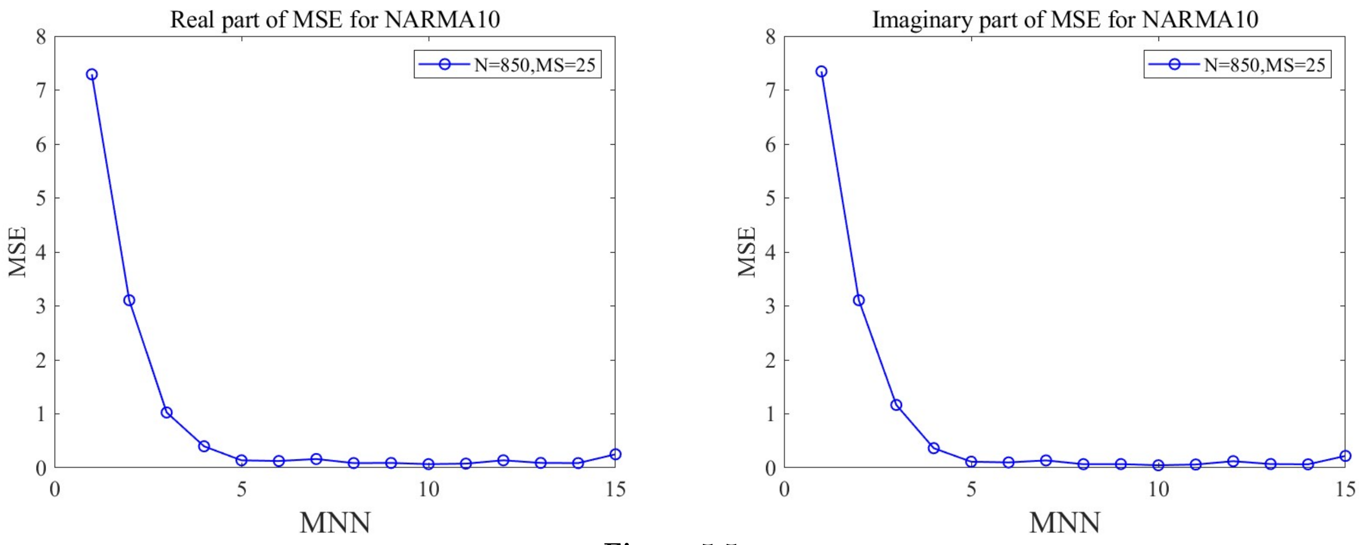

Figure 5.5

NARMA10 task with variation in MS

In this specific experiment, we hold the MNN at 15 and the N at 850, while varying the MS within the range of 1 to 25. The objective of this experiment was to examine the impact of changes in MS on the efficacy of RC when applied to the NARMA10 task. The results derived from this experiment are vividly presented in **Figure 5.6**.

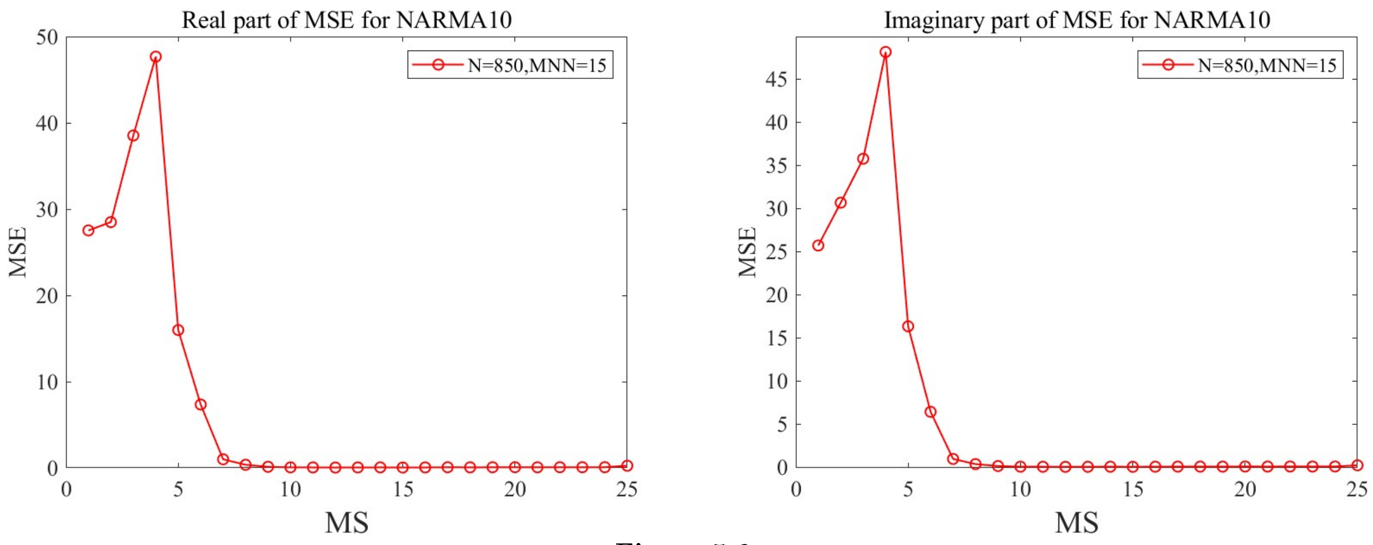

Figure 5.6

As evidenced in **Figure 5.6**, an increase in the MS seems to correspond to a decrease in the MSE.

Notably, when MS is equal to or exceeds 9, the MSE approximates zero. Comparatively, when MS is particularly low, the MSE significantly exceeds the values observed in **Figure 5.5**. Thus, it can be inferred that manipulating the MS has a profound effect on enhancing the performance of RC when tasked with the NARMA10 system.

Prediction visualization

In this section, we will visualize both NARMA10 output and RC prediction to see how well RC fits the NARMA10 task. In this section, we will compare RC in Ref[1] and the RC introduced in this project. For RC in Ref[1], the configuration is "MNN=1, MS=1, and N=50". For RC in this project, parameters: MNN, MS, and N are set to 15, 25, and 1000 respectively. Parameters α, β, F_{NL} are selected by grid search for RC in both Ref[1] and this project.

In this section, we aim to provide a visual comparison of the NARMA10 output and the predictions generated by RCs to clearly observe the effectiveness of the RC model in fitting the NARMA10 task. The experiment result is shown following:

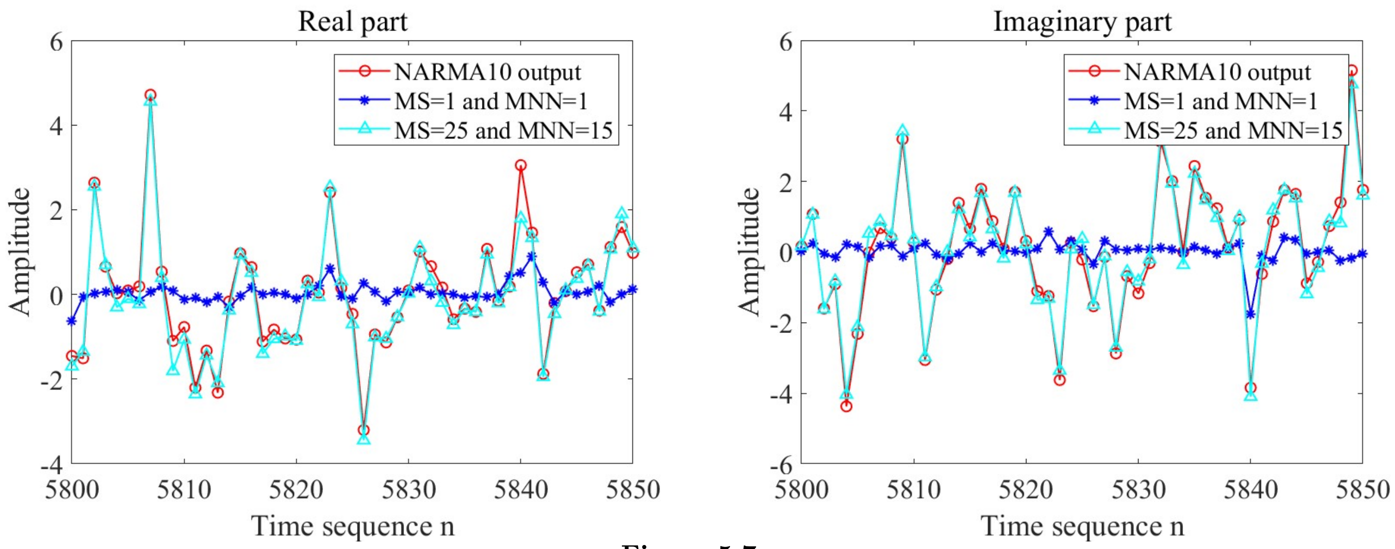

Figure 5.7

From our observations, it is evident that the Reservoir Computing (RC) with the configuration of "N=1000, MS=25, MNN=15" exhibits a substantially better fit to the NARMA10 task than the RC model proposed in Ref[1], despite few discrepancies. Overall, the prediction from this configuration closely aligns with the NARMA10 task output.

Contrastingly, the RC model suggested in Ref[1] does not perform well on the NARMA10 task due to the considerable divergence between its predictions and the actual NARMA10 output. Therefore, it can be concluded that the RC framework proposed in this project significantly enhances the performance of

RC on the NARMA10 task.

5.3 Simulation

In this section, we will conduct a comprehensive simulation, which includes modeling both multi-path propagation in a wireless channel and the effects of hardware non-linearity in both the receiver and the transceiver. Then we use RC to anticipate these effects and finally use the received signal to subtract RC's prediction to get a correct signal.

For the simulation of hardware non-linearity at both ends, we use non-linearity related to telecommunication applications in equation (5) from Ref[1]. This nonlinearity can be considered as a PH model with a memory length of 0 and non-linear order of 3.

In regard to the multi-path propagation in wireless channels, we use the TDL-A model. TDL-A model provides normalized delay and power for each path and Fading distribution for the TDL-A model is Rayleigh. To get the desired delay, we need to multiply normalized delay by delay spread. this expression is given by:

$$\tau_{n,scaled} = \tau_{n,model} \cdot DS_{desired} \tag{5.8}$$

Where:

- The normalized delay of the n_{th} path in a particular Tapped Delay Line (TDL) model is represented by $\tau_{n,model}$.
- The value denoted by $\tau_{n,scaled}$ represents the targeted delay, measured in nanoseconds (ns).
- The symbol $DS_{desired}$ defines the target delay spread, expressed in nanoseconds (ns).

We use the MATLAB function 'nrTDLChannel' to simulate multi-path and the details for the TDL-A model are in TR 38.901 Section 7.7.2, Tables 7.7.2-1 to 7.7.2-5. The TR 38.901 file is in Ref[26]. The parameters' setting for MATLAB function 'nrTDLChannel' is given by:

Parameter	setting
DelayProfile	TDL-A
DelaySpread	100ns
SampleRate	50MHz
MaximumDopplerShift	0Hz
NumTransmitAntennas	1
NumReceiveAntennas	1

Table 5.1: Parameters' setting for MATLAB function 'nrTDLChannel'

The signal processed through this simulation task can be modeled by the following expressions:

$$T_X(n) = u(n) + 0.036 \cdot u(n)^2 - 0.011 \cdot u(n)^3 \tag{5.9}$$

$$chan(n) = \sum_{i=1}^{k} h_i \cdot T_X(n - \tau_i) \tag{5.10}$$

$$R_X(n) = chan(n) + 0.036 \cdot chan(n)^2 - 0.011 \cdot chan(n)^3 \tag{5.11}$$

- **u(n):** Transmitted 5G signals.
- $T_X(n)$**:** Non-linear distortion at the transmitter.
- **chan(n):** Multipath propagation in the Rayleigh fading channel.
- $R_X(n)$**:** Non-linear distortion at the receiver.

In our simulation, we use cancellation depth as a key metric to evaluate the performance of RC. The expression for cancellation depth is :

$$\text{Cancellation depth(dB)} = 20 \cdot \log_{10} \left(\frac{\sum_{i=1}^{n} |y(i)|}{\sum_{i=1}^{n} |y(i) - \hat{y}(i)|} \right) \tag{5.12}$$

Cancellation depth is a measure used to quantify the efficacy of a noise or interference cancellation system. It's expressed in decibels (dB), which is a logarithmic scale commonly used to represent the ratio of two quantities. The higher the cancellation depth, the more effectively noise or interference has been eliminated from the signal.

The measuring steps are different from the benchmarks section and it is given by:

1. Use function *generate_transmitted_signal*() 2 times to generate 2 data sets with the dimension of 9216 × 1, and mark these 2 data sets as X_{train} and X_{test} respectively.
2. Equation(5.9), Equation(5.10), and Equation(5.11) process these 2 data sets in series and mark the final output as Y_{train} and Y_{test}, respectively.
3. Update Y_{train}: $Y_{train} = Y_{train} - X_{train}$.
4. Create a set of RC with different configurations.
5. Train each RC with input X_{train} and corresponding target Y_{train}.
6. Use each RC to predict output with X_{test} as input and the prediction is marked as $Y_{predict}$.
7. Update $Y_{predict}$: $Y_{predict} = Y_{test} - Y_{predict}$
8. Calculate cancellation depth for both the real part and imaginary part of test data with $Y_{predict}$ and Y_{test} as input, and mark them as $\Re$(Cancellation depth) and $\Im$(Cancellation depth) respectively.

9. Update cancellation depth as : Cancellation depth $= \frac{\Re(Cancellation\ depth)+\Im(Cancellation\ depth)}{2}$

In the forthcoming section, we explore what degree of impact can MS, MNN, and non-linear functions exert on RC's performance in simulated wireless cases. Ultimately, we will enhance our comparative analysis by presenting visualized predictions, allowing for a more clear evaluation of the performance of RC as depicted in Ref[1] and the RC within this project.

The set of non-linear functions under consideration comprises:

- $\frac{sinh(x)}{1+e^{-x}}$
- $sin(x)$
- $cos(x)$
- $tanh(x)$
- $\frac{1}{1+e^{-x}}$

5.3.1 Simulation result

In the conducted simulation, a comparative study is executed between two distinctive configurations: "MS=1, MNN=1, and N=50" and "MS=35, MNN=25, and N=1800". Additionally, the study extends to investigate the influence of diverse non-linear functions on these two configurations. Within this simulation, parameters α and β are configured at values of 0.5 and 0.3, respectively. To mitigate the possibility of results being random events, each experimental trial is performed 30 times, and the ensuing results are averaged to acquire reliable statistics. The resulting experimental statistics are as follows:

Non-linear function	Cancellation depth(dB)
tanh(x)	3.09
sin(x)	3.10
cos(x)	3.04
$\frac{1}{1+e^{-x}}$	3.06
$\frac{sinh(x)}{1+e^{-x}}$	3.08

Table 5.2: MS=1, MNN=1 and N=50

Non-linear function	Cancellation depth(dB)
tanh(x)	15.65
sin(x)	22.65
cos(x)	27.88
$\frac{1}{1+e^{-x}}$	31.55
$\frac{sinh(x)}{1+e^{-x}}$	29.51

Table 5.3: MS=35, MNN=25 and N=1800

The observations indicate that with MS and MNN set to 1, the cancellation depth approximates 3dB, irrespective of the non-linear functions applied. In this context, a cancellation depth of 3dB implies that the error contributes about 71% of the correct signal's magnitude on average. This significant error is comparable to the target signal, meaning a substantial failure.

However, when MS and MNN are increased to 35 and 25, respectively, there is a distinct improvement in performance across all non-linear functions. Despite this enhancement, discrepancies in performance become evident. The RC incorporating tanh(x) as its non-linear function exhibits the weakest performance, with a cancellation depth of 15.65dB, signifying that the error's magnitude constitutes 16.5% of the target signal.

To make the error negligible, the target signal should be at least 20 times greater than the error. Only the RC with the non-linear functions cos(x), $\frac{1}{1+e^{-x}}$, or $\frac{sinh(x)}{1+e^{-x}}$ meets this criteria. Among these, the RC with the non-linear function $\frac{1}{1+e^{-x}}$ outperforms the others, achieving a cancellation depth of 31.55dB. In this case, the error is merely 2.6% of the target signal on average, suggesting that the target signal is at least 37 times greater than the error.

5.3.2 Prediction visualization

In this section, we aim to provide a visual comparison of the correct signal against the RC prediction. The intention is to observe how effectively the RC methodology mitigates non-linearity and the multi-path effect. Here, we will compare the RC implementation from Ref[1] with the RC implementation in our project. For the RC in Ref[1], the parameters are defined as follows: MNN=1, MS=1, and N=50. For RC in this project, parameters: N, MNN, and MS are set to 1800, 35, and 25, respectively. Parameters such as α, β, and F_{NL} for RC in both Ref[1] and our project are optimized by grid search. The visualization result is given by:

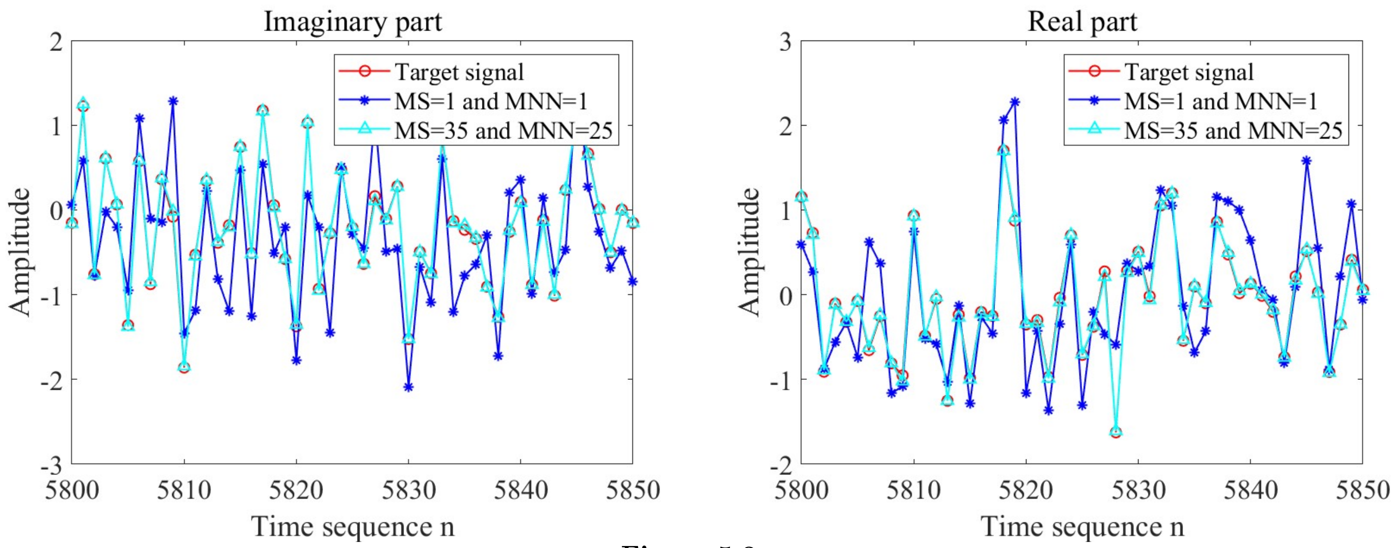

Figure 5.8

From **Figure 5.8**, it's clear that the Reservoir Computing (RC) model as proposed in Ref[1] exhibits distinct discrepancies between target signal and predicted signal for both real and imaginary part. This is because the cancellation depth is 3.12dB, which means the error constitutes 69.8% of the target signal. Such a high error rate indicates that the error is comparable to the target signal amplitude.

Conversely, the RC model introduced in this project with the configuration of MS=35 and MNN=25 outperforms the former, exhibiting almost no discernible deviation from the correct signal. With a cancellation depth of 32.12dB, the magnitude of the correct signal is approximately 44 times greater than the error. Consequently, any error becomes negligible.

These findings unequivocally demonstrate the superior performance of the RC model designed in this project, underscoring its potential to significantly enhance signal correction capabilities.

Chapter 6

Discussion and Conclusion

The primary objective of this research is to assess the potential utility of reservoir computing (RC) to counteract non-linearity and multi-path interference in wireless communication scenarios. Nevertheless, we identified an inherent limitation in conventional RC methodology, where it may forget prior inputs more rapidly than the given task may necessitate. To address this challenge, we introduced a novel approach incorporating additional parameters: Memory Strength (MS) and Memory Node Number (MNN). This approach is evaluated using two benchmark tasks: Memory Capacity and NARMA10. A comprehensive simulation is also conducted to examine its effectiveness in real-world scenarios.

6.1 Experiment results interpretation

6.1.1 Interpretation on MC tasks

For all MC tasks, when both MS and MNN are set to one, RC performs competently only on tasks that necessitate recent input memories. However, it underperforms tasks that require more distant input memories because earlier memories are diminished to near zero.

Our observations indicate that a singular increase in MS fails to effectively handle tasks demanding higher delay. This deficiency primarily arises due to the lack of adequate diversity - subpar diversity leads to poor approximation. However, when both MS and MNN are elevated, there is a significant enhancement in RC's performance, especially for tasks necessitating extensive memory.

In the case of both LMC and QMC tasks, RC exhibits distinct behaviors on the real and imaginary parts of the signal when configured with "N=850, MS=25, MNN=1". This discrepancy could be attributed to the fact that the mask array consists of complex numbers, while the coefficient for each task is a real number.

For the CMC task, setting MS and MNN at 25 and 1, respectively, results in pronounced oscillations of CMC values between CMC=0.65 and CMC=0 for k in the range between 8 and 30. This oscillatory behavior is induced because the CMC task poses more significant challenges than LMC and QMC tasks, leading to overfitting by RC and a consequent loss in its ability to reconstruct CMC temporal dependencies.

Overall, the performance for the CMC task is inferior to the QMC and LMC tasks. This underperformance can be linked back to the inherent complexity of the CMC task, which surpasses that of the other tasks.

6.1.2 Interpretation on NARMA10 tasks

In the NARMA10 benchmark, a significant decrease in MSE is only achieved by a proper increase in both MS and MNN. This confirms the earlier stated impact of MS and MNN. Conversely, escalating N while maintaining MS and MNN at one results in a gradual increase in MSE. With both MS and MNN at 1, RC lacks the capability to learn underlying patterns. Consequently, RC learns erroneous patterns in its efforts to approach training target data. The heightened N generally enhances RC's approximation to training target data; however, this also enables it to learn More severely incorrect patterns. The complexity inherent in the NARMA10 system exacerbates this issue, as the greater inaccuracy inevitably leads to larger errors.

6.1.3 Interpretation on simulation

In the simulation, MS and MNN show the same impact as illustrated before. Certain non-linear functions exhibited superior performance, whereas others did not meet expectations. Regardless of the non-linear function, an improvement is seen in the performance of RC with an increase in both MS and MNN values. It is important to note that the degree of non-linearity required can differ based on the task. Hence, adjusting the parameter non-linear function to best fit specific tasks could significantly enhance performance.

6.2 Limitations

This research introduces the parameters Memory Strength (MS) and Memory Node Number (MNN) to improve the performance of Reservoir Computing (RC) in handling complex tasks, but this approach comes with certain limitations.

The primary constraint is the computational expense. Tasks that exhibit high delay necessitate a larger MS, while tasks with significant non-linearity demand a higher MNN. As the number of nodes required needs to be at least (MS+1)×MNN, the computational cost increases if both parameters need to be substantially large.

The second limitation arises due to the addition of MS and MNN as new parameters, which effectively increases the count of hyperparameters. This results in an increasing number of all possible combinations in the search space for optimal hyperparameters, consequently increasing the time required for optimization.

Lastly, there seems to be a performance plateau associated with these parameters. Beyond certain thresholds, further increases in either MS or MNN fail to bring about any noticeable improvements in the RC's performance.

6.3 Conclusion

In conclusion, the innovative structure of Reservoir Computing (RC) implemented in this project has exhibited an exceptional capability to manage fading memory, thereby tailoring itself to the assigned task. The RC model, trained on a data set that simulates real-world conditions, has demonstrated remarkable performance in predicting and mitigating associated impacts. Despite the increase in computational cost, this novel method substantially enhances the performance of RC in handling more complex tasks, as evidenced by comparisons with the RC introduced in Ref[1]. This research underscores the transformative potential of RC in significantly improving signal clarity and reliability in wireless communications, thereby heralding a new era of more robust and reliable wireless communication systems.

Chapter 7

Future work

In current stage, some work has been done successfully and all matlab files for this project is in **Appendix C**. However, there are still some work need to be done in the future.

7.1 Finished work

In current stage, all finished work is listed below:

- Understand principles behind reservoir computing.
- Build a novel reservoir model
- Build non-linearity model
- Conduct two benchmark tasks: MC tasks and NARMA10 tasks.
- Build simulation for a real wireless communication of one transmitter and one receiver.
- Findings for all experiments.

7.2 Work to be done at next stage

Work to be done in the future is listed below:

- Find a PH model or guess a reasonable PH model for simulation.
- Build simulation of multiple transmitters and multiple receivers.
- Practically implement reservoir computing(Optional).
- Built another common technique to compare with reservoir computing(Optional).
- Find a way to improve the performance of reservoir computing.

Acknowledgements

My profound thanks go to everyone who afforded me the opportunity to finalize this project. I reserve particular acknowledgment for my project supervisor, Tharm Ratnarajah, for his vital inspiration, suggestions, and support that assisted in shaping my project.

In addition, my sincere gratitude extends to Haifeng Luo, who orchestrated the planning of my project and promptly addressed my queries concerning this project.

Furthermore, appreciation is extended to the authors of the references utilized in this work. Their insights and contributions have enriched my understanding and inspired various aspects of my project.

References

[1] Duport, F., Schneider, B., Smerieri, A., Haelterman, M. and Massar, S., 'All-optical reservoir computing,' *Opt. Express*, vol. 20, no. 20, pp. 22 783–22 795, 2012. DOI: `10.1364/OE.20.022783`.

[2] Tanaka, G., Yamane, T., Héroux, J. B. *et al.*, 'Recent advances in physical reservoir computing: A review,' *Neural Networks*, vol. 115, pp. 100–123, 2019. DOI: `https://doi.org/10.1016/j.neunet.2019.03.005`.

[3] Mathuranathan. 'Fading channel – complex baseband equivalent models', GaussianWaves. (2010), `https://www.gaussianwaves.com/2010/02/fading-channels-rayleigh-fading-2/`.

[4] Anttila, L., Handel, P. and Valkama, M., 'Joint mitigation of power amplifier and i/q modulator impairments in broadband direct-conversion transmitters,' *IEEE Transactions on Microwave Theory and Techniques*, vol. 58, no. 4, pp. 730–739, 2010. DOI: `10.1109/TMTT.2010.2041579`.

[5] Donges, N. 'A guide to recurrent neural networks: Understanding rnn and lstm networks'. (), `https://builtin.com/data-science/recurrent-neural-networks-and-lstm`.

[6] Xue, F., Li, Q. and Li, X., 'The combination of circle topology and leaky integrator neurons remarkably improves the performance of echo state network on time series prediction,' *PLOS ONE*, vol. 12, no. 7, pp. 1–17, 2017. DOI: `10.1371/journal.pone.0181816`.

[7] Rodan, A. and Tino, P., 'Minimum complexity echo state network,' *IEEE Transactions on Neural Networks*, vol. 22, no. 1, pp. 131–144, 2011. DOI: `10.1109/TNN.2010.2089641`.

[8] Rodan, A. and Tiňo, P.: 'Simple deterministically constructed recurrent neural networks,' in *Proceedings of the 11th International Conference on Intelligent Data Engineering and Automated Learning*, pp. 267–274.

[9] *What is a neural network?* [Online]. Available: `https://www.ibm.com/topics/neural-networks`.

[10] Singh, G.: *Introduction to artificial neural networks.* [Online]. Available: `https://www.analyticsvidhya.com/blog/2021/09/introduction-to-artificial-neural-networks/`.

[11] *Artificial neural network tutorial.* [Online]. Available: `https://www.javatpoint.com/artificial-neural-network`.

[12] Kurama, V.: *Regression in machine learning: What it is and examples of different models.* [Online]. Available: `https://builtin.com/data-science/regression-machine-learning`.

[13] Castillo, D.: *Machine learning regression explained*, Aug. 2021. [Online]. Available: https://builtin.com/data-science/regression-machine-learning.

[14] Vadapalli, P.: *6 types of regression models in machine learning you should know about*, Aug. 2021. [Online]. Available: https://www.upgrad.com/blog/types-of-regression-models-in-machine-learning/.

[15] Nabi, J.: *Recurrent neural networks (rnns)*, Jul. 2019. [Online]. Available: https://towardsdatascience.com/recurrent-neural-networks-rnns-3f06d7653a85.

[16] Kalita, D.: *A brief overview of recurrent neural networks (rnn).* [Online]. Available: https://www.analyticsvidhya.com/blog/2022/03/a-brief-overview-of-recurrent-neural-networks-rnn/.

[17] Kalita, D.: *Recurrent neural network tutorial (rnn).* [Online]. Available: https://www.datacamp.com/tutorial/tutorial-for-recurrent-neural-network.

[18] Noravesh, F.: *Part 1: Echo state networks.* [Online]. Available: https://www.youtube.com/watch?v=uF4i9_7IQlI.

[19] RODAN, A. A. A. A.: *Architectural designs of echo state network.* [Online]. Available: https://etheses.bham.ac.uk/id/eprint/3610/1/Alrodan12PhD.pdf.

[20] *Hands-on reservoir computing: A tutorial for practical implementation.* [Online]. Available: https://iopscience.iop.org/article/10.1088/2634-4386/ac7db7/pdf.

[21] *Introduction to reservoir computing methods.* [Online]. Available: https://amslaurea.unibo.it/8268/1/melandri_luca_tesi.pdf.

[22] Paquot, Y., Duport, F., Smerieri, A. *et al.*, 'Optoelectronic reservoir computing,' *Scientific Reports*, vol. 2, no. 1, p. 287, 2012. DOI: 10.1038/srep00287.

[23] Jaeger, H.: 'Short term memory in echo state networks', GMD - German National Research Institute for Computer Science, Tech. Rep. 152, 2001.

[24] Dambre, J., Verstraeten, D., Schrauwen, B. and Massar, S., 'Information processing capacity of dynamical systems,' *Scientific reports*, vol. 2, no. 1, p. 514, 2012. DOI: https://doi.org/10.1038/srep00514.

[25] Kubota, T., Nakajima, K. and Takahashi, H.: *Dynamical anatomy of narma10 benchmark task*, Jun. 2019. [Online]. Available: https://www.arxiv-vanity.com/papers/1906.04608/.

[26] *Etsi tr 138 901 v14.0.0 (2017-05)*, May 2017. [Online]. Available: https://www.etsi.org/deliver/etsi_tr/138900_138999/138901/14.00.00_60/tr_138901v140000p.pdf.

[27] Chicco, D., Warrens, M. J. and Jurman, G., 'The coefficient of determination r-squared is more informative than smape, mae, mape, mse and rmse in regression analysis evaluation,' *PeerJ Computer Science*, vol. 7, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:236196832.

Appendix A

Derivation of Ridge Regression

A.1 Ridge Regression Formula

The Ridge Regression estimator for the coefficient vector W_{out} is given by the formula:

$$W_{out} = (X^T X + \lambda I)^{-1} X^T Y \tag{A.1}$$

Where:

- W_{out} is the weight vector.
- X is the input matrix.
- Y is the output vector.
- X^T is the transpose of the input matrix.
- λ is the regularization parameter.
- I is the identity matrix.

A.2 Mathematical Derivation

The goal of ridge regression is to find the values for the weight vector W_{out} which minimize the ridge cost function:

$$C(W_{out}) = ||Y - XW_{out}||^2 + \lambda||W_{out}||^2 \tag{A.2}$$

Taking the derivative of this cost function with respect to the weight vector W_{out} and setting the result to zero gives us the equation for ridge regression.

$$\frac{\partial C(W_{out})}{\partial W_{out}} = 0 \tag{A.3}$$

This results in:

$$
\begin{aligned}
\frac{\partial}{\partial W_{out}}(Y - XW_{out})^T(Y - XW_{out}) + \lambda W_{out}^T W_{out} &= 0 \\
\frac{\partial}{\partial W_{out}}(Y^TY - W_{out}^T X^T Y - Y^T X W_{out} + W_{out}^T X^T X W_{out}) + \lambda W_{out}^T W_{out} &= 0 \\
-2X^TY + 2X^TXW_{out} + 2\lambda W_{out} &= 0 \\
2X^TXW_{out} + 2\lambda W_{out} &= 2X^TY \\
(X^TX + \lambda I)W_{out} &= X^TY \\
W_{out} &= (X^TX + \lambda I)^{-1}X^TY
\end{aligned}
$$

The above derivation shows the derivation of the Ridge Regression formula from the cost function. The mathematical derivation for regression is the same as above, which only requires to set λ to 0.

Appendix B

Grid search

This appendix will introduce detailed settings for grid search of reservoir computing.

B.1 Hyperparameters

There are six hyperparameters that can be tuned, which are α, β, F_{NL}, MS, MNN, and N, respectively. However, because the sum of α and β can't be greater than 1, α and β set is used, which is marked as α_β_Set.

B.2 Grid

Not all hyperparameters are tuned in the experiment part and it depends on the task. If one hyperparameter does not need to tune, we can give an array of this hyperparameter with the size of 1. The setting for each hyperparameter is given by:

- α_β_Set = [0.1, 0.8; 0.2, 0.7; 0.3, 0.6; 0.4, 0.5; 0.5, 0.4; 0.6, 0.3; 0.7, 0.2; 0.8, 0.1]
- F_{NL} = { @x tanh(x), @x sin(x), @x cos(x), @x $\frac{1}{1+e^(-x)}$, @x $\frac{sih(x)}{1+e^(-x)}$ }
- MS = 1:35 = [1, 2, 3, 4, 5, ..., 31, 32, 33, 34, 35]
- MNN = 1:25 = [1, 2, 3, 4, 5,..., 21, 22, 23, 24, 25]
- N = 50:50:3000 = [50, 100, 150, 200, ..., 2800, 2850, 2900, 2950, 3000]

Then we combine all needed hyperparameters to form a grid with all possible combinations. Then, give these hyperparameters to reservoir computing and run RCs with different hyperparameters to test the performance of each RC. Finally, choose hyperparameters with the best performance.

B.3 k-folds

In the experiment, we set k to 3, which means 1/3 data is used for testing and 2/3 data is used for training. This process will repeat k times and each time testing data and training data are randomly chosen. Finally, we averaged the metric for all repeats to get the final metric. The metric we used for the grid search is mean absolute error(MAE). The expression for MAE is given by:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{B.1}$$

Where:

- n is the total number of data points.
- y_i represents the true value of the i^{th} data point.
- $\hat{y}_i$ represents the predicted value of the i^{th} data point.
- $|\cdot|$ denotes the absolute value function.

The commonly used k values are 3, 5, and 10.

Appendix C

MATLAB files

At the current stage, this project encompasses a total of 17 MATLAB files, all organized within the same directory. The 'main.m' file serves as the primary executable that coordinates the functions of all the other files, running a comprehensive suite of tasks. Thus, only 'main.m' needs to be executed to generate all results. There are six MATLAB packages used in this project. These packages are "5G Toolbox", "Communications Toolbox", "Data Acquisition Toolbox", "DSP System Toolbox", "Signal Processing Toolbox", and "Statistics and Machine Learning Toolbox".

The subsequent sections in this appendix correspond to each individual MATLAB file. Each section is titled after its respective MATLAB file name and provides a detailed explanation of the specific functions and tasks carried out by that file. By reviewing these subappendices, readers can gain a thorough understanding of the purpose and functionality of each component within this MATLAB project.

C.1 Reservoir.m

This MATLAB code defines a class for a reservoir computing system. It has properties that define a reservoir's state, as well as methods that allow the reservoir to transform inputs, fit data, and make predictions. The code for this file is given by:

```
classdef Reservoir
    properties
        node_num;
        Non_linear_function;
        alpha;
        beta;
        mask;
        W;
        MS;
        MNN;
```

```
end

methods
    function obj = Reservoir(node_num, Non_linear_function, alpha, beta, MS,
         MNN)
        obj.node_num = node_num;
        obj.Non_linear_function = Non_linear_function;
        obj.alpha = alpha;
        obj.beta = beta;
        obj.mask = unifrnd(-1/sqrt(2), 1/sqrt(2), [1,node_num]) + unifrnd
            (-1/sqrt(2), 1/sqrt(2), [1,node_num])*1i;
        obj.W = zeros(node_num, 1);
        obj.MS = MS;
        obj.MNN = MNN;
    end

    function node_states = transform(obj, X)
        node_states = zeros(length(X), obj.node_num);
        previous_states = zeros(obj.node_num,1);
        Delay_nodes = zeros(obj.MS, obj.MNN);

        for i = (1:length(X))
            for j = (1:obj.node_num)
                if j <= obj.MS*obj.MNN
                    node_states(i,j) = obj.Non_linear_function(obj.alpha*
                        Delay_nodes(floor((j-1)/obj.MNN)+1,rem(j-1,obj.MNN)
                        +1) + obj.beta*obj.mask(j)*X(i));

                    if j <= obj.MNN
                        Delay_nodes(1,j) = previous_states(obj.node_num - (
                            obj.MNN - j));
                    end

                else
                    node_states(i,j) = obj.Non_linear_function(obj.alpha*
                        previous_states(j-obj.MS*obj.MNN) + obj.beta*obj.
                        mask(j)*X(i));
                end
            end

            for k = (1:obj.node_num)
                previous_states(k) = node_states(i,k);
            end
```

```
                for l = (obj.MS:-1:2)
                    for m = (1:obj.MNN)
                        Delay_nodes(l,m) = Delay_nodes(l-1,m);
                    end
                end

            end

        end

        function obj = fit(obj, X, y, lambda)

            if nargin < 4
                lambda = 0;
            end

            X_train = obj.transform(X);

            obj.W = pinv(((X_train' * X_train) + lambda * eye(size(X_train, 2)))
               ) * (X_train') * y;

        end

        function y = predict(obj, X)

            node_states = obj.transform(X);

            y = node_states * obj.W;
        end
    end
end
```

C.2 PH.m

This file is simply built for Parallel Hammerstein Model. The following MATLAB function named ‘PH‘ takes as input a data vector ‘X‘, a matrix of coefficients ‘Coefficients‘, a memory length ‘Memory_length‘, and a non-linear order ‘Non_linear_order‘. It outputs a vector ‘y‘ of the same length as ‘X‘. The function computes ‘y‘ by applying a non-linear transformation to each element in ‘X‘ within the memory length. The code is given by:

```
function [y] = PH(X,Coefficients,Memory_length,Non_linear_order)
y = zeros(length(X),1);
```

```
for i = 1 : length(X)
    for k = 1 : 2 : Non_linear_order
        for j = 0 : 1 : Memory_length
            if i <= j
                break;
            else
                y(i) = y(i) + Coefficients(floor(k./2)+1,j+1)*X(i-j)*(abs(X(i-j)
                    ).^(k-1));
            end
        end
    end
end
end
```

C.3 fun_codedPayloadGen.m

The following MATLAB function named ‘fun_codedPayloadGen‘ is used to generate a payload and related outputs like the input bits and modulated symbols in a communication system. It takes several parameters including the modulation method, number of symbols, number of subcarriers, number of transmitting antennas, trellis, and others. Depending on the modulation method (‘OTFS‘, ‘OFDM‘ or else) and modulation scheme (‘QPSK‘, ‘16QAM‘, ‘64QAM‘), the payload and other outputs are generated in a different way. The code is given by:

```
function [payload,inputBits,modSym,encoderIdx,turboEncoder] =
    fun_codedPayloadGen(MOD, numSymbols, numSubcarriers, numTxAnt, trellis, n, L
    , Method, varargin)

if strcmp(Method,'OTFS') == true

    if strcmp(MOD,'QPSK') == true
        % OTFS - QPSK
        numBits = ceil((numSubcarriers*2*numSymbols-2*L)/(2*n-1));
        encoderIdx = randperm(numBits);
        inputBits = zeros(numBits, numTxAnt);
        turboEncoder = comm.TurboEncoder(trellis,encoderIdx);
        pskModulator = comm.PSKModulator(4, pi/4);
        for l = 1 : numTxAnt
            inputBits(:,l) = randi([0 1],numBits,1);
            encodedData = turboEncoder(inputBits(:,l));
            encodedBits = reshape(encodedData,2,length(encodedData)/2).';
            encodedBits = binaryVectorToDecimal(encodedBits);
            modSym(:,l) = pskModulator(encodedBits);
        end
        payload = modSym(1:numSubcarriers*numSymbols,:);

    elseif strcmp(MOD,'16QAM')==true
        % OTFS - 16QAM
        numBits = ceil((numSubcarriers*numSymbols*4-2*L)/(2*n-1))+1;
        encoderIdx = randperm(numBits);
        turboEncoder = comm.TurboEncoder(trellis,encoderIdx);
        inputBits = zeros(numBits,numTxAnt);
        for l = 1:numTxAnt
            inputBits(:,l) = randi([0 1],numBits,1);
            encodedData = turboEncoder(inputBits(:,l));
            encodedBits = reshape(encodedData,4,length(encodedData)/4).';
            encodedBits = binaryVectorToDecimal(encodedBits);
            modSym(:,l) = 1/sqrt(10)*qammod(encodedBits,16);
        end
        payload = modSym(1:numSubcarriers*numSymbols,:);

    else
        % OTFS - 64QAM
        numBits = ceil((numSubcarriers*numSymbols*6-2*L)/(2*n-1));
        encoderIdx = randperm(numBits);
        turboEncoder = comm.TurboEncoder(trellis,encoderIdx);
        inputBits = zeros(numBits,numTxAnt);
        for l = 1:numTxAnt
            inputBits(:,l) = randi([0 1],numBits,1);
            encodedData = turboEncoder(inputBits(:,l));
            encodedBits = reshape(encodedData,6,length(encodedData)/6).';
            encodedBits = binaryVectorToDecimal(encodedBits);
            modSym(:,l) = 1/sqrt(42)*qammod(encodedBits,64);
        end
        payload = modSym(1:numSubcarriers*numSymbols,:,:);

    end

elseif strcmp(Method,'OFDM')==true

    ZP = varargin{1};

    if strcmp(MOD,'QPSK')==true
        % OFDM - QPSK
        numBits = ceil(((ZP)*2-2*L)/(2*n-1));
        encoderIdx = randperm(numBits);
        turboEncoder = comm.TurboEncoder(trellis,encoderIdx);
        inputBits = zeros(numBits,numTxAnt);
        pskModulator = comm.PSKModulator(4,pi/4);
        for l = 1:numTxAnt
            inputBits(:,l) = randi([0 1],numBits,1);
            encodedData = turboEncoder(inputBits(:,l));
            encodedBits = reshape(encodedData,2,length(encodedData)/2).';
            encodedBits = binaryVectorToDecimal(encodedBits);
            modSym(:,l) = pskModulator(encodedBits);
        end
        payload = modSym(1:ZP,:);

    elseif strcmp(MOD,'16QAM')==true
        % OFDM - 16QAM
        numBits = ceil(((ZP)*4-2*L)/(2*n-1));
        encoderIdx = randperm(numBits);
        turboEncoder = comm.TurboEncoder(trellis,encoderIdx);
        inputBits = zeros(numBits,numTxAnt);
        for l = 1:numTxAnt
            inputBits(:,l) = randi([0 1],numBits,1);
            encodedData = turboEncoder(inputBits(:,l));
            encodedBits = reshape(encodedData,4,length(encodedData)/4).';
            encodedBits = binaryVectorToDecimal(encodedBits);
            modSym(:,l) = 1/sqrt(10)*qammod(encodedBits,16);
        end
        payload = modSym(1:ZP,:);

    else
        % OFDM - 64 QAM
        numBits = ceil(((ZP)*6-2*L)/(2*n-1));
        encoderIdx = randperm(numBits);
        turboEncoder = comm.TurboEncoder(trellis,encoderIdx);
        inputBits = zeros(numBits,numTxAnt);
        for l = 1:numTxAnt
            inputBits(:,l) = randi([0 1],numBits,1);
            encodedData = turboEncoder(inputBits(:,l));
            encodedBits = reshape(encodedData,6,length(encodedData)/6).';
            encodedBits = binaryVectorToDecimal(encodedBits);
            modSym(:,l) = 1/sqrt(42)*qammod(encodedBits,64);
        end
        payload = modSym(1:ZP,:);

    end
```

```
else

    ZP = varargin{1};
    if strcmp(MOD,'QPSK')==true
        % ZP - QPSK
        numBits = ceil(((numSubcarriers-ZP)*numSymbols*2-2*L)/(2*n-1));
        encoderIdx = randperm(numBits);
        turboEncoder = comm.TurboEncoder(trellis,encoderIdx);
        inputBits = zeros(numBits,numTxAnt);
        pskModulator = comm.PSKModulator(4,pi/4);
        for l = 1:numTxAnt
            inputBits(:,l) = randi([0 1],numBits,1);
            encodedData = turboEncoder(inputBits(:,l));
            encodedBits = reshape(encodedData,2,length(encodedData)/2).';
            encodedBits = binaryVectorToDecimal(encodedBits);
            modSym(:,l) = pskModulator(encodedBits);
        end
        payload = modSym(1:(numSubcarriers-ZP)*numSymbols,:);

    elseif strcmp(MOD,'16QAM')==true
        % ZP - 16QAM
        numBits = ceil(((numSubcarriers-ZP)*numSymbols*4-2*L)/(2*n-1));
        encoderIdx = randperm(numBits);
        turboEncoder = comm.TurboEncoder(trellis,encoderIdx);
        inputBits = zeros(numBits,numTxAnt);
        for l = 1:numTxAnt
            inputBits(:,l) = randi([0 1],numBits,1);
            encodedData = turboEncoder(inputBits(:,l));
            encodedBits = reshape(encodedData,4,length(encodedData)/4).';
            encodedBits = binaryVectorToDecimal(encodedBits);
            modSym(:,l) = 1/sqrt(10)*qammod(encodedBits,16);
        end
        payload = modSym(1:(numSubcarriers-ZP)*numSymbols,:);

    else
        % ZP - 64QAM
        numBits = ceil(((numSubcarriers-ZP)*numSymbols*6-2*L)/(2*n-1));
        encoderIdx = randperm(numBits);
        turboEncoder = comm.TurboEncoder(trellis,encoderIdx);
        inputBits = zeros(numBits,numTxAnt);
        for l = 1:numTxAnt
            inputBits(:,l) = randi([0 1],numBits,1);
            encodedData = turboEncoder(inputBits(:,l));
            encodedBits = reshape(encodedData,6,length(encodedData)/6).';
            encodedBits = binaryVectorToDecimal(encodedBits);
            modSym(:,1) = 1/sqrt(42)*qammod(encodedBits,64);
        end
        payload = modSym(1:(numSubcarriers-ZP)*numSymbols,:);

    end

end
```

C.4 generate_transmitted_signal.m

This Matlab code represents a function generate_transmitted_signal() which is designed to generate an Orthogonal Frequency Division Multiplexing (OFDM) signal for 5G New Radio (NR) Frequency Range 2 (FR2).

This function simulates the transmission of an OFDM signal over a 5G NR channel with parameters like bandwidth, transmit power, subcarrier spacing, number of data subcarriers, and number of OFDM symbols.

The code is given by:

```
function [Tx_Sig_OFDM_CP] = generate_transmitted_signal()
%% Parameter for 5G NR FR2 %%
numTxAnt = 1;                      % Number of transmit antenna
numRxAnt = 1;                      % Number of receive antenna
numSubcarriers = 1024;             % number of data subcarriers %
numSymbols = 8;                    % Number of OFDM Symbol %
bandWidth = 20e6;                  % Bandwidth %
carrierSpace = 15e3;               % subcarrier spacing: 15 KHz %

txPowBs_dBm = 23;                  % Transmit power of BS %
txPowUe_dBm = 0;                   % Transmit power of UE %

symbolDrt = 1/carrierSpace;      % one time symbol duration in OTFS frame %
zeroPad = numSubcarriers/16;                    % Zero padded %
dftMatrix = dftmtx(numSymbols);                 % Generate the DFT matrix
dftMatrix = dftMatrix./norm(dftMatrix);         % Normalize the DFT matrix
noiseAmp = 10^((-174+10*log10(bandWidth))/20);  % Noise amplitude %

%% Turbo Code Setting
```

```
trellis = poly2trellis(4,[13 15],13); % Trellis for Turbo Code %
n = log2(trellis.numOutputSymbols);
L = log2(trellis.numStates)*n;
nitr = 4;
it = 1; Fit = 0;

%% Chose modulation %%
mod = '16QAM';

%% Paylaod Data %%
plt = [];
for l = 1 : 2 : numSymbols
    p = (0:8:numSubcarriers-zeroPad-1)+((l-1)*(numSubcarriers-zeroPad)+1);
    plt = [plt p];
end
Idplt_DL(:,1) = plt.';Idplt_DL(:,2) = plt.'+1;
Idplt_UL(:,1) = plt.'+2;Idplt_UL(:,2) = plt.'+3;
Idsym = setdiff(1:(numSubcarriers-zeroPad)*numSymbols,[Idplt_DL(:);Idplt_UL(:)])
    ;

[x_ofdm,b_ofdm,s_ofdm,idx_ofdm] = fun_codedPayloadGen...
    (mod,numSymbols,numSubcarriers,numTxAnt,trellis,n,L,'OFDM',length(Idsym));

%% OFDM Modulation DL %%
Tx_Sig_OFDM_CP = zeros(numSubcarriers,numSymbols+1,numTxAnt);
Tx_Sig_OFDM = zeros(numSubcarriers-zeroPad,numSymbols,numTxAnt);

for k=1:numTxAnt
    x = zeros((numSubcarriers-zeroPad)*numSymbols,1);
    x(Idplt_DL(:,k)) = 2;
    x(Idsym) = x_ofdm(:,k);
    Tx_Sig_OFDM(:,:,k) = reshape(x,numSubcarriers-zeroPad,numSymbols);
    for l=1:numSymbols
        Tx = sqrt(length(Tx_Sig_OFDM))*ifft(Tx_Sig_OFDM(:,l,k));
        Tx_Sig_OFDM_CP(:,l,k) = [Tx(end-zeroPad+1:end);Tx];
    end
end
Tx_Sig_OFDM_CP = Tx_Sig_OFDM_CP(:);
Tx_Sig_OFDM_CP = reshape(Tx_Sig_OFDM_CP,length(Tx_Sig_OFDM_CP)/numTxAnt,numTxAnt
    );
end
```

C.5 Tx_distortion.m

This Matlab function named Tx_distortion() is designed to distort a transmitted Orthogonal Frequency Division Multiplexing (OFDM) signal by applying a specified nonlinear function to it.

The function takes two arguments:

- 'Tx_Sig_OFDM_CP': The transmitted OFDM signal after the Cyclic Prefix (CP) has been added.
- 'Nonlinear_function': A nonlinear function that is applied to each element of the transmitted signal.

The function loops through each element of Tx_Sig_OFDM_CP and applies the Nonlinear_function to it, effectively distorting the signal. This could be used to simulate certain effects in a communication system, such as amplifier non-linearity or distortion introduced by the transmission channel. The distorted signal, Distorted_Tx_Sig, is then returned.

The code is given by:

```
function [Distorted_Tx_Sig] = Tx_distortion(Tx_Sig_OFDM_CP,Nonlinear_function)
    x = Tx_Sig_OFDM_CP;
    for k = 1 : length(x)
        x(k) = Nonlinear_function(x(k));
    end
    Distorted_Tx_Sig = x;
end
```

C.6 R_square.m

The given Matlab function, R_square(), computes the R-squared (coefficient of determination) statistic(which is also NMSE) between the target and prediction arrays. The coefficient of determination can take values in the range (-∞, 1] according to the mutual relation between the ground truth and the prediction model.[27] Higher values represent a better model fit.

The function takes two input parameters:

- target: This is the true or observed values of the target variable (also known as the dependent variable).
- prediction: This is the predicted values of the target variable, produced by a regression model.

The code is given by:

```
function R = R_square(target , prediction)
    meanValue = mean(target);
    T1 = 0;
    T2 = 0;
    for i = 1:length(target)
        T1 = T1 + (target(i) - prediction(i))^2;
        T2 = T2 + (target(i) - meanValue)^2;
    end
    R = (1 - (T1 / T2)) * 100;
end
```

C.7 MSE.m

The given MATLAB function, MSE(), calculates the Mean Squared Error (MSE) for both the real and imaginary components of a given signal. The code is given by:

```
function [Real_MSE,Imag_MSE] = MSE(true_y,predict_y)
    Real_MSE = 0;
    Imag_MSE = 0;
    Real_true_y = real(true_y);
    Imag_true_y = imag(true_y);
    Real_predict_y = real(predict_y);
    Imag_predict_y = imag(predict_y);

    for i = 1:length(true_y)
        Real_MSE = Real_MSE + (((Real_predict_y(i)-Real_true_y(i)).^2)/length(
            true_y));
        Imag_MSE = Imag_MSE + (((Imag_predict_y(i)-Imag_true_y(i)).^2)/length(
            true_y));
    end

end
```

C.8 NARMA10.m

This MATLAB function NARMA10(X) implements a 10th-order Nonlinear AutoRegressive Moving Average (NARMA) model. The NARMA model is used for creating benchmarks in the field of system identification and control. The code is given by:

```
function y = NARMA10(X)
    % Input:
    % X : Input sequence
    %
    % Output:
    % y : Output sequence of NARMA10 system
    %
    % Description:
    % This function calculates the NARMA10 output for a given input X

    % Initialize parameters
    n = length(X);
    y = zeros(n, 1);

    % NARMA10 computation
    for t = 1:10
        y_1 = 0;
        y_sum = 0;
        if t-1 >= 1
            y_1 = y(t-1);
        end

        for i = 1:10
            if t-i >= 1
                y_sum = y_sum + y(t-i);
            else
                break;
            end
        end
        y(t) = 0.3*y_1 + 0.05*y_1*y_sum + 0.1;
    end

    for t = 11:n
        y(t) = 0.3*y(t-1) + 0.05*y(t-1)*sum(y(t-1:t-10)) + 1.5*X(t-1)*X(t-10) +
            0.1;
    end
end
```

C.9 gridSearch.m

This MATLAB function, gridSearch, is used to perform a grid search of hyperparameters for a reservoir computing model.

The function takes as input multiple sets of hyperparameters (like the number of reservoir nodes, non-linear functions, alpha and beta values, MS values, MNN values, and so on) and employs the K-fold cross-validation method to evaluate each combination of hyperparameters.

The function calculates an error measure (based on either a NARMA10 model or a simulation model) for each fold and averages this to give the overall error for that particular combination of hyperparameters. It keeps track of the combination that gives the lowest error, and after it has tried all combinations, it returns the best reservoir, the best parameters, and the minimum error.

The code is given by:

```
function [best_reservoir, best_params, min_error] = gridSearch(N_set,
    non_linear_funcs, alpha_beta_set, MS_values, MNN_values, folds,
    output_gen_method)

min_error = inf;
best_params = [];
best_reservoir = [];
tdl = nrTDLChannel('DelayProfile','TDL-A','DelaySpread',100e-9,'SampleRate'
    ,50e6,'MaximumDopplerShift',0,'NumReceiveAntennas',1);
for n = 1 : length(N_set)
    for f = 1 : length(non_linear_funcs)
        for a = 1 : length(alpha_beta_set)
            for ms = 1 : length(MS_values)
                for mnn = 1 : length(MNN_values)
                    reservoir = Reservoir(N_set(n), non_linear_funcs{f},
                        alpha_beta_set(a,1), alpha_beta_set(a,2), MS_values(
                        ms), MNN_values(mnn));
                    error_sum = 0;
                    error = 0;
                    X = [generate_transmitted_signal();
                        generate_transmitted_signal();
                        generate_transmitted_signal()];
                    cv = cvpartition(length(X),'KFold',folds);
                    for i = 1:folds
                        trIdx = cv.training(i);
                        teIdx = cv.test(i);
                        X_train = X(trIdx);
                        y_train = [];
                        X_test = X(teIdx);
                        y_test = [];
                        if strcmp(output_gen_method, 'NARMA10')
                            y_train = NARMA10(X_train);
                            y_test = NARMA10(X_test);
                        elseif strcmp(output_gen_method, 'Simulation')
```

```
                                y_train = Tx_distortion(X_train,@(x) x+0.036*x
                                    .^2 - 0.011*x.^3);
                                y_train = tdl(y_train);
                                y_train = Tx_distortion(y_train,@(x) x+0.036*x
                                    .^2 - 0.011*x.^3) - X_train;
                                y_test = Tx_distortion(X_test,@(x) x+0.036*x.^2
                                    - 0.011*x.^3);
                                y_test = tdl(y_test);
                                y_test = Tx_distortion(y_test,@(x) x+0.036*x.^2
                                    - 0.011*x.^3);
                            else
                                error('Invalid output generating method')
                            end
                            reservoir = reservoir.fit(X_train, y_train, 0);

                            prediction = reservoir.predict(X_test);

                            if strcmp(output_gen_method, 'Simulation')
                                prediction = y_test - prediction;
                            end
                            if strcmp(output_gen_method, 'Simulation')
                                error = mean((abs(real(prediction - X_test)) +
                                    abs(imag(prediction - X_test)))./2);
                            elseif strcmp(output_gen_method,'NARMA10')
                                error = mean((abs(real(prediction - y_test)) +
                                    abs(imag(prediction - y_test)))./2);
                            end
                            error_sum = error_sum + error/folds;
                        end

                        if error_sum < min_error
                            min_error = error_sum;
                            best_params = [N_set(n), non_linear_funcs(f),
                               alpha_beta_set(a,1), alpha_beta_set(a,2),
                               MS_values(ms), MNN_values(mnn)];
                            best_reservoir = reservoir;
                        end
                    end
                end
            end
        end
    end
end
```

C.10 L1_Norm.m

The given MATLAB function L1_Norm calculates the L1 norm error, also known as the mean absolute error, between the target and the predicted signals separately for the real and imaginary components. The L1 norm is a measure of the total absolute differences between two vectors, in this case, the target signal and the predicted signal.

In this function, the real and imaginary parts of the target and predicted signals are first separated. The absolute difference between each corresponding pair of real and imaginary parts from the target and predicted signals is then calculated and summed up to obtain the total real and imaginary errors respectively. These total errors are then normalized by the sum of the absolute values of the target's real and imaginary components to obtain the normalized mean absolute error (L1 norm) for the real and imaginary components separately.

The code is given by:

```
function [RE_L1, IM_L1] = L1_Norm(target, prediction)
    target_RE = real(target);
    target_IM = imag(target);
    prediction_RE = real(prediction);
    prediction_IM = imag(prediction);
    Total_RE_error = 0;
    Total_IM_error = 0;
    for j = 1 : length(target)
        Total_RE_error = Total_RE_error + abs(prediction_RE(j) - target_RE(j));
        Total_IM_error = Total_IM_error + abs(prediction_IM(j) - target_IM(j));
    end
    RE_L1 = Total_RE_error/sum(abs(target_RE));
    IM_L1 = Total_IM_error/sum(abs(target_IM));
end
```

C.11 linear_memory_capacity.m

The provided MATLAB function linear_memory_capacity is used to calculate and visualize the linear memory capacity (LMC) for different models of a reservoir computing system.

Linear memory capacity is a measure of a system's ability to remember past input information. In this function, five different reservoir models are created, each having different parameters such as the size of the reservoir (N), the function used to calculate the output of each neuron in the reservoir (sinh(x)/(1+exp(-x)) in this case), and the masking and neuron normalization values (MS and MNN, respectively). For each model, the function runs a loop from initial_k to end_k to simulate the system for different memory lengths.

In each iteration of this loop, training and test datasets are generated and used to train and test each reservoir model. The LMC is then calculated for each model by comparing the system's output to a delayed version of the input signal. This process is repeated rep number of times for statistical averaging.

Finally, the function plots the LMC for each model, both for the real and imaginary parts of the model's output. These plots help visualize how the memory capacity of each model changes with the memory length (k).

The code is given by:

```
function [real_LMC_array,imag_LMC_array] = linear_memory_capacity(initial_k,
    end_k,rep)

    real_LMC_array = zeros(5,end_k-initial_k+1,1);
    imag_LMC_array = zeros(5,end_k-initial_k+1,1);
    X_train = zeros(rep,9216,1);
    X_test = zeros(rep,9216,1);
    y_train = zeros(rep,9216,1);
    y_test = zeros(rep,9216,1);
    reservoir = Reservoir(50, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 1, 1);
    reservoir1 = Reservoir(200, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 1, 1);
    reservoir2 = Reservoir(850, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 1, 25);
    reservoir3 = Reservoir(850, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 25, 1);
    reservoir4 = Reservoir(850, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 25, 15);
    for i = 1:rep
        X_train(i,:,:) = generate_transmitted_signal();
        X_test(i,:,:) = generate_transmitted_signal();
    end
    all_reservoirs = [reservoir; reservoir1; reservoir2; reservoir3; reservoir4
        ];
    for i = initial_k : end_k
        for r = 1:rep
            for j = 1 : length(X_train(r,:)')
                if j <= i
                    y_train(r,j,1) = 0;
                    y_test(r,j,1) = 0;
                else
                    y_train(r,j,1) = X_train(r,j-i,1);
                    y_test(r,j,1) = X_test(r,j-i,1);
                end
            end

            for l = 1:5
                reservoir = all_reservoirs(l).fit(X_train(r,:)',y_train(r,:)',1e
                    -4);
                prediction = reservoir.predict(X_test(r,:)');
                real_LMC_array(l,i-initial_k+1) = real_LMC_array(l,i-initial_k
                    +1) + R_square(real(y_test(r,:)'),real(prediction))/(100*rep
                    );
                imag_LMC_array(l,i-initial_k+1) = imag_LMC_array(l,i-initial_k
                    +1) + R_square(imag(y_test(r,:)'),imag(prediction))/(100*rep
                    );
            end

        end

    end
    x = 1:end_k-initial_k+1;
    figure;
    plot(x,real_LMC_array(1,:),'k-o', 'DisplayName', 'N=50,MS=1,MNN=1');
    hold on;
    plot(x,real_LMC_array(2,:),'b-*', 'DisplayName', 'N=200,MS=1,MNN=1');
    hold on;
    plot(x,real_LMC_array(3,:),'c-^', 'DisplayName', 'N=850,MS=1,MNN=25');
    hold on;
    plot(x,real_LMC_array(4,:),'g-d', 'DisplayName', 'N=850,MS=25,MNN=1');
    hold on;
    plot(x,real_LMC_array(5,:),'r-s', 'DisplayName', 'N=850,MS=25,MNN=15');
    legend();
    title('LMC for real part of Model : y(n) = u(n-k)');
    xlabel('k');
    ylabel('Linear memory capacities');
    xlim([1 end_k-initial_k+1]);
    ylim([0 1.1]);
    hold off;

    figure;
    plot(x,imag_LMC_array(1,:),'k-o', 'DisplayName', 'N=50,MS=1,MNN=1');
    hold on;
    plot(x,imag_LMC_array(2,:),'b-*', 'DisplayName', 'N=200,MS=1,MNN=1');
    hold on;
    plot(x,imag_LMC_array(3,:),'c-^', 'DisplayName', 'N=850,MS=1,MNN=25');
    hold on;
```

```
    plot(x,imag_LMC_array(4,:),'g-d', 'DisplayName', 'N=850,MS=25,MNN=1');
    hold on;
    plot(x,imag_LMC_array(5,:),'r-s', 'DisplayName', 'N=850,MS=25,MNN=15');
    legend();
    title('LMC for imaginary part of Model : y(n) = u(n-k)');
    xlabel('k');
    ylabel('Linear memory capacities');
    xlim([1 end_k-initial_k+1]);
    ylim([0 1.1]);
    hold off;
end
```

C.12 quadratic_memory_capacity.m

The provided MATLAB function, quadratic_memory_capacity, is used to calculate and visualize the Quadratic Memory Capacity (QMC) for different models of a reservoir computing system.

Quadratic Memory Capacity is another measure of a system's ability to remember past input information. This measure is similar to Linear Memory Capacity, but instead of merely reproducing the past inputs, the system attempts to reproduce the square of past inputs.

This function creates five different reservoir models, similar to the earlier function for calculating LMC, each with different parameters such as the size of the reservoir (N), the output calculation function, and masking and neuron normalization values (MS and MNN). For each model, the function runs a loop from initial_k to end_k to simulate the system for different memory lengths.

In each iteration, training and test datasets are generated and used to train and test each reservoir model. The QMC is then calculated for each model by comparing the system's output to the squared version of a delayed input signal. This process is repeated rep number of times for statistical averaging.

Finally, the function plots the QMC for both real and imaginary parts of each model's output. These plots provide a visual representation of how the memory capacity of each model changes with the memory length (k).

The code is given by:

```
    function [real_QMC_array,imag_QMC_array] = quadratic_memory_capacity(
        initial_k,end_k,rep)
    real_QMC_array = zeros(5,end_k-initial_k+1,1);
    imag_QMC_array = zeros(5,end_k-initial_k+1,1);
    X_train = zeros(rep,9216,1);
```

```
X_test = zeros(rep,9216,1);
y_train = zeros(rep,9216,1);
y_test = zeros(rep,9216,1);
reservoir = Reservoir(50, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 1, 1);
reservoir1 = Reservoir(200, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 1, 1);
reservoir2 = Reservoir(850, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 1, 25);
reservoir3 = Reservoir(850, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 25, 1);
reservoir4 = Reservoir(850, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 25, 15);
for i = 1:rep
    X_train(i,:,:) = generate_transmitted_signal();
    X_test(i,:,:) = generate_transmitted_signal();
end
all_reservoirs = [reservoir; reservoir1; reservoir2; reservoir3; reservoir4
   ];
for i = initial_k : end_k
    for r = 1:rep
        for j = 1 : length(X_train(r,:)')
            if j <= i
                y_train(r,j,1) = 0;
                y_test(r,j,1) = 0;
            else
                y_train(r,j,1) = X_train(r,j-i,1).^2;
                y_test(r,j,1) = X_test(r,j-i,1).^2;
            end
        end

        for l = 1:5
            reservoir = all_reservoirs(l).fit(X_train(r,:)',y_train(r,:)',1e
                -4);
            prediction = reservoir.predict(X_test(r,:)');
            real_QMC_array(l,i-initial_k+1) = real_QMC_array(l,i-initial_k
                +1) + R_square(real(y_test(r,:)'),real(prediction))/(100*rep
                );
            imag_QMC_array(l,i-initial_k+1) = imag_QMC_array(l,i-initial_k
                +1) + R_square(imag(y_test(r,:)'),imag(prediction))/(100*rep
                );
        end

    end

end
x = 1:end_k-initial_k+1;
figure;
```

```
    plot(x,real_QMC_array(1,:),'k-o', 'DisplayName', 'N=50,MS=1,MNN=1');
    hold on;
    plot(x,real_QMC_array(2,:),'b-*', 'DisplayName', 'N=200,MS=1,MNN=1');
    hold on;
    plot(x,real_QMC_array(3,:),'c-^', 'DisplayName', 'N=850,MS=1,MNN=25');
    hold on;
    plot(x,real_QMC_array(4,:),'g-d', 'DisplayName', 'N=850,MS=25,MNN=1');
    hold on;
    plot(x,real_QMC_array(5,:),'r-s', 'DisplayName', 'N=850,MS=25,MNN=15');
    legend();
    title('QMC for real part of Model : y(n) = u(n-k)^2');
    xlabel('k');
    ylabel('Quadratic memory capacities');
    xlim([1 end_k-initial_k+1]);
    ylim([0 1.1]);
    hold off;

    figure;
    plot(x,imag_QMC_array(1,:),'k-o', 'DisplayName', 'N=50,MS=1,MNN=1');
    hold on;
    plot(x,imag_QMC_array(2,:),'b-*', 'DisplayName', 'N=200,MS=1,MNN=1');
    hold on;
    plot(x,imag_QMC_array(3,:),'c-^', 'DisplayName', 'N=850,MS=1,MNN=25');
    hold on;
    plot(x,imag_QMC_array(4,:),'g-d', 'DisplayName', 'N=850,MS=25,MNN=1');
    hold on;
    plot(x,imag_QMC_array(5,:),'r-s', 'DisplayName', 'N=850,MS=25,MNN=15');
    legend();
    title('QMC for imaginary part of Model : y(n) = u(n-k)^2');
    xlabel('k');
    ylabel('Quadratic memory capacities');
    xlim([1 end_k-initial_k+1]);
    ylim([0 1.1]);
    hold off;

end
```

C.13 cross_memory_capacity.m

The provided MATLAB function, cross_memory_capacity, is designed to calculate and visualize the Cross Memory Capacity (CMC) for different models of a reservoir computing system.

The Cross Memory Capacity is a measure of a system's ability to recall cross-correlations between past input signals. In this case, the system attempts to reproduce the multiplication of past inputs (i.e., u(n-k)u(n-1)).

Similar to the previous functions, five reservoir models are created with different parameters such as the size of the reservoir (N), the output calculation function, and masking and neuron normalization values (MS and MNN).

The function runs a loop from initial_k to end_k, generating training and testing datasets for each iteration. It then trains and tests each reservoir model. The CMC is calculated for each model by comparing the system's output to the multiplication of two delayed input signals.

This process is repeated rep number of times to average out the results, reducing the impact of randomness and improving the reliability of the result.

Finally, the function plots the CMC for both the real and imaginary parts of each model's output. These plots provide a visual representation of how the cross memory capacity of each model changes with the memory length (k).

The code is given by:

```
function [real_CMC_array,imag_CMC_array] = cross_memory_capacity(initial_k,end_k
    ,rep)

    real_CMC_array = zeros(5,end_k-initial_k+1,1);
    imag_CMC_array = zeros(5,end_k-initial_k+1,1);
    y_train = zeros(rep,9216,1);
    y_test = zeros(rep,9216,1);
    X_train = zeros(rep,9216,1);
    X_test = zeros(rep,9216,1);

    reservoir = Reservoir(50, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 1, 1);
    reservoir1 = Reservoir(200, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 1, 1);
    reservoir2 = Reservoir(850, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 1, 25);
    reservoir3 = Reservoir(850, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 25, 1);
    reservoir4 = Reservoir(850, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 25, 15);
    for i = 1:rep
        X_train(i,:,:) = generate_transmitted_signal();
        X_test(i,:,:) = generate_transmitted_signal();
    end
    all_reservoirs = [reservoir; reservoir1; reservoir2; reservoir3; reservoir4
        ];
    for i = initial_k : end_k
        for r = 1:rep
            for j = 1 : length(X_train(r,:)')
                if j <= i
                    y_train(r,j,1) = 0;
                    y_test(r,j,1) = 0;
                else
                    y_train(r,j,1) = X_train(r,j-i,1)*X_train(r,j-1,1);
                    y_test(r,j,1) = X_test(r,j-i,1)*X_test(r,j-1,1);
                end
            end

            for l = 1:5
                reservoir = all_reservoirs(l).fit(X_train(r,:)',y_train(r,:)',1e
                    -4);
                prediction = reservoir.predict(X_test(r,:)');
                real_CMC_array(l,i-initial_k+1) = real_CMC_array(l,i-initial_k
                    +1) + R_square(real(y_test(r,:)'),real(prediction))/(100*rep
                    );
                imag_CMC_array(l,i-initial_k+1) = imag_CMC_array(l,i-initial_k
                    +1) + R_square(imag(y_test(r,:)'),imag(prediction))/(100*rep
                    );
            end

        end

    end
    x = 1:end_k-initial_k+1;
    figure;
    plot(x,real_CMC_array(1,:),'k-o', 'DisplayName', 'N=50,MS=1,MNN=1');
    hold on;
    plot(x,real_CMC_array(2,:),'b-*', 'DisplayName', 'N=200,MS=1,MNN=1');
    hold on;
    plot(x,real_CMC_array(3,:),'c-^', 'DisplayName', 'N=850,MS=1,MNN=25');
    hold on;
    plot(x,real_CMC_array(4,:),'g-d', 'DisplayName', 'N=850,MS=25,MNN=1');
    hold on;
    plot(x,real_CMC_array(5,:),'r-s', 'DisplayName', 'N=850,MS=25,MNN=15');
    legend();
    title('CMC for real part of Model : y(n) = u(n-k)u(n-1)');
    xlabel('k');
    ylabel('Cross memory capacities');
```

```
    xlim([1 end_k-initial_k+1]);
    ylim([0 1.1]);
    hold off;

    figure;
    plot(x,imag_CMC_array(1,:),'k-o', 'DisplayName', 'N=50,MS=1,MNN=1');
    hold on;
    plot(x,imag_CMC_array(2,:),'b-*', 'DisplayName', 'N=200,MS=1,MNN=1');
    hold on;
    plot(x,imag_CMC_array(3,:),'c-^', 'DisplayName', 'N=850,MS=1,MNN=25');
    hold on;
    plot(x,imag_CMC_array(4,:),'g-d', 'DisplayName', 'N=850,MS=25,MNN=1');
    hold on;
    plot(x,imag_CMC_array(5,:),'r-s', 'DisplayName', 'N=850,MS=25,MNN=15');
    legend();
    title('CMC for imaginary part of Model : y(n) = u(n-k)u(n-1)');
    xlabel('k');
    ylabel('Cross memory capacities');
    xlim([1 end_k-initial_k+1]);
    ylim([0 1.1]);
    hold off;

end
```

C.14 NARMA10_test.m

The provided MATLAB code defines a function NARMA10_test, which evaluates the performance of different Reservoir Computing (RC) models on the task of modeling a Nonlinear AutoRegressive Moving Average (NARMA) system of order 10.

The NARMA10 model is a commonly used benchmark in testing the performance of recurrent neural networks. It is a discrete-time nonlinear system that represents a 10th order time series.

In the script, the RC models are defined by different parameters - reservoir size (N), masking value (MS), and mean neuron normalization value (MNN).

The function iterates through various configurations of the RC models, fits the model to the training data and then tests the model on the test data. There will be repeats for each configuration(each repeat with different training and testing data) and the number of repeats is determined by the parameter "rep". For each configuration, the averaged Mean Squared Error (MSE) is computed and stored in an array.

Finally, the averaged MSEs are plotted against the respective parameters, providing a visual comparison of the performance of different models.

The code is given by:

```
function [real_N_MSE,real_MS_MSE,real_MNN_MSE,imag_N_MSE,imag_MS_MSE,
    imag_MNN_MSE] = NARMA10_test(rep)

real_N_MSE = zeros(17,1);
real_MS_MSE = zeros(25,1);
real_MNN_MSE = zeros(15,1);
imag_N_MSE = zeros(17,1);
imag_MS_MSE = zeros(25,1);
imag_MNN_MSE = zeros(15,1);
X_train = zeros(rep,9216,1);
X_test = zeros(rep,9216,1);
y_train = zeros(rep,9216,1);
y_test = zeros(rep,9216,1);

for i = 1:rep
    X_train(i,:,:) = generate_transmitted_signal();
    X_test(i,:,:) = generate_transmitted_signal();
    y_train(i,:,:) = NARMA10(X_train(i,:,:));
    y_test(i,:,:) = NARMA10(X_test(i,:,:));
end

for i = 1:17
    reservoir = Reservoir(50*i, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 1, 1);
    for j = 1:rep
        reservoir = reservoir.fit(X_train(j,:,:)',y_train(j,:,:)');
        prediction = reservoir.predict(X_test(j,:,:)');
        [Real_MSE,Imag_MSE] = MSE(y_test(j,:,:)',prediction);
        real_N_MSE(i) = real_N_MSE(i) + Real_MSE/rep;
        imag_N_MSE(i) = imag_N_MSE(i) + Imag_MSE/rep;
    end
end

reservoir = Reservoir(850, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 1, 15);
for i = 1:25
    reservoir.MS = i;
    for j = 1:rep
        reservoir = reservoir.fit(X_train(j,:,:)',y_train(j,:,:)');
        prediction = reservoir.predict(X_test(j,:,:)');
        [Real_MSE,Imag_MSE] = MSE(y_test(j,:,:)',prediction);
        real_MS_MSE(i) = real_MS_MSE(i) + Real_MSE/rep;
        imag_MS_MSE(i) = imag_MS_MSE(i) + Imag_MSE/rep;
    end
end

reservoir = Reservoir(850, @(x) (sinh(x)/(1+exp(-x))), 0.5, 0.3, 25, 1);
for i = 1:15
    reservoir.MNN = i;
    for j = 1:rep
        reservoir = reservoir.fit(X_train(j,:,:)',y_train(j,:,:)');
        prediction = reservoir.predict(X_test(j,:,:)');
        [Real_MSE,Imag_MSE] = MSE(y_test(j,:,:)',prediction);
        real_MNN_MSE(i) = real_MNN_MSE(i) + Real_MSE/rep;
        imag_MNN_MSE(i) = imag_MNN_MSE(i) + Imag_MSE/rep;
    end
end

x = 50:50:850;
x1 = 1:25;
x2 = 1:15;
figure;
plot(x,real_N_MSE,'k-o', 'DisplayName', 'MS=1,MNN=1');
legend();
title('Real part of MSE for NARMA10');
xlabel('N');
ylabel('MSE');
hold off;

figure;
plot(x,imag_N_MSE,'k-o', 'DisplayName', 'MS=1,MNN=1');
legend();
title('Imaginary part of MSE for NARMA10');
xlabel('N');
ylabel('MSE');
hold off;

figure;
plot(x1,real_MS_MSE,'r-o', 'DisplayName', 'N=850,MNN=15');
legend();
title('Real part of MSE for NARMA10');
xlabel('MS');
ylabel('MSE');
hold off;
```

```
figure;
plot(x1,imag_MS_MSE,'r-o', 'DisplayName', 'N=850,MNN=15');
legend();
title('Imaginary part of MSE for NARMA10');
xlabel('MS');
ylabel('MSE');
hold off;

figure;
plot(x2,real_MNN_MSE,'b-o', 'DisplayName', 'N=850,MS=25');
legend();
title('Real part of MSE for NARMA10');
xlabel('MNN');
ylabel('MSE');
hold off;

figure;
plot(x2,imag_MNN_MSE,'b-o', 'DisplayName', 'N=850,MS=25');
legend();
title('Imaginary part of MSE for NARMA10');
xlabel('MNN');
ylabel('MSE');
hold off;

end
```

C.15 Prediction_visualization.m

The provided MATLAB code defines a function Prediction_visualization, which is a utility function for visualizing the predictions of two Reservoir Computing (RC) models, along with the actual target values, for a given time sequence.

In the function:

- target, RC1_prediction, and RC2_prediction are input vectors representing the target and prediction values from two different RC models, respectively.
- display_names is an array of string names for the target and two prediction series.
- x_label and y_label are strings that represent labels for the X and Y axes, respectively.
- graph_title is a string that represents the title of the graph.

- time_sequence_begin and time_sequence_end are integers representing the start and end indices of the time sequence to be visualized.

The function plots the target and predicted series on the same graph for comparison. It uses the time sequence as the X-axis and the corresponding target or prediction values as the Y-axis.

The code is given by:

```
function [] = Prediction_visualization(target,RC1_prediction,RC2_prediction,
    display_names,x_label,y_label,graph_title,time_sequence_begin,
    time_sequence_end)
tar = target(time_sequence_begin:time_sequence_end);
RC1_p = RC1_prediction(time_sequence_begin:time_sequence_end);
RC2_p = RC2_prediction(time_sequence_begin:time_sequence_end);
x = time_sequence_begin:time_sequence_end;
figure;
plot(x,tar,'r-o', 'DisplayName', display_names(1));
hold on;
plot(x,RC1_p,'b-*', 'DisplayName', display_names(2));
hold on;
plot(x,RC2_p,'c-^', 'DisplayName', display_names(3));
hold off;
legend();
title(graph_title);
xlabel(x_label);
ylabel(y_label);
end
```

C.16 Simulation.m

This MATLAB code defines a function Simulation, which is used to simulate and calculate the Cancellation Depth of a communication signal processed through a nonlinear Reservoir Computing (RC) model.

In this function:

- rep is an integer representing the number of repetition for the simulation.
- reservoir is an instance of a Reservoir Computing model.

The function uses a tapped delay line (TDL) channel model with TDL-A profile, a delay spread of 100 nanoseconds, a sample rate of 50 MHz, no Doppler shift, and a single receive antenna.

The function generates transmitted signals (X_train and X_test), applies a nonlinear distortion function to them, passes them through the TDL channel, and re-applies the distortion function. The target output (y_train) for the Reservoir Computing model is the difference between the distorted signal and the original transmitted signal.

The RC model is trained with X_train and y_train, and then used to predict the cancellation on the test data X_test. The cancellation is performed by subtracting the predicted signal from the distorted test signal. The function calculates the cancellation depth using the L1 norm method and accumulates it for all the repetitions.

Finally, the function returns the average Cancellation Depth over all the repetitions.

The code is given by:

```
function [Cancellation_depth] = Simulation(rep,reservoir)
tdl = nrTDLChannel('DelayProfile','TDL-A','DelaySpread',100e-9,'SampleRate',50e6
    ,'MaximumDopplerShift',0,'NumReceiveAntennas',1);
Cancellation_depth = 0;
for i = 1:rep
    X_train = generate_transmitted_signal();
    X_test = generate_transmitted_signal();
    y_train = Tx_distortion(X_train,@(x) x + 0.036*x.^2 - 0.011*x.^3);
    y_train = tdl(y_train);
    y_train = Tx_distortion(y_train,@(x) x + 0.036*x.^2 - 0.011*x.^3) - X_train;
    y_test = Tx_distortion(X_test,@(x) x + 0.036*x.^2 - 0.011*x.^3);
    y_test = tdl(y_test);
    y_test = Tx_distortion(y_test,@(x) x + 0.036*x.^2 - 0.011*x.^3);
    reservoir = reservoir.fit(X_train,y_train);
    prediction = y_test - reservoir.predict(X_test);
    [RE_L1, IM_L1] = L1_Norm(X_test, prediction);
    Cancellation_depth = Cancellation_depth + (10*log10(((1/RE_L1 + 1/IM_L1)./2)
       .^2))./rep;
end

end
```

C.17 main.m

This MATLAB script executes a comprehensive simulation of signal processing and prediction using Reservoir Computing (RC). It evaluates various memory capacity tasks, simulates the NARMA10 prediction task, and calculates cancellation depths for nonlinear communication signals, all using RC with different parameters and non-linear functions. Visualizations of prediction tasks are also provided.

The script begins by initializing the workspace and conducting Linear Memory Capacity (LMC), Quadratic Memory Capacity (QMC), and Cross Memory Capacity (CMC) tasks. The NARMA10 task is then simulated, and the Mean Square Error (MSE) is calculated for different parts of the signal (real and imaginary).

The script then generates transmitted signals, applies a nonlinear distortion function to them, and defines several non-linear functions that will be used in reservoir computing models. Different sets of parameters for the RC models are also defined.

The script continues with the visualization of the NARMA10 prediction, using the grid search method to find the best RC model parameters for the task. A series of simulations is then conducted, where the script initializes multiple RC models with different non-linear functions and parameters, and calculates the cancellation depths of signals processed through them.

The script concludes with a visualization of the simulation predictions. The best RC model parameters for the simulation task are determined through a grid search. The script then computes the received signals through a TDL channel and fits the RC models to the received and transmitted signals. The cancellation depth is calculated for each prediction using the L1 norm method, and the prediction results are visualized.

This comprehensive script offers insights into the performance of Reservoir Computing in signal processing and prediction tasks, with variations in model parameters and nonlinear functions. The code is given by:

```
clear all
clc

%Memory capacity tasks
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%LMC tasks
[real_LMC_array,imag_LMC_array] = linear_memory_capacity(1,100,30);

%QMC tasks
[real_QMC_array,imag_QMC_array] = quadratic_memory_capacity(1,100,30);

%CMC tasks
[real_CMC_array,imag_CMC_array] = cross_memory_capacity(1,100,30);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%NARMA10 tasks
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[real_N_MSE,real_MS_MSE,real_MNN_MSE,imag_N_MSE,imag_MS_MSE,imag_MNN_MSE] =
    NARMA10_test(rep);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



Tx_Sig_OFDM_CP = generate_transmitted_signal();
Tx_Sig_OFDM_CP1 = generate_transmitted_signal();

Distorted_Tx_Sig = Tx_distortion(Tx_Sig_OFDM_CP,@(x) x + 0.036*x.^2 - 0.011*x
    .^3);
Distorted_Tx_Sig1 = Tx_distortion(Tx_Sig_OFDM_CP1,@(x) x + 0.036*x.^2 - 0.011*x
    .^3);

F_NL1 = @(x) tanh(x);
F_NL2 = @(x) sin(x);
F_NL3 = @(x) cos(x);
F_NL4 = @(x) 1/(1+exp(-x));
F_NL5 = @(x) sinh(x)/(1+exp(-x));

NLFS = {F_NL1, F_NL2, F_NL3, F_NL4, F_NL5};
alpha_beta_set =
    [0.1,0.8;0.2,0.7;0.3,0.6;0.4,0.5;0.5,0.4;0.6,0.3;0.7,0.2;0.8,0.1];
N_set = 50:50:3000;
MS_set = 1:25;
MNN_set = 1:15;



%NARMA10 prediction visualization
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
y_train = NARMA10(Tx_Sig_OFDM_CP);
y_test = NARMA10(Tx_Sig_OFDM_CP1);

[best_reservoir, best_params, min_error] = gridSearch([50], NLFS, alpha_beta_set
    , [1], [1], 3, 'NARMA10');
[best_reservoir1, best_params1, min_error1] = gridSearch([1000], NLFS,
    alpha_beta_set, [25], [15], 3, 'NARMA10');
```

```
best_reservoir = best_reservoir.fit(Tx_Sig_OFDM_CP,y_train);
prediction = best_reservoir.predict(Tx_Sig_OFDM_CP1);
best_reservoir1 = best_reservoir1.fit(Tx_Sig_OFDM_CP,y_train);
prediction1 = best_reservoir1.predict(Tx_Sig_OFDM_CP1);
Prediction_visualization(real(y_test),real(prediction),real(prediction1),["
    NARMA10 output" "MS=1 and MNN=1" "MS=25 and MNN=15"],'Time sequence n','
    Amplitude','Real part',5800,5850);
Prediction_visualization(imag(y_test),imag(prediction),imag(prediction1),["
    NARMA10 output" "MS=1 and MNN=1" "MS=25 and MNN=15"],'Time sequence n','
    Amplitude','Imaginary part',5800,5850);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%Simulation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

reservoir1 = Reservoir(50,@(x) tanh(x),0.5,0.3,1,1);
reservoir2 = Reservoir(50,@(x) sin(x),0.5,0.3,1,1);
reservoir3 = Reservoir(50,@(x) cos(x),0.5,0.3,1,1);
reservoir4 = Reservoir(50,@(x) 1/(1+exp(-x)),0.5,0.3,1,1);
reservoir5 = Reservoir(50,@(x) sinh(x)/(1+exp(-x)),0.5,0.3,1,1);
[Cancellation_depth1] = Simulation(10,reservoir1);
[Cancellation_depth2] = Simulation(10,reservoir2);
[Cancellation_depth3] = Simulation(10,reservoir3);
[Cancellation_depth4] = Simulation(10,reservoir4);
[Cancellation_depth5] = Simulation(10,reservoir5);


reservoir6 = Reservoir(1800,@(x) tanh(x),0.5,0.3,35,25);
reservoir7 = Reservoir(1800,@(x) sin(x),0.5,0.3,35,25);
reservoir8 = Reservoir(1800,@(x) cos(x),0.5,0.3,35,25);
reservoir9 = Reservoir(1800,@(x) 1/(1+exp(-x)),0.5,0.3,35,25);
reservoir10 = Reservoir(1800,@(x) sinh(x)/(1+exp(-x)),0.5,0.3,35,25);
[Cancellation_depth6] = Simulation(10,reservoir6);
[Cancellation_depth7] = Simulation(10,reservoir7);
[Cancellation_depth8] = Simulation(10,reservoir8);
[Cancellation_depth9] = Simulation(10,reservoir9);
[Cancellation_depth] = Simulation(10,reservoir10);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%Simulation prediction visualization
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[best_reservoir, best_params, min_error] = gridSearch([50], NLFS, alpha_beta_set
    , [1], [1], 3, 'Simulation');
[best_reservoir1, best_params1, min_error1] = gridSearch([1800], NLFS,
    alpha_beta_set, [35], [25], 3, 'Simulation');

tdl = nrTDLChannel('DelayProfile','TDL-A','DelaySpread',100e-9,'SampleRate',50e6
    ,'MaximumDopplerShift',0,'NumReceiveAntennas',1);
tdlinfo = info(tdl);
[signalOut,pathGains,sampleTimes] = tdl(Distorted_Tx_Sig);
[signalOut1,pathGains1,sampleTimes1] = tdl(Distorted_Tx_Sig1);
Rx = Tx_distortion(signalOut,@(x) x + 0.036*x.^2 - 0.011*x.^3);

best_reservoir = best_reservoir.fit(Tx_Sig_OFDM_CP,Rx - Tx_Sig_OFDM_CP);
Rx1 = Tx_distortion(signalOut1,@(x) x + 0.036*x.^2 - 0.011*x.^3);
prediction1 = Rx1  - best_reservoir.predict(Tx_Sig_OFDM_CP1);

best_reservoir1 = best_reservoir1.fit(Tx_Sig_OFDM_CP,Rx - Tx_Sig_OFDM_CP);
prediction2 = Rx1  - best_reservoir1.predict(Tx_Sig_OFDM_CP1);

[RE_L1, IM_L1] = L1_Norm(Tx_Sig_OFDM_CP1, prediction1);
real_cancellation_depth_RC1 = 10*log10((1/RE_L1).^2);
imaginary_cancellation_depth_RC1 = 10*log10((1/IM_L1).^2);

[RE_L2, IM_L2] = L1_Norm(Tx_Sig_OFDM_CP1, prediction2);
real_cancellation_depth_RC2 = 10*log10((1/RE_L2).^2);
imaginary_cancellation_depth_RC2 = 10*log10((1/IM_L2).^2);

Prediction_visualization(real(Tx_Sig_OFDM_CP1),real(prediction1),real(
    prediction2),["Target signal" "MS=1 and MNN=1" "MS=35 and MNN=25"],'Time
    sequence n','Amplitude','Real part',5800,5850);
Prediction_visualization(imag(Tx_Sig_OFDM_CP1),imag(prediction1),imag(
    prediction2),["Target signal" "MS=1 and MNN=1" "MS=35 and MNN=25"],'Time
    sequence n','Amplitude','Imaginary part',5800,5850);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```