Chaos Computing

From Theory to Application

by

Behnam Kia

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved September 2011 by the
Graduate Supervisory Committee:

William Ditto, Chair
Liang Huang
Ying-Cheng Lai
Stephen Helms Tillery

ARIZONA STATE UNIVERSITY

December 2011

ABSTRACT

In this thesis I introduce a new direction to computing using nonlinear chaotic dynamics. The main idea is rich dynamics of a chaotic system enables us to (1) build better computers that have a flexible instruction set, and (2) carry out computation that conventional computers are not good at it. Here I start from the theory, explaining how one can build a computing logic block using a chaotic system, and then I introduce a new theoretical analysis for chaos computing. Specifically, I demonstrate how unstable periodic orbits and a model based on them explains and predicts how and how well a chaotic system can do computation. Furthermore, since unstable periodic orbits and their stability measures in terms of eigenvalues are extractable from experimental times series, I develop a time series technique for modeling and predicting chaos computing from a given time series of a chaotic system. After building a theoretical framework for chaos computing I proceed to architecture of these chaos-computing blocks to build a sophisticated computing system out of them. I describe how one can arrange and organize these chaos-based blocks to build a computer. I propose a brand new computer architecture using chaos computing, which shifts the limits of conventional computers by introducing flexible instruction set. Our new chaos based computer has a flexible instruction set, meaning that the user can load its desired instruction set to the computer to reconfigure the computer to be an implementation for the desired instruction set.

Apart from direct application of chaos theory in generic computation, the application of chaos theory to speech processing is explained and a novel application for chaos theory in speech coding and synthesizing is introduced. More specifically it is demonstrated how a chaotic system can model the natural turbulent flow of the air in the human speech production system and how chaotic orbits can be used to excite a vocal tract model.

Also as another approach to build computing system based on nonlinear system, the idea of Logical Stochastic Resonance is studied and adapted to an autoregulatory gene network in the bacteriophage λ.

This thesis is dedicated to Seville.

ACKNOWLEDGMENTS

Foremost, I would like to express my deepest gratitude to my advisor, Dr. William Ditto, for his continuous support, motivation and inspiration that made everything that I have achieved toward my PhD degree possible.

Besides my advisor, I would like to thank the rest of my thesis committee, Dr. Stephen Helms Tillery, Dr. Ying-Cheng Lai, and Dr. Liang Huang, for their encouragement and insightful comments.

I would like to express my deep gratitude to Kathleen Russell, former associate director of school of biological and health systems engineering, for her kind support and motivation. It was my pleasure to work with her and to know her.

My sincere thanks also go to Dr. Mark Spano and Dr. Anna Dari for the productive collaboration during my PhD and their friendship.

I would like to thank David Stanely, who as a good friend, was always willing to help. It would be a lonely lab without him.

Many thanks also go to my good friend, Younes Anari, for his friendship. He was the one that I could count on.

Last but not least, I would like to thank my parents and two sisters, Mehrnaz and Sarvenaz, for their unflagging love and support throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Computers and Computation

A computer is a physical device to carry out computation; in other words one can consider a computer as an implementation of a computation. Notice that in this chapter by saying *computer* I am not addressing modern, general-purpose digital computers; instead I define any physical device that can implement some computation as a computer, including modern digital computers as well.

To be an implementation of a computation, some sort of congruency between the behavior and nature of the physical computer and the type of computation is required.

The history of computers goes back to the stone age. The next section summarizes the history of computers and their evolution over the course of time. This study of the evolution of computers will clarify the needs and requirements for a new computing system and a new logic, which will be introduced in this thesis.

1.2. History of Computers

1.2.1. Devices for Counting

Man's fingers were probably the first computer used by humankind! This simple computer was used as a counting device through simple one-to-one correspondence [1].

The first reported computer hardware built by humankind seems to be the tally stick. Again, this computer was used as a device for counting (through one-to-one correspondence) as well as storing the numbers. The most ancient discovered tally stick is the *Ishango Bone,* which is on exhibition at the Royal Belgian Institute of Natural Sciences. It is believed to be more than 20,000 years old [2].

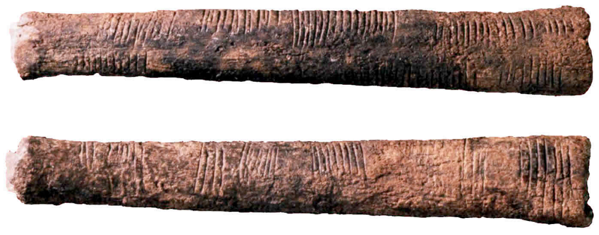
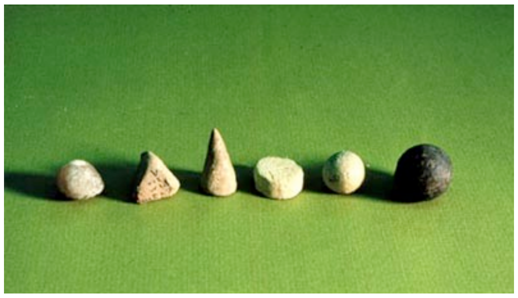


Fig. 1.1. *Ishango Bone,* the oldest discovered stick tally; it dates back to more than 20,000 years ago. Picture is from the Royal Belgian Institute of Natural Sciences [3].

Tokens are other ancient computing devices that date back to 8000-3000 BC. These clay artifacts have geometric forms such as cones, spheres, disks, cylinders, or others that take naturalistic shapes such as miniature animal heads, vessels, tools, and furniture [4].

Tokens were counters used to keep track of goods with each token form standing for one specific unit of a commodity. The number of units of merchandise was shown in one-to-one correspondence. It is known that, in the fourth millennium BC, the tokens were an accounting device used by the Mesopotamian temple administration to record entries or expenditures of goods offered by worshippers during monthly religious festivals [4]. Pictures of discovered tokens are illustrated in Fig. 1.2.

The ancient computing devices introduced in this section were mainly used for counting and representing the numbers and the main idea was very simple and primitive: one-to-one correspondence between each unit of goods and one finger/notch/token. Counting and representing the numbers were useful to keep track of livestock, grain, etc., but it wasn't enough. The ancients needed some computation and mathematics like addition, subtraction, or multiplication and a computing system that help them to do such computation. In the following chapter the next generation of computing devices is introduced, where the device stores numbers and helps the user to do primitive operation on the stored numbers.



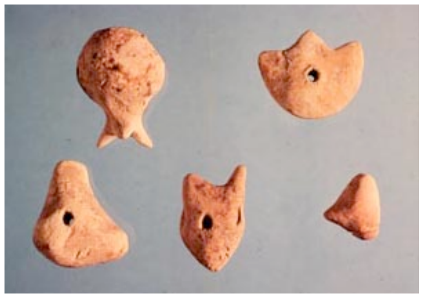


Fig. 1.2. Left: Plain tokens, Mesopotamia, Iraq, 4000 B.C. The cone, spheres and disk represented various grain measures; the tetrahedron stood for a unit of labor. [Credit: Denise Schmandt-Besserat, The University of Texas at Austin [4]]. Right: Naturalistic tokens representing animals, a vessel and a fruit, Iran, 3300 B.C. Credit: Musée du Louvre, Département des Antiquités Orientales, Paris.

1.2.2. Devices for Pseudo-computing

The next generation of computing devices wasn't just for representing the numbers; instead they helped the user to do computation on the stored numbers as well. At this stage, the computation wasn't fully automated inside the computing device. The device helps the user keep track of numbers as they do the computation, hence the name *pseudo-computing device*. The abacus and counting rods are examples of this generation of computing devices.

The abacus is a calculating tool used for performing arithmetic. Sumerians built the first abacus about 2700 BC [5]. This simple, but efficient computing system became popular in all ancient civilizations, including Rome, Persia, Greece, China, etc., and it has lasted even until now when it is still used as a simple calculator.

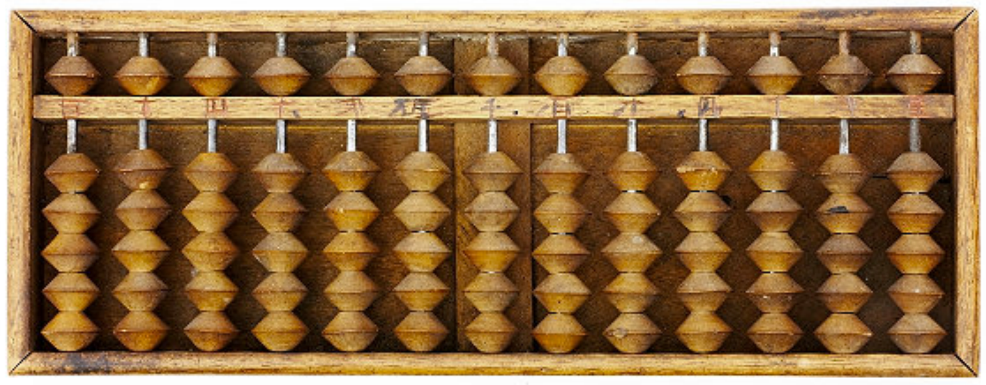


Fig. 1.3. A Korean-style abacus, dating back to early 1900 AD, Credit: Gwen and Gordon Bell [6]

Counting rods were used in ancient China, Japan, and Korea to perform mathematical and algebraic calculations. The most ancient specimens of counting rods were discovered from Chinese tombs dating from the second to first centuries BC [7]. Counting rods are placed either horizontally or vertically to represent any number and any fraction. The method for using counting rods for mathematical calculation was called rod calculation and by using it the ancient mathematicians were able to solve problems as hard as systems of linear equations or roots of numbers [7].

Such computing devices were a great help for computation. However, the user himself needs to know the method for computation and these devices were used as a tool to perform the computation. The turning point in the history of computing devices was introduced by mechanical computers designed for astronomical computation, where the computer physically models the dynamics of the system and as a result the computing system automatically solves the problem.

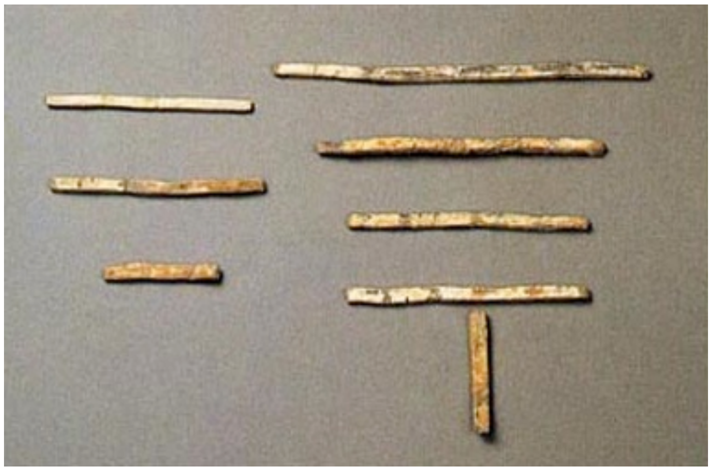


Fig. 1.4. The metal counting rod of the western Han Dynasty [8].

1.2.3 Mechanical Computers for Astronomical Computation

The introduction of mechanical computers for calculating astronomical positions was an important historical turning point for computing systems, because the computer was able to automatically carry out the requested computation with no help from the user. A non-expert user could easily enter the input data, e.g., the calendar date and local time, and the computer automatically gives the astronomical positions of the sun, moon, or other planets or vice versa.

Ancient scientists were the first to learn that a device that models a specific computation can be used as a computer for carrying out that type of computation. This is indeed the main and basic idea behind any other computer system that has since been built. Mechanical computers for astronomical computation are the first series of computing devices built based on this idea. Such computers were composed a series of mechanical gears whose movement models the celestial and planetary system. As a result such a device can automatically calculate the astronomical positions of planets or stars given the local time and date.

The most ancient specimen of a mechanical analog computer used for astronomical computation was discovered from a shipwreck named *Antikythera* in 1901 and it was called the Antikythera mechanism [9]. Studies revealed that this computing machine dates back to 100-150 BC and that it calculated and displayed celestial information, particularly cycles such as the phases of the moon and a lunisolar calendar [9].

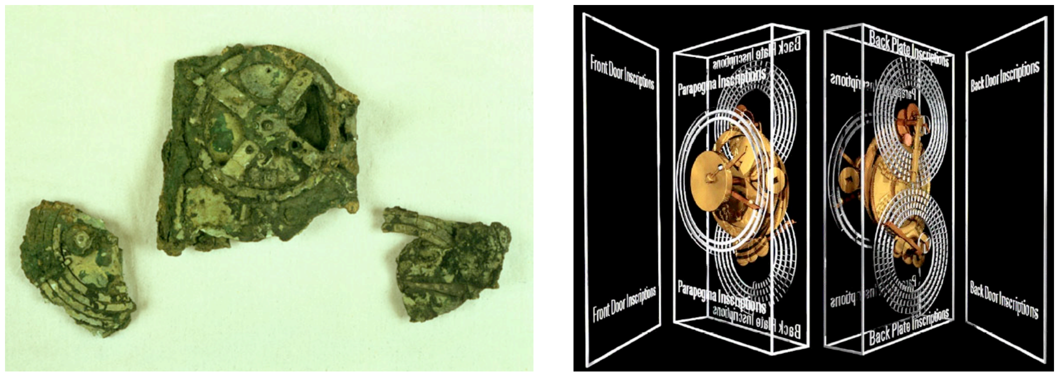


Fig. 1.5. Left: The Antikythera mechanism discovered from a shipwreck. Credit: National Archaeological Museum, Athens, Greece [10]. Right: A reconstructed schematic view of the Mechanism to illustrate the position of major inscriptions and dials [9].

The Astrolabe [11], the Planisphere [12], and the Equatorium [13] are other mechanical and analog computers used for astronomical calculation that were invented and used during ancient days and middle ages.

These computers were built for calculating celestial information and lunisolar calendars because (1) for ancient societies timing agricultural activity and fixing religious festivals had great importance, (2) eclipses and planetary motions were often interpreted as omens, while the calm regularity of the astronomical cycles must have been philosophically attractive in an uncertain and violent world [9].

There have been a few other important and interesting achievements in computing systems during Middle Ages and renaissance, famous examples are Pascal's calculator Da Vinci’s clock, however we don’t see a new generation of computers until industrial revolution. Industrial revolution introduced new problems and computations to humankind which required advanced types of computers. Mechanical calculators and early models of mechanical programmable computers are the devices designed and built based on these new needs.

1.2.4. Mechanical Calculators and Mechanical Programmable Computers

The industrial revolution put a new and limitless set of problems to solve in front of humankind, which in turn required more sophisticated forms of computer. This new quest for sophisticated computers started from the 18th century, when different scientists started to build new calculators and programmable computers. The common characteristics of these new series of computers were that (1) they were mainly mechanical and (2) the designers were trying to make these computing systems general purpose and programmable.

Babbage's difference engine and analytical engine were probably the most famous computer systems of this era. The difference engine was designed to compute values of polynomial functions. Also Babbage recruited the idea of punched card programming introduced by Joseph-Marie Jacquard in 1801 and tried to build a programmable, general-purpose mechanical computer, named the analytical engine. However his efforts to build it failed because of lack of enough funding. Recent studies have proved that his designs and architectures for the difference engine were correct and working versions of his design have been built successfully [14].

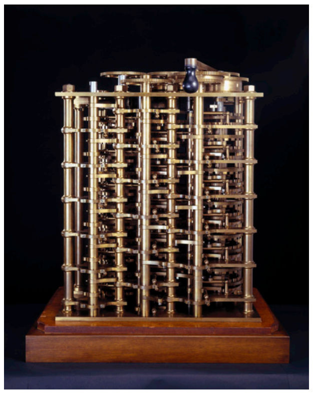
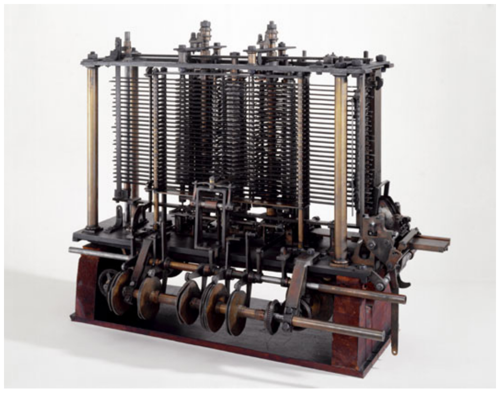



Fig. 1.6 Left: Difference engine built by Babbage. Right: A trial version of the analytical engine built by Babbage. Both pictures are from science museum, London, UK [15].

At this era, a few technologically and commercially successful calculator machines were built and introduced too. The arithmometer patented by Thomas de Colmar and manufactured from 1851 to 1915 is a good example [16]. This computing device was a mechanical calculator that could add and subtract directly and could perform long multiplications and divisions effectively by using a movable accumulator for the result.

The quest for building a programmable general-purpose computer mainly failed because of the lack of the necessary technology or funding until the mid 20th century. Meanwhile analog computers continued progressing and they extended their scope from solving algebraic problems and equations to differential equations.

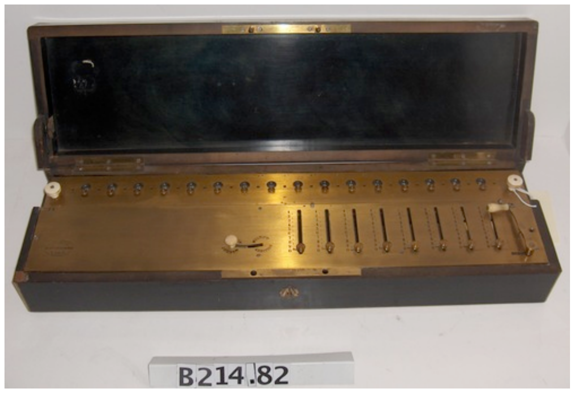


Fig. 1.7. Thomas de Colmar's Arithmometer, 1890 AD. Credit: science museum, London, UK [15]

1.2.4. Modern Analog Computers for Differential Equations

In late 19th and early 20th centuries, advances in engineering and technology introduced new problems to solve. Based on Newtonian mechanics, differential equations govern the motion of an object, the trajectory of a bullet, the growth rate of the economy, etc. This need for computers capable of solving differential equations initiated dozen of different projects and resulted in the invention of new analog computers for solving differential equations during the early 20th century.

The trajectory of a missile and the dynamics of a simple spring-dashpot mechanical system can be governed by the same differential equation. This similarity between the dynamics was the main idea behind analog computers: build a simple mechanical or electrical system that is governed by the differential equation of interest. The time evolution of the system is the solution of the differential equation for the other.

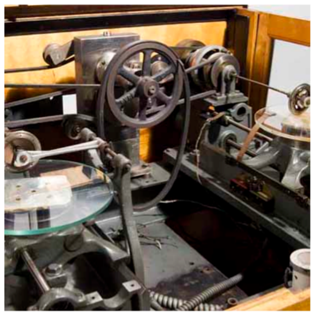
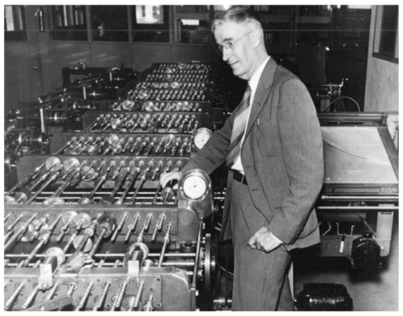



Fig. 1.8. Left: An integrator block of differential analyzer, MIT. Right: Vannevar Bush standing near to his analog computer. Pictures are from MIT's online museum [17].

The differential analyzer of MIT is a famous example of this generation of computers built in MIT by Vannevar Bush and his students during 1928–1931 [18]. This analog computer was composed of 6 mechanical integrators that had been introduced by William Thomson (Lord Kelvin) as well as thousands of other gears and rods [19].

These analog computers were a huge step forward. However they suffer from critical weaknesses. The most important problem was that their computation was restricted to solving differential equations. These analog computers were able to solve any problem or equation that was *buildable* on the mechanical hardware. If the problem was outside the dynamical behavior of the hardware, the computer fails to solve the problem. The other problem was the programmability of these computers. Not only were these computers programmable to just a narrow set of problems, but also this programming required mechanical reconfiguration of the connections and setup. Claude Shannon, the famous American engineer and mathematician, operated and reconfigured the differential analyzer of MIT early in his career. During his work with this machine, he discovered that switching circuits, used in the controller part of the differential analyzer, can be modeled and simplified using a then relatively unknown branch of mathematics named Boolean algebra. This discovery opened the doors to the invention of digital programmable computers.

1.2.5 Digital Computers

1.2.5.1 Digital Logic

George Boole, a British mathematician, was interested in formulating a calculus of reasoning, and in 1847 he published a pamphlet titled “Mathematical Analysis of Logic” [20]. In this article he developed and introduced novel ideas on logical reasoning and argued that logic should be considered as a separate branch of mathematics, rather than being considered a part of philosophy. Boole argued that there are mathematical laws to express the operation of reasoning in the human mind [20]. Boole’s work on what is now called Boolean algebra remained relatively unknown for many years as it seemed to have little practical use to society. However, Claude Shannon in his master's thesis, *A Symbolic Analysis of Relay and Switching Circuits*, proved that Boolean algebra could be used to simplify the arrangement of the electromechanical relays then used in telephone routing switches and, much more importantly, he proved that it should be possible to use arrangements of relays to solve Boolean algebra problems [21]. This concept, of utilizing the properties of electrical switches to do logic, is the basic concept that underlies all digital computers.

Boolean algebra (Boolean logic) was a new, abstract way to express reasoning and computation, and, via the introduction of switching circuits as an implementation of Boolean logic, a new generation of computers was developed.

However, this new computer still wasn’t programmable. It was Von Neumann and his architecture for a stored program computer that described how to build a general-purpose programmable computer.

1.2.5.2 Von Neumann Architecture

The earliest computing machines had fixed programs that were designed (hardwired) to do a specific task. For example, the Antikythera mechanism was only able to calculate planetary positions. The limitation of a fixed (hardwired) program computer is that it is designed and programmed to do a specific task. If it is required to change the program, then it is usually necessary to re-wire and re-design the machine, which was a complex manual process and which involved engineering designs and physical changes (if re-wiring and re-designing was possible at all).

Von Neumann architecture explains how one can build a general purpose, programmable computer. It says that a general-purpose computer must store data and a program (or sequence of instructions from and instruction set) in a storage structure and then call and execute these instructions one by one in a CPU [20].

The main idea behind Von Neumann architecture is to define a set of basic instructions that can encode any problem (an instruction set) and then implement these fundamental instructions within the hardware. Then for any problem one can write a program, which consists of instructions from this instruction set, and then feed this program along with the corresponding data inputs to the computer. The computer reads this program and executes it line by line. Execution of an instruction is nothing more than sending the appropriate data to the hardware implementing each specific instruction. Almost all modern computers have been designed and built based on this idea.

In the next chapter, conventional computers and their drawbacks are described.

1.3 Need for a New, Dynamic-based Programmable Computer

The main problems of modern digital computers are that (1) although the technology used for implementation can be extremely fast, the computation can remain relatively slow. The reason is that, in modern digital computers, instead of direct calculation of the problem a programmed version of the problem in terms of that computer's instruction set of the computer is executed. Such programs are executed instruction by instruction, with each instruction initiated by a master clock. Each instruction takes one or more cycles (unless parallelism is available and feasible). (2) The second problem is that, although the computer is "programmable", ironically the hardware is not programmable. As said before, for each instruction in the fundamental set of instructions, there is one physically separate hardware implementation, and, at the arrival of each instruction to be run, the corresponding hardware implementation of that instruction is used while the remainder of the hardware is idle. As a result, inside the hardware for a programmable digital computer, there are dozen of implementations for instructions, but at each cycle, just one of them is used. These unused implementations of instructions are just wasting power and chip area.

In this thesis a new programmable computer will be introduced, which reconfigures itself to be the exact implementation for each instruction encountered in the program. This new computer system will utilize all of its hardware, with minimal redundant, unused hardware. Such a computer extends the meaning of programmability to new levels, where the intrinsic dynamics of the system is programmable. The enabling technology for such a brand new computer is based on a novel idea for building logic circuits called chaos computing [22,23,24,25,26,27,37,28,29,57,58,59,60,61,70].

Chaos computing is a new approach to implementing a logic circuit. Similar to ancient mechanical computers, or 19th century analog computers, the intrinsic dynamics of a chaos computing engine models a function. Furthermore, since a chaotic system exhibits different behaviors and patterns, it can implement different types of functions [37,24,25,26,27,28,29].

In this thesis, after introducing and studying the chaos-computing model, a new computer architecture designed for these logic blocks (which replace the hardwired instruction set of digital computers) will be introduced. This architecture will use these chaos-based logic blocks to obtain a truly reconfigurable computer, in which the hardware of the computer is truly reprogrammable. Such a computer will utilize all of its resources, so there will be no waste of power or IC area. Furthermore, since the hardware itself is deeply reconfigurable, the programmability of the computer will not be restricted to just an instruction set, and the user will be able to program the computer to be an almost exact implementation of his application.

1.4. Need for a New Logic

The operations of modern digital computers are restricted to a basic set of instructions. The problem is that in some applications it is really hard, if not impossible, to describe the problem and solution in terms of the basic instructions. Famous examples are pattern recognition, artificial intelligence, perception, etc.

In this thesis, even beyond introducing a new chaos-based implementation for the logic and computation, a new form of computation employing chaotic dynamics will be introduced for applications like speech modeling and artificial intelligence.

The main idea is that I will show that a chaotic system can and should be used for modeling speech. Since a chaotic system can better physically model the process of speech production (speech production involves *turbulence*, a form of chaos), it can provide a superior implementation for modeling speech.

1.5. Organization of the Thesis

The main concepts of chaos theory are introduced and explained in chapter 2. The focus of this chapter is on the main ideas and concepts of chaos theory that will be used in next chapters of this thesis. In this chapter the importance of chaos and nonlinearity will be discussed in detail and it will be explained why chaotic dynamics can be very beneficial in computation.

In chapter 3 a short review of chaos computing is presented. I will show how one can build different digital functions using a chaotic system.

In chapter 4 a new theoretical analysis for chaos computing is presented. Until now no direct technique has been introduced to determine the possible functions that a given chaotic system can implement or the control inputs that select these instructions. Rather the evolution of any chaotic computing model under different inputs is observed and monitored to determine its instruction set. In this chapter the computational capabilities and properties of a chaos-based computer are connected to the dynamical properties of the underlying chaotic system. Specifically I demonstrate that the instructions that a chaotic system can implement can be directly determined from the periodic orbit structure and the dynamics of the system. Furthermore, in Chapter 5 I use the *unstable periodic orbit* (UPO) theory and the UPO model to estimate the robustness of a chaotic system in doing computation.

Examples for application of UPO in chaos computing and determining and estimating the functionality and robustness of chaotic system in computation are presented in Chapter 6.

Furthermore, since UPOs are experimentally extractable from a time series, one can determine the computational functionality of a chaotic system and its robustness based on a time series with no need to know exact dynamical equations of the system. In chapter 7 I explain how from a time series one can reconstruct a UPO based model for chaos computing and how this model can be used in determining and estimating the functionality and robustness of the underlying chaotic system in computation. An example of this extraction of functionality and robustness from a time series is presented in chapter 8. UPOs and their eigenvalues are extracted to reconstruct the UPO model and to estimate the functionality and robustness of underlying chaotic system in doing computation.

In chapter 9 a brand new architecture for chaos computing is proposed. This architecture explains how one can arrange chaos-based logic blocks beside each other to obtain a computing system. The computing system I build will have a flexible instruction set, meaning that the user has the option to choose the desired suitable instruction set for his application.

In chapter 10 I describe how chaotic systems can be used in coding and synthesizing human speech. In this chapter the focus is on a famous speech coding and modeling technique, named CELP, and I demonstrate how random number generators and random series can be replaced with chaotic systems and chaotic time series to obtain better performance. Chaotic excitation of the filter which models the vocal tract is more biological oriented and it results in better performance.

In chapter 11 a new nonlinear dynamical systems-based approach for computation, named logical stochastic resonance, is introduced and it is adapted on a regulatory gene network, named lambda phage λ. This genetic regulatory based logic block is able to build AND and OR gates.

At the end Chapter 12 presents the conclusions I can draw from this work and it explains the future directions following the thesis.

CHAPTER 2

CHAOS THEORY, THE MAIN CONCEPTS

Even large dedicated books for chaos theory fail to cover all aspects of this new and broad branch of modern science, so there is no way I can introduce and explain all aspects of chaos theory in this short chapter. Instead I will introduce the main concepts and ideas of chaos and nonlinear dynamical systems theory that I will use in my work in next chapters. Furthermore, I keep the mathematics and definitions as simple as possible in favor of meaning, implication and application of each concept. The main reason is the aim of this thesis is to apply chaos theory in information processing. The main idea is to bridge the concepts between chaos theory and information processing systems, and find counterparts for each concept of information processing in chaos theory. The counterparts in chaos theory will be a *realization* of the information-processing concepts. For such pairing of concepts, having a deep conceptual understanding of chaos theory is more important than the mathematical details and equations. Therefore the main focus of this chapter, which covers the background of chaos theory, will be on the main ideas and concepts. I start from the definition of a nonlinear system, and will explain why and how it differs from linear systems, and what the implications of nonlinearity are.

Readers who are seeking more mathematical details of the chaos theory and dynamical systems theory are encouraged to refer these text books [30,31].

2.1. Dynamical System

A *Dynamical system* is a system described by a set of equations that gives the time evolution of the state of the system from an initial state. These equations present the continuous-time evolution of the system:

$$\dot{\mathbf{x}} = \mathbf{F}(\mathbf{x}) \tag{1}$$

where $\mathbf{x}$ is a state vector of the system, or the equations can be discrete, presenting the discrete-time evolution of the system:

$$\mathbf{x_{n+1}} = \mathbf{F}(\mathbf{x_n}) \tag{2}$$

where $\mathbf{x_k}$ represents the state of the system at time step *k*. The sequence of states trace an orbit in state vector space of the system.

Maxwell's equations for plasma, the Navier-Stokes equations for a fluid, and Newton's equations for a motion of a particle are examples of dynamical systems.

A dynamical equation can be linear or nonlinear. In either case, the chosen variables which comprise the system's state vector must *span* the state space. That is, they must *completely* describe the current state of the system. A corollary is that they consequently also fully describe all future states of the system.

2.1.1. Linear Dynamical System

A linear dynamical system is a system whose dynamical equations are linear; that is, the dynamical variables describing the properties of the system, e.g. position, velocity, acceleration, current, voltage, etc., appear in the equation in a linear form.

A linear system can be broken down into parts, each part can be solved separately, and finally these solutions are recombined to get the answer to the linear system. This reduction allows a huge simplification of the complex systems. Laplace transforms, Fourier transforms, and the superposition argument are examples of the application of such simplifications in a linear system, where the system is described and solved in terms of a basis set of simple solutions.

The good thing about a linear dynamical system is that one can solve it and obtain a closed-form equation for its time evolution. Having such a closed-form solution for the linear dynamical system implies that knowing the initial condition of the system results in full knowledge about its entire future evolution. One basically needs to put the initial condition in the closed form solution of the system and it will give him the exact state of the system at any time of future.

Notice in nature there is no perfect linear system. Any system that appears to be linear in some range of parameters will eventually behave nonlinearly if we change or increase the parameter values. A linear dynamical system usually models the behavior of a real life system in some specific range of parameters. However, because of its ease of use and closed form solutions, scientists prefer to approximate and model real life systems with a linear model at some range of parameters.

2.1.2. Nonlinear Dynamical Systems

A nonlinear dynamical system is a system whose dynamical equations are nonlinear; that is, some of the dynamical variables describing the properties of the system, e.g. position, velocity, acceleration, current, voltage, etc., appear in the equation in a nonlinear form.

Most nonlinear systems are impossible to solve analytically. In a nonlinear system, the solution of the system is not simply the combination of the solution of the subsystems. There is an extra solution which arises from nonlinear interactions too. Also superposition doesn't hold true for a nonlinear system. A solution of a nonlinear system to a combination of inputs is not the combination of the solutions of the system to the individual inputs. The bottom line is there is no generic, global method for solving a nonlinear dynamical system and obtaining a closed form solution.

This nonlinearity has different implications: (1) the system solution and output can be different and more than the sum of the solutions of its subsystems. Some brand new and novel features and behaviors may *emerge* from a nonlinear system without having any root in subsystems of the system. (2) At some range of parameters the solution of the nonlinear system can be very sensitive to initial conditions, which makes the long-term future prediction of the system impossible, even though the dynamical equation and initial condition of the system is known and in hand. These two important concepts are discussed in next two sections.

2.2. Emergence from Nonlinear System

A nonlinear system is more than a combination of subsystems. It may show some behaviors or features that cannot be tracked down to a specific subsystem. Such newborn features and behaviors arising from nonlinearity and complexity are called *emergent* behaviors.

Most of interesting features, orders, behaviors, and even concepts in nature seem to be emergences from the nonlinearity of nature [32,33,34]. Famous examples of such emergent orders or concepts are the temperature of a gas, intelligence, or even the life itself [32,33,34]. Many of the concepts that have troubled the scientist in understanding nature are usually emergences from complexity and nonlinearity. The old approach of science, reductionism, which says reduce the system to subsystems and study the individual subsystems to understand the overall system fails in understanding an emergence. The reason is obvious. An emergent behavior arises from nonlinear and complex interactions, and it usually has no clear track in the individual behavior of subsystems. But when these subsystems are put together and nonlinearity comes into the equation, the emergent behavior emerges from nonlinear interactions that are hard to understand and solve.

A famous emergent behavior from a nonlinear system is the phenomenon of chaos. Chaos is a random-like behavior from a fully deterministic system. I will return back to this concept below.

2.3. Sensitivity to Initial Conditions

At some parameter values of a nonlinear system, a small change of initial condition results in a dramatic change of orbit path. Such divergence of nearby orbits in a nonlinear system is a signature of a phenomenon called *chaos*. In a chaotic system the behavior of the system is aperiodic and it's "apparently" random. The keyword *apparently* is here to emphasize the fact that the system is not random; it is fully deterministic. But the behavior of the system is random-seeming and unpredictable in the long term.

As an example, in Fig. 2.1 a very simple and 1-D map like the Logistic map, $x_{n+1}=4x_n(1-x)$, is iterated from two different, but nearby initial conditions. The orbits starting from these slightly different initial conditions behave completely differently after a few initial iterations.

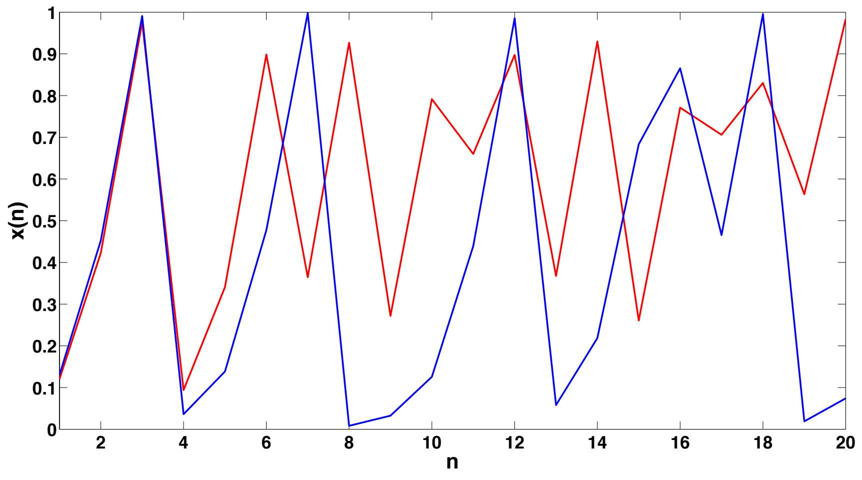


Fig. 2.1. Two orbits starting from nearby initial conditions diverge from each other and behave very differently. This sensitivity to initial conditions is a signature of a chaotic system.

2.4. Determinism and Unpredictability, Are They Collectable in One System?

A chaotic system is a deterministic system, meaning that knowing the dynamical equation and initial state of the system the future evolution of the system should be known [35]. But a chaotic system looks to be unpredictable. The question is how can these two paradoxical concepts be gathered together?

Sensitivity to initial conditions is the answer. If someone knows the *exact* initial condition, and can integrate the dynamical equations precisely without any error, he will be able to know the exact time evolution of the system and the orbits will be predictable. But in real life, we can not *in principal* know the exact value of the initial condition of a system or, for that matter, its exact equations, and even if we do, the next problems will be (1) round-off errors, arising from the fact that we can not store a real number in a memory element with finite precision, (2) truncation and discretization error in numerical integration of the nonlinear system. On the other hand in real life and as an experiment, by setting the initial state of a chaotic system to some initial value and running the system for some time, we are not going to see the same time evolution. The imprecisions in setting the initial conditions and the background noise are enough to diverge the orbits and as a result again the system behavior will look unpredictable and random-like.

To sum up, I can say that the future of a chaotic system is indeterminate even though it is a deterministic system.

2.5. Nonlinearity, Good or Evil? Motivation of the Thesis

The lack of analytical methods and generic closed form solutions for nonlinear systems has made them hard to cope with. For a science or engineering undergraduate student, a nonlinear differential equation can be the hardest, most unpleasant and unappealing problem to solve. The majority of the science and engineering community have been engaged in studying nonlinear systems and problems and millions and even billions of dollars each year is dedicated for funding such research. Based on these facts one may conclude that we would have a better life if the world were linear. But the truth is that nonlinearity plays a vital role in nature and furthermore without nonlinearity there would be no life! The implications, importance, and applications of nonlinearity and chaos are discussed in the following.

2.5.1. Chaos and Information

A chaotic system provides us with both determinism and unpredictability at the same time. Based on Shannon's information theory point of view, unpredictability is information [36]. So I can say that unpredictable orbits of a chaotic system convey and represent information. To clarify the concept let's compare a chaotic system with a periodic system. In a periodic system periodic evolution of the system represents no information since it's just repeating itself and the observer learns nothing new by watching different repetitions of a cycle. But a chaotic evolution is aperiodic and unpredictable, so there is information (unpredictability) embedded in it. The question that arises here is where this information is coming from? In other words, a chaotic orbit is aperiodic and will not repeat itself, so it represents an infinite amount of information. Where is the source of this infinite amount of information? The answer is connected to the sensitivity to initial conditions in the chaotic system. We need infinite precision in setting the initial condition of a system so that we precisely determine the infinite evolution of the system.

The determinism in a chaotic system allows us to encode information in a chaotic system though initial condition setting. But the need for infinite precision in setting the initial conditions seems to be problematic. However the problem can be easily solved by restricting the evolution time of the orbits. A finite precision in setting the initial conditions suffices to encode the information required to represent the unpredictability in this limited evolution of the system. I used this idea to build chaos-based computing system [22,23,37,61,70] as I will review it in chapter 3.

2.5.2 Chaotic System as a Rich Library of Different Patterns and Behaviors

Unlike linear systems, in a nonlinear system with a change of parameters the system may undergo qualitative as well as quantitative changes in its behavior. As an example, consider the bifurcation diagram of the logistic map, $x_{n+1} = \lambda x_n(1 - x_n)$, where λ is the bifurcation parameter as depicted in Fig. 2.2. A bifurcation diagram represents the steady state solutions of a chaotic system versus a given parameter, called the *bifurcation parameter*. Bifurcation theory itself is the mathematical study of the qualitative change of a dynamical system when parameters change [30]. As is illustrated in Fig. 2.2, by change of one parameter the system exhibits completely different behaviors. At some parameter values the system is periodic, and at some others it's chaotic (periodic with periodicity of infinity). Furthermore, at parameter values where there is chaos, the system is composed of an infinite number of unstable periodic orbits [64,35].

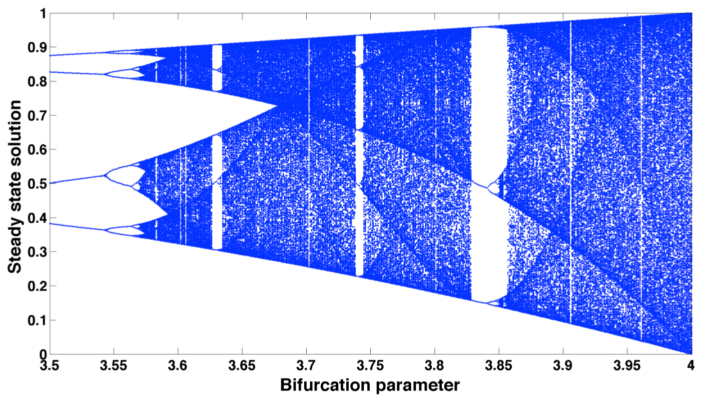


Fig. 2.2. Bifurcation diagram of the Logistic map is depicted. At different values of the parameter, the system behavior qualitatively as well as quantitatively changes.

A linear system lacks such a broad range of patterns and behaviors. Changing a parameter of a linear system results in change of amplitude and/or frequency of the system solution. It is nonlinearity and chaos that provide us with a library of different patterns and behaviors, and furthermore, since the dynamics is deterministic, these patterns or behaviors are distinct and can be selectively stabilized, as will be discussed in section 2.7. Thus the main idea of the following chapter is:

Utilizing a chaotic system as a library of different patterns and behaviors, in which we can select each pattern or behavior based on our needs.

2.5.3. Chaos and Emergence

It seems that most of the interesting things we see in this world are nothing more than emergences from complex and nonlinear systems. Examples are like intelligence, temperature, or life itself. Without nonlinearity there will be none of these entities.

2.6. Three Main Characteristics of Chaos

There is no main mathematical definition for chaos. Instead there are three main characteristic behaviors that are associated with a chaotic system [38]. They are (1) sensitivity to initial condition, (2) density of unstable periodic orbits in a chaotic attractor, (3) Topological transitivity.

2.6.1 Sensitivity to Initial Condition

Sensitivity to initial conditions was discussed as the main signature of chaos earlier in section 2.3. In a chaotic system nearby orbits diverge exponentially. The Lyapunov exponent is a famous quantitative test to detect whether a system is chaotic in terms of average divergence of nearby orbits, and, if the system is found to be chaotic, how strongly chaotic it is, again in terms of average divergence rate. Consider simple 1-D map. Let x_0 be the initial condition, and $x_0 + \delta_0$ be another nearby initial condition, where δ_0 is extremely small. Let δ_n be the separation after *n* iterations (assuming that the dynamical system is discrete). If $|\delta_n| \approx |\delta_0|e^{n\lambda}$, then λ is called the Lyapunov exponent. A positive λ indicates chaos, the exponential divergence of initial conditions. Notice some references call this definition a *local* Lyapunov exponent.

By knowing the exact dynamical equations of the system, one can easily compute the Lyapunov exponent. Assume the system is discrete, and $x_{n+1}=f(x_n)$. After taking logarithms from the definition of Lyapunov exponent and noting that $\delta_n = f^n(x_0 + \delta_0) - f^n(x_0)$, I obtain

$$\lambda \approx \frac{1}{n} ln \left|\frac{\delta_n}{\delta_0}\right| = \frac{1}{n} ln \left|\frac{f^n(x_0+\delta_0)-f^n(x_0)}{\delta_0}\right| = \frac{1}{n} ln|(f^n)'(x_0)| \tag{3}$$

and based on our assumption, $\delta_0 \to 0$. By expanding the *n*th order derivative based on the chain rule I have

$$\lambda \approx \frac{1}{n} ln|\prod_{i=0}^{n-1} f'\,(x_i)| = \frac{1}{n} ln|\prod_{i=0}^{n-1} f'\,(x_i)| = \frac{1}{n}\sum_{i=0}^{n-1} ln\,|f'(x)_i| \tag{4}$$

Similar calculations can be carried out to compute the Lyapunov exponent for a continuous system too [39]. Also the Lyapunov exponent can be computed from experimental time series as well [40].

2.6.2. Density of Unstable Periodic Orbits

A chaotic attractor is composed of an infinite number *unstable periodic orbits* (UPOs) and these UPOs approach every point in the attractor arbitrarily closely [64,30,31]. UPOs play a very critical role in a chaotic system because they are the skeleton of the chaotic attractor [64]. Furthermore, periodic orbit theory [64,65] says that a collection of short UPOs is enough to model a chaotic attractor and to estimate invariant measures of the system [64,65]. In chapters 4 and 5 I will use a similar UPO-based model to describe a chaotic system and to explain and estimate the computational functionality of the system.

The other interesting property of UPOs, which makes them even more fascinating, is that a UPO is experimentally extractable from the time series [64]. In chapter 7 I will show how this extractability of UPOs from a time series enables us to model the underlying chaotic system and estimate its functionality and robustness in doing computation solely from the time series.

2.6.3 Topological Transitivity

Topological transitivity (topological mixing) in a chaotic system means that under chaotic evolution of the dynamical system a mapping of points residing in one region of the phase space will, after a sufficient number of mappings, visit any other given region of the space. This concept is closely related to, but not identical with, *ergodicity*.

2.7. Manipulating Chaos

Unpredictability and random-like behavior in chaotic evolution of a system can arise from a deterministic system. As a result determinism enables us to reliability manipulate the observed chaotic evolution. Here I list a few of these important techniques, where the underlying determinism is utilized to select, control, or synchronize the chaotic evolution.

2.7.1. Initial Condition Selection

Chaotic systems are sensitive to their initial conditions, and because of our inability to precisely set the initial condition of the chaotic system, I will not be able to accurately select a certain desired orbit through setting the initial conditions. A minor error and deviation will exponentially magnify via the chaotic evolution of the system and at some point, the system behavior will be completely different from the desired orbit. However, by selecting initial condition with finite precision and a small error, the short-term evolution of the system will be the same as the desired one. This technique simply allows us to program short term evolution of a chaotic system by simply setting the initial condition of the system to an appropriate value.

In this technique, the evolution time is restricted, so with finite precision in initial condition setting I can program the system orbit. However the system is still chaotic, so we have a broad range of behaviors to select in this way. I have used this simple technique for building chaos-based computing systems [37,61,70].

2.7.2. Chaos Control

I introduced a chaotic attractor as a library of different patterns. Since the system is deterministic, I can modify or control the dynamics to stabilize the desired patterns. As an example, as discussed before, a chaotic system is composed of an infinite number of UPOs. We can explain the chaotic dynamics and evolution using these UPOs. Starting from an initial condition, the chaotic orbit stays in a neighborhood of the nearby UPO, but after a short time because of the fact that UPOs are dense, the orbit will be nearer to another UPO, so it will diverge from the first UPO and will follow the new UPO. However, again, after a short time there will another newer and nearer UPO and the orbit will follow it, and this processes continues. The overall motion is like the chaotic orbit is wandering between these dense and infinite numbered UPOs. The main idea in chaos control is to stabilize a UPO so that the orbit remains near to it.

In 1990 two separate groups of researchers, a theory group at the University of Maryland in College Park, and an experimental research group the Naval Surface Warfare Center, theoretically and experimentally showed how the chaotic, unpredictable motion of a system can be tamed and stabilized around one of the pre-existing UPOs using time dependent, tiny perturbations to a system parameter [41, 42]. The discrete-time technique is called OGY, after the name of three authors, Ott, Grebogi and Yorke, of the theoretical paper and it was followed by numerous extensions and treatments.

Two years later Pyragas introduced a new time-delayed feedback controller for continuous chaos control [43]. Similar to OGY, this method for chaos control stabilizes one of many existing UPOs of the chaotic attractor, but here it works in a continuous fashion. Again, this paper initiated a series of extensions, all named time-delayed feedback control of chaos.

Another class of chaos control techniques is open loop control methods [44]. As the name suggests, there is no feedback from the current state of the chaotic system; instead the controller excites the chaotic system with some stimulation function which is usually periodic or quasiperiodic [44]. In such control techniques the chaotic system is not stabilized to a desired pattern; instead the chaos is suppressed and system behavior is stabilized to some pattern or orbit that did not necessarily arise from the existing UPO structure of the system [44].

These aforementioned approaches to chaos control are the three historically earliest and most actively developing directions of research. However there are other directions to chaos control as well. Another approach to chaos control is to use conventional classic control theory techniques in controlling chaos. For example, in [45] a linear control technique is introduced to control chaos, or in [46,47,48] more elaborate nonlinear techniques are used for chaos control. Note that it has been shown that the OGY method also fits into this category since it is equivalent to using pole-placement theory to control the system [49].

A threshold controller is another simple, but effective technique for controlling chaos [50]. In this technique whenever the state of the system exceeds some threshold the state is reset to the threshold value. Using this technique one can stabilize the chaotic system to different periodic orbits. However these periodic orbits don’t belong to the main attractor of the uncontrolled system; instead they are created because of the coupling of system and controller [50].

Also for chaos control modern approaches of control have been introduced too. The examples are neural networks [51,52] or fuzzy modeling and control [53] are applied too.

2.7.3. Chaos Synchronization

In 1990 Pecora and Carroll showed that two chaotic systems could be synchronized [54]. It was a very interesting finding because in two chaotic systems, in which (1) parameters will differ from each other no matter how precise the process of manufacturing and (2) no matter how exact the initial conditions of the two systems are set to the same value, there will be still some error, orbits of the systems naturally tend to diverge from each other because of chaos. Pecora and Carroll demonstrated that by designing a common link between two chaotic systems so that the Lyapunov exponents of the subsystems are negative, one can synchronize the two chaotic systems.

Furthermore, it is demonstrated that two different chaotic systems can be synchronized as well [55,56], and it is called generalized synchronization of chaos.

2.8. Summary and Discussion

Nonlinearity is introduced as the main source and cause of important features and phenomena in the nature, including life and intelligence. At some parameter values of a nonlinear system chaos can happen, which shows itself as extreme sensitivity of orbits to initial conditions. Such sensitivity to initial condition causes the orbits to seem unpredictable and random.

A chaotic attractor is composed of an infinite number of UPOs, where these UPOs are dense. UPOs can be utilized for modeling an attractor and play a critical role in chaos theory.

A chaotic system was introduced as a library of different patterns and behaviors, and since chaos arises from a deterministic system these patterns are selectable.

Three main methods were introduced for manipulating a chaotic system to select a desired pattern: (1) initial condition choice, where we set the initial condition of a system to program the system to behave based on our desire for a short-term evolution, (2) chaos control, where a controller is recruited to stabilize a pattern, (3) chaos synchronization, where the behaviors of two chaotic systems are synchronized, so that one precisely follows the other’s chaotic evolution.

Having such a rich library of patterns, named chaos, in one hand, and being able to program it on the other hand, makes a chaotic system a suitable candidate for implementing information processing tasks.

CHAPTER 3

CHAOS COMPUTING, THE MAIN IDEA

The main idea of chaos computing is to harness the library of orbits/patterns inherent in chaotic systems to select out logic operations and to utilize the sensitivity to initial conditions of such systems to perform rapid switching (morphing) between all of these logic functions [70]. These features are sufficient to perform reconfigurable logic operations using the chaotic system.

Data and control inputs to a chaotic system (either continuous or discrete) may be encoded as either the initial conditions of the chaotic system or the parameters of the system. Here I focus on the former technique. After applying the inputs, the system is allowed to evolve for a predefined time, after which time this "final state" of the chaotic system is decoded as the computation's output.

To be more precise, consider the *m* digital data inputs, $X^1_{Data}, X^2_{Data}, \ldots, X^m_{Data}$, to a computing engine and the *n* digital control inputs, $X^1_{Control}, X^2_{Control}, \ldots, X^n_{Control}$. Computation with this system consists of three steps:

Step 1: Each set of data and control inputs is mapped to a point on the unstable manifold of the chaotic system. This point will be used as the initial condition for the chaotic system. Let *T* map (encode) the *m* data and *n* control inputs onto the space of the initial conditions. If *L* is a binary set {0,1}, then $L^{(n+m)}$represents the domain of *T*, which consists of all the possible combinations of digital data and control inputs. Let β be the unstable manifold of the chaotic system, R^s the general state space of the chaotic system, and *Y* the output of the encoding map on the unstable manifold. In this case the general form of the encoding map, *T*, is as follows:

$$\begin{aligned} &T : L^{(n+m)} \to \beta, \quad \beta \subset R^{s}, \quad L = \{0,1\} \\ &Y = T(X^{1}_{Data}, X^{2}_{Data}, \ldots, X^{m}_{Data}, X^{1}_{Control}, X^{2}_{Control}, \ldots, X^{n}_{Control}) \end{aligned} \tag{5}$$

Step 2: Starting from the initial conditions produced by the encoding map, the chaotic system evolves for a fixed time (or for a fixed iteration number, if the chaotic system is discrete).

Step 3: After the evolution time, the system stops working and its state at the end of the evolution time is sampled and decoded to the outputs using a decoding map.

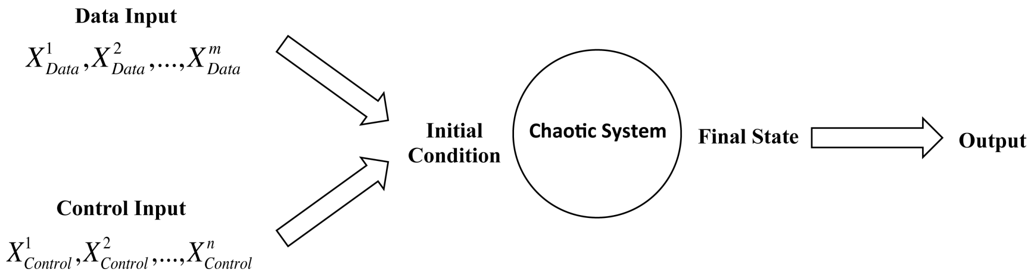

Data Input
$X^{1}_{Data}, X^{2}_{Data}, \ldots, X^{m}_{Data}$
Control Input
$X^{1}_{Control}, X^{2}_{Control}, \ldots, X^{n}_{Control}$
Initial Condition
Chaotic System
Final State
Output

Fig. 3.1. Schematic of chaotic computing model. Inputs are mapped to an initial condition of the chaotic system working as a computing engine, and the final state of the chaotic system is decoded to output.

A schematic of this computing model is shown in Fig. 3.1. The encoding map maps different sets of the inputs to different points on unstable manifold of the chaotic system and these points are used as initial conditions for the chaotic system. Since the system is on the unstable manifold, the orbits of the chaotic system are very sensitive to the inputs and the orbits dramatically change with just a one-bit change in the control input. Thus control inputs can select a chaotic logic function. To evaluate which digital function is selected with a particular control input, one notes the association of this control input with the logic function and then enumerates all possible combinations of data inputs to construct the truth table of the function.

By changing the control input and repeating this procedure (of constructing the truth table of the digital function), one may observe a second digital function different (with high probability) from the first one. This is the meaning of the *reconfigurability* of chaos computing. By using all possible control inputs and finding the type of function that the chaotic system implements, I obtain the full instruction set of the chaotic system [70].

So far different implementations for chaos-based computing have been introduced [57,58,59,60,61,62]. These implementations were mainly for proof of concept, showing that the idea of chaos based computing is practically possible and realizable. As an example, in Fig. 2.2 a picture of a circuit that I designed and built in [61] to implement chaos computing model is depicted. This realization of chaos computing, which is called a chaos based logic block, is able to reconfigure to construct any two input, one output digital functions. In this chaos-based logic block there is a chaotic Chua circuit which works as computing engine.

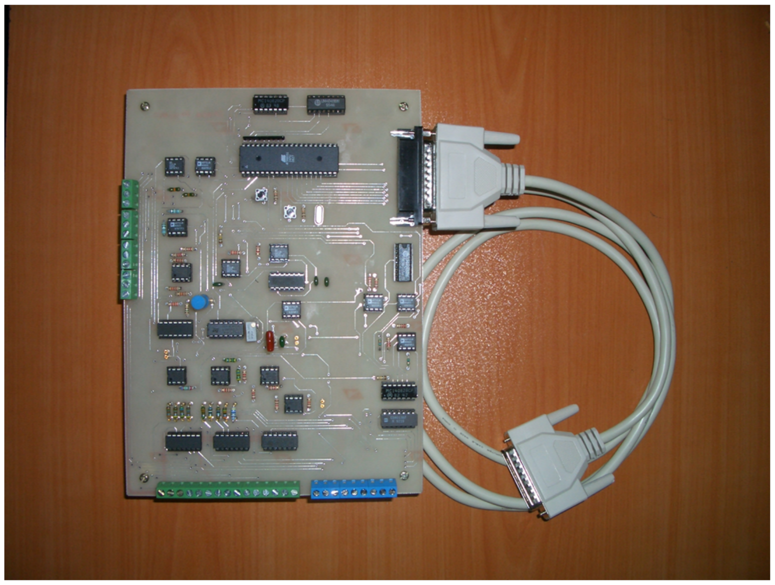


Fig. 3.2. Chua circuit-based logic block.

In what follows I address the important remaining questions: Why do we observe a specific form of logic function from a chaotic dynamic system? What are all the possible logic functions that we can obtain from any given chaotic system? How can we connect computation to the dynamics of chaos computing? In the next part these questions are addressed by connecting the chaos computing model to the dynamics of the chaotic system.

CHAPTER 4

DYNAMICS AND COMPUTATION

Let ***x*** be the dynamic state of a chaotic system and let the chaotic discrete evolution of the system be governed by the dynamical equation:

$$x_{p+1} = f(x_p) \tag{6}$$

The aim is to compute directly the spectrum of functions that a given chaotic system can implement and the robustness of these functions against noise from the dynamical Eq. (6).

The description of a low-dimensional chaotic system in terms of unstable periodic orbits, which is known as *periodic orbit theory*, is a powerful tool for the analysis of chaotic systems [64,65,66,63]. Periodic orbit theory is an efficient approach to study a chaotic dynamical system in terms of the fundamental orbits of its attractor [64]. Here I explain, model, and study chaos computing in terms of these basic periodic orbits.

Periodic orbits were introduced into the theory of dynamical systems by Poincare, and they have played a primary role in the mathematical work on dynamical systems ever since [64,65,66]. Periodic orbits provide a detailed, invariant characterization for deterministic low dimensional dynamical systems [64,65,66]. As a result, explaining chaos computing in terms of these periodic orbits has profound theoretical consequences.

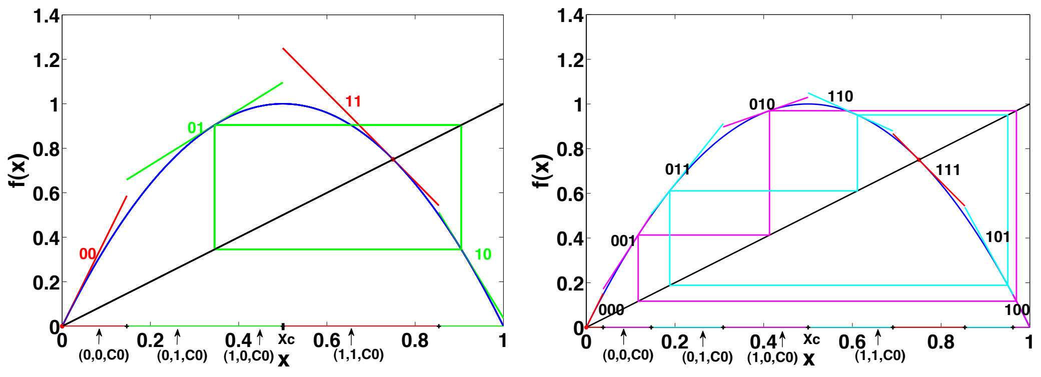
A chaotic system is composed of an infinite number of unstable periodic orbits (UPOs) [67]. It is known that a collection of short-period UPOs is enough to obtain a very precise approximation of a sufficiently low dimensional chaotic system [64,65,66]. Here I approximate the chaotic system with an appropriate collection of short period orbits to estimate the computational functionality and robustness of the chaotic system. As was mentioned in chapter 3, during step 2 of computation, the chaotic system undergoes a specific number of iterations, which I denote as p. I claim that, in one dimensional unimodal chaotic maps where the critical point x_c is mapped to unity and $f(0)=f(1)=0$, for the p iterations that the chaotic system undergoes, approximating the chaotic system by all of its UPOs of length $p+1$ is enough to determine the function set of the chaotic system and to approximate the robustness of these functions against noise. This method works for any other chaotic system where all symbolic sequences are admissible and therefore the topological entropy is $ln(2)$. But in other chaotic systems we might need to use slightly higher length UPOs to model the system. This case will be studied in the Gaussian map example.

In a unimodal map, where the height of the map is unity and where $f(0)=f(1)=0$, there are 2^p different unstable periodic points of order p, including repetition of periodic points of lower order [68]. For example, there are 24 unstable periodic points of period 4, which includes two unstable fixed points and two unstable periodic points (one unstable periodic orbit) of period 2. Thus in a unimodal map of height unity, there are exactly 24 possible symbolic sequences of length 4, and for each symbolic sequence there is a neighborhood of initial conditions where all the initial conditions have the same four-symbol iterates. Therefore there is a one-to-one relationship between UPOs and the neighborhood of similarly-behaved initial conditions. The same argument is correct for any other chaotic system that has no forbidden symbolic sequence or, equivalently, whose topological entropy is *ln*(2) [68]. Fig. 4.1(a) shows how all periodic orbits of length two produce a polygonal approximation of the unimodal map. The unimodal map has two period-1 unstable fixed points, which are at the intersection of the map and the identity line, and two period-2 unstable fixed points. Two repetitions of the period-1 unstable fixed points are considered as periodic orbits of length two as well.

Because the behavior of the dynamical system in the neighborhood of each of these points may be approximated linearly, the unstable fixed point and nearby points lying on a straight line are a good approximation of the dynamics near that unstable fixed point. If we have sufficient numbers of these linear approximations, we can approximate the map in its entirety. Therefore each fixed point neighborhood is one of the four faces of the polygonal (piecewise) approximation for the map, as illustrated by red (period-1) and green (period-2) tangent lines in Fig. 4.1(a).

As explained above, each face of the approximation is composed of an unstable fixed point or periodic point, plus all close-by points. These points are those whose Jacobian is qualitatively similar. Notice that projecting each face of the polygonal approximation on the *x*-axis results in a neighborhood around each periodic point, where all the initial conditions within this neighborhood symbolically (in the *symbolic dynamics* sense) behave the same as the periodic orbit. In unimodal maps the critical point, x_c, can be used for partitioning of state space and assigning symbolic itineraries to initial conditions. As an example, the *symbolic itinerary* for $x = x_0$ is $X_0X_1X_2 \dots X_k$, where each succeeding digit in the itinerary denotes the next iteration of the map. I (arbitrarily) choose $X_i = 0$ if

$f^{(i)}(x_0) < x_c$ and $X_i = 1$ if $x_c < f^{(i)}(x_0)$. Therefore, each periodic point locally explains the symbolic behavior of the initial conditions around itself during a specific number of iterations of the map. More specifically, UPOs of length *p+1* represent the symbolic behavior of nearby orbits during the first *p* iterations of the chaotic map. Furthermore, the measure of the robustness against noise of each periodic orbit in terms of eigenvalues is a good approximation for the robustness of orbits around it until the *p*th iteration of the chaotic map.



a b

Fig. 4.1. The figure at the left (right) shows how UPOs of length two (three) and the neighborhoods around them can be used to determine the functionality of the system when it undergoes one (two) iteration(s). Consider the left graph and recall that I have chosen $X_i = 0$ if $f^{(i)}(x_0) < x_c$ and $X_i = 1$ if $x_c < f^{(i)}(x_0)$. Any set of initial conditions in the area denoted “00” will return to that area, generating the symbolic itinerary 00. An initial condition in the region denoted “01” will start below x_c but the next iteration will take it above x_c, thus generating the symbolic itinerary 01. Similarly for the other regions. Note that some UPO neighborhoods are not used to implement the function of interest here.

In Figure 4.1(a) on the *x* axis, the neighborhoods of initial conditions that symbolically behave the same as the UPOs are denoted in the same colors. As an example, the first neighborhood on the *x* axis, which is illustrated by red color, contains the initial conditions that symbolically behave the same as the UPO at 0 and all of which produce the symbolic itinerary 00 when they evolve under the chaotic map, i.e., for any initial condition in this neighborhood x_0, $x_0<x_c$ and $f(x_0)<x_c$. In Fig. 4.1(b) a different polygonal approximation for the chaotic map using period three UPOs is illustrated. This approximation is composed of two unstable fixed points and two new UPOs of period three, resulting in an eight-faced polygonal approximation. Two faces of the polygonal approximation are delineated by the two unstable fixed points of the chaotic system, and the remaining six faces are related to two unstable periodic orbits of period three, each unstable fixed point of the periodic orbit centering a face. In these two approximations, the boundaries between neighborhoods on the *x*-axis are the preimages of the critical point, x_c.

As described above, chaos computation encodes the data as well as the control inputs to form the initial conditions, next evolving the chaotic system from these initial conditions for some number of iterations, and lastly decoding the final state to obtain the output of the computation. The technique for obtaining the instruction set of a chaotic system for use in computation is as follows: when the chaotic system is iterated *p* times, approximate the chaotic system by its UPOs of length *p+1*. Determine in which UPO neighborhood the encoding map places each initial condition and the characteristic itinerary (e.g., (0,1) in Fig. 4.1(a)) for that neighborhood. The last symbol of this itinerary represents the output of the computation for those specific data and control inputs. By applying this technique for all combinations of data and control inputs, instruction set of the chaotic system can be directly obtained.

Consider as an example a one-humped map, as shown in Fig. 4.1(a). This might be the logistic map or any other similar map. Now let us consider that this chaotic system undergoes one iteration and that, as explained above, I will be using period-2 UPOs. If I denote the selected control inputs from Eq. 5, $X_{Control}^{1}, X_{Control}^{2}, \ldots, X_{Control}^{n}$, collectively as C_0, all four possible initial conditions produced by the encoding map are illustrated on the x-axis. Here, the aim is to implement a two-input function; therefore we have four different combinations of initial conditions. In this example, the encoding map encodes the data inputs (0,0) to the point (0,0,C_0), (0,1) to (0,1,C_0) , (1,0) to (1,0,C_0), and (1,1) to (1,1,C_0). The initial condition (0,0,C_0) falls in the first neighborhood on x-axis, which produces 0 after one iteration. Therefore the output of the computation for the (0,0) input is 0. The second and third initial conditions, (0,1,C_0) and (1,0,C_0), fall in the second neighborhood, which is represented by the periodic orbit 01, and so the output of the computation for these two inputs is 1. The last initial condition, (1,1,C_0), settles in the third neighborhood, which corresponds to the 11 periodic orbit. Therefore the output of the computation will be 1. As a result control inputs C_0 thus constructs an OR gate. This procedure can be repeated for other control inputs to obtain the instruction set of any given chaotic system.

The instruction set of the chaotic system for other iteration numbers can be obtained in a similar way. As a further example, I show this for iteration number two in Fig. 4.1(b). From the figure it is clear that the set of control inputs

C_0 constructs a function that produces the output 1 when the inputs are (0,0) and (0,1), but produces the output 0 when inputs are (1,0), and (1,1).

Here functionality of a chaotic system in doing computation is determined based on UPOs and their neighborhoods. In next chapter I demonstrate how UPOs and UPO model can be used to estimate the robustness of these computational functionalities.

CHAPTER 5

ROBUSTNESS AGAINST NOISE

UPOs can also help us in approximating the robustness against noise of the chaotic system while doing computation. The robustness of each UPO against noise can be measured by evaluating its Jacobian matrix. In our 1-D case, the measure of the robustness of each UPO is simply the product of the slopes of all the tangent lines at each UPO. For example, for the dynamical system $x_{n+1}=f(x_n)$, the robustness against noise for a UPO of length $p+1$, $x_0^{UPO}, x_1^{UPO}, \dots, x_p^{UPO}, x_{p+1}^{UPO} = x_0^{UPO}$ is $\lambda_0 \times \lambda_1 \times \dots \times \lambda_p$ where, $\lambda_i = f'(x_i^{UPO})$ [70].

This robustness measure for each UPO can be used as an approximation for the robustness of orbits that start in the neighborhood of the UPO. To construct a specific function, the chaotic system maps the initial conditions produced by the encoding map to the final states. Therefore to evaluate the robustness of each function in doing computation, the robustness for each orbit needs to be obtained, and the overall robustness of the function is the robustness measure of the least robust orbit, i.e. the worst case.

We assume the noise to the system is additive, $x_{n+1} = f(x_n) + D\varepsilon(t)$ where D is the intensity of the noise and $\varepsilon(t)$ is the white noise. We also assume the noise is approximately Gaussian white noise with zero mean and unit variance, $\varepsilon(t) = N(0,1)$. Earlier we claimed that, when the chaotic system iterates p times, approximating the chaotic system by its UPOs of length $p+1$ is sufficient to determine the robustness against noise of the functions implemented by the chaotic system. To demonstrate this, let the chaotic system f iterates p times from a given initial condition, x_0, producing the noisy orbit:

$$\begin{aligned} &x_0+\varepsilon(0), f(x_0+\varepsilon(0))+\varepsilon(1), f(f(x_0+\varepsilon(0))+\varepsilon(1))+\varepsilon(2),\ldots \\ &\ldots, f(\ldots(f(f(x_0+\varepsilon(0))+\varepsilon(1))+\varepsilon(2))+\ldots)+\varepsilon(p) \end{aligned} \tag{7}$$

By use of the polygonal approximation by UPOs of length $p+1$, the orbit can be approximated by:

$$\begin{aligned} &x_0+\varepsilon(0), f(x_0)+D\lambda_1\varepsilon(0)+\varepsilon(1), f^2(x_0)+D\lambda_1\lambda_2\varepsilon(0)+D\lambda_2\varepsilon(1)+\varepsilon(2),\ldots \\ &\ldots, f^p(x_0)+D\lambda_1\lambda_2..\lambda_p\varepsilon(0)+D\lambda_2..\lambda_p\varepsilon(1)+\ldots+\varepsilon(p) \end{aligned} \tag{8}$$

where $\lambda_i = f^{'}(x_i^{UPO})$ and x_i^{UPO} is an iterate of the UPO into whose neighborhood $f^{(i)}(x_0)$ places the iterate of the initial condition, x_0.

Since $\varepsilon(t)$ is a normal Gaussian random variable, $\varepsilon(t)=N(0,1)$, the deviation of the final state in the noisy case from the original final state will be a Gaussian random process:

$$D\lambda_1\lambda_2..\lambda_p\varepsilon(0)+D\lambda_2..\lambda_p\varepsilon(1)+\ldots+\varepsilon(p)=N(0,D^2\lambda_1^2\lambda_2^2\lambda_3^2..\lambda_n^2+D^2\lambda_2^2\lambda_3^2..\lambda_n^2+\ldots+D^2\lambda_n^2+D^2) \tag{9}$$

Let *y* be the minimum distance of the noiseless final state, $f^{(p)}(x_0)$, from the boundaries of the neighborhood in which it resides. If the deviation introduced by the noise exceeds this value, the orbit will enter another neighborhood, and it may result in an incorrect (undesired) output symbol. Therefore the output symbol is robust to noise only if the noise cannot move the final state out of the neighborhood where it settles. If z is a Gaussian random variable, $z=N(0,\sigma)$, the probabilities that z is greater than σ, 2σ, and 3σ are

84.2%, 97.8%, and 99.9% respectively. We observe that the probability of $3\sigma < z$ is just 0.1%. This fact suggests that if $3\sigma < y$, where σ is the standard deviation of ε, then the outcome will be robust against this noise 99.9% of the time. Since y is the minimum distance of final state, $f^{(p)}(x_0)$, from the boundary of the neighborhood, it can be easily computed. Therefore the noise intensity should be limited by:

$$D < D_{\max} \equiv \frac{y}{3 \times \sqrt{\lambda_1^2\lambda_2^2\lambda_3^2 ... \lambda_n^2 + \lambda_2^2\lambda_3^2 ... \lambda_n^2 + ... + \lambda_n^2 + 1}} \qquad (10)$$

Therefore the symbol of the final state is robust against noise when the signal to noise ratio (SNR) is greater than $20 \log \frac{A_{rms}}{D_{\max}}$, where A_{rms} is the root mean square of $f(x)$ over all *x*. Notice that because of the ergodicity of the chaotic map, f, A_{rms} *does not depend* on the selection of the initial condition.

Here we derived a measure of the robustness of an orbit against noise. To compute a robustness measure for a function, we apply the procedure to all orbits produced by the encoding map and set the lowest allowed SNR (highest *D*), as determined over all the orbits.

CHAPTER 6

EXAMPLES FOR DETERMINING THE TYPE OF COMPUTATION FROM DYNAMICS

6.1. Logistic Map

As an example, consider the functionality of the logistic map for doing computation and approximate the robustness of the resulting functions against noise. In this example we assume that an additive noise perturbs the dynamics as follows:

$$x_{n+1} = 4x_n(1 - x_n) + D\varepsilon(t) \tag{11}$$

A simple digital-to-analog converter with 10 binary digital inputs will be used as the encoding map. Two inputs are allocated for data, which enables us to construct two input functions, and the eight remaining inputs are used as controls to reconfigure the chaotic system by morphing between different functions. As the first step of the 3-step computing algorithm, the two binary data inputs and 8 binary control inputs are each encoded to either 0 or 1, yielding a combined initial value in [0, 1), as follows:

$$x_0 = (0.I_1 I_2 C_1 C_2 \ldots C_8)_{base 2} \tag{12}$$

where I_1, I_2 are the two binary data inputs, and C_1, C_2, …,C_8 are the eight control inputs.

At the second step of the algorithm, we allow the logistic map to undergo different numbers of iterations in order to determine the instruction set for each of those different numbers of iterations.

As the last stage of the computing model, the final state of the system is decoded to obtain the output of the computation as follows:

$$output = \begin{cases} 0 & if\ x \le 0.5 \\ 1 & if\ x > 0.5 \end{cases} \tag{13}$$

where *x* has obviously been converted to base 10. We have computed period 2, 3, 4, 5, 6 and 7 UPOs for the logistic map. Then we have found the aforementioned neighborhoods around these UPOs and have computed the robustness of these UPOs against noise. Then for each iteration, e.g., *p*-1, we approximate and model the chaotic logistic map with period *p* UPOs. By use of this model we directly compute the instruction set of the chaotic logistic map when it undergoes *p* iterations. The results are listed in Table I for different values of *p*, $1 \le p \le 6$.

In Table I each instruction set consists of 4-tuples, the first element being the type of function that the logistic map constructs. The format that we use for identifying each of these functions is as follows: Table II presents the truth table of a sample function. We denote this function by a function number defined as $2^3 \times O_3 + 2^2 \times O_2 + 2^1 \times O_1 + 2^0 \times O_0$. Based on this definition, a chaotic system would present a 2-input AND gate (with outputs 1000) as function number 8 and a 2-input OR gate (with outputs 1110) as function number 14.

The second element of the 4-tuple is the control inputs that construct this sample function. There are 8 binary digital control inputs to the system, so the control inputs are numbered from 0 to 255. To evaluate the accuracy of our method in obtaining the functionality obtainable from a chaotic map, we have applied the computed control inputs to the logistic map and in practice we have observed computationally that they construct the same functions that were predicted based on the periodic orbit approximation.

The third element of each 4-tuple is the computed SNR using the UPO approximation. To examine the precision of these SNRs, we experimentally compute the SNR (called SNR_e) for all functions and report it as the forth element of each 4-tuple. To compute these experimental SNRs, we statistically compute the probabilities that the desired functions are constructed, when the noise intensity is changed. For this example, we choose the noise intensity such that a given threshold value for noise intensity results in 99.9% success in constructing the desired function. We then use this same noise intensity to compute SNR_p, based on the formula $20\,log\frac{A_{rms}}{D_{Threshold}}$. In order to facilitate understanding of the last two elements of the 4-tuples, the estimated SNR and the experimental SNRs, we compute the statistical mean and variance of the differences between these two SNRs, defined as $r = SNR_p - SNR_e$, for different iteration numbers, *p-1*. As explained above, SNR_p is the predicted SNR based on UPOs of order *p* and SNR_e is the experimental SNR. The results are plotted as solid lines in Fig. 6.1. The overall trend is that with increasing iteration number, the mean and variance of the error signal grow. The predicted SNRs are not very accurate, since we approximate a large portion of the map, *f*, or the iterated map, $f^{(p-1)}$, with a straight line. To obtain more accurate SNR predictions, we need more precise modeling and approximation. In this example we have computed all the UPOs up to period-7, so an alternative, more precise (and no additional cost) approximation would use these already-computed period-7 UPOs for a better calculation of the SNRs for lower iteration numbers as well.

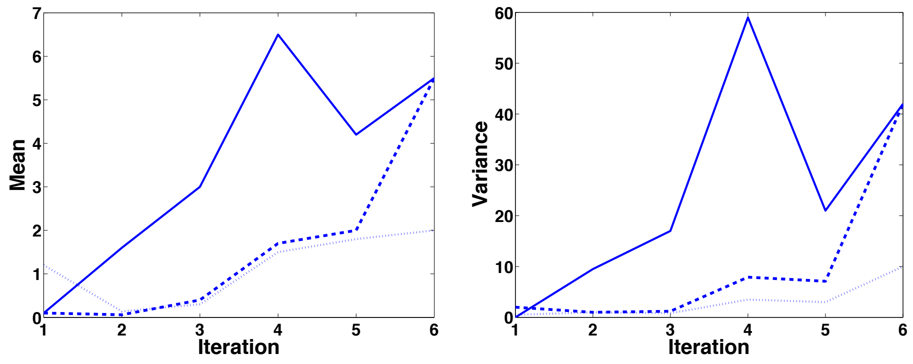


Fig. 6.1. Statistical measures, mean and variance, of the error in estimating robustness of different instructions against noise are reported. The error is the difference between the estimated SNR and the experimental SNR for each instruction. The mean of these errors at each iteration is reported in the left panel, and the variance of the error at each iteration is presented in right panel. The solid lines denote cases where, for (*p*-1) iterations of the map, period *p* UPOs are used for modeling. Dashed lines denote the means and variances when period-7 UPOs are used for predicting the SNR. Dotted lines show the means and variances of the difference *r*, where linearization is performed along each orbit.

The mean and variance of the error, $r = SNR_7 - SNR_e$, where SNR_7 is the predicted SNR based on UPOs of order 7, is computed for different iteration numbers. The results are presented in Fig. 6.1 by dot-dashed lines. We observe that when the iteration number is less than 6, these predicted SNRs are considerably more precise than the previous predicted SNRs, because of more accurate modeling and approximation. Based on Fig. 6.1, we observe that modeling the chaotic orbits by their nearby UPOs results in a very good approximation of the symbolic behavior of the orbits during limited iteration of the chaotic map. This observation follows the main claim of periodic orbit theory: a collection of short-period UPOs is enough to obtain a very precise approximation of a sufficiently low dimensional chaotic system [64].

Finally, to examine the accuracy of the approximated SNRs by use of UPOs, we approximate SNRs directly based on the slopes of the orbits, starting from the chosen initial conditions. Thus, instead of finding a nearby UPO and using its robustness measure, we compute directly the slope of the main orbit at various points on the orbit and we use these slopes directly in the formula $D_{max} = \frac{y}{3\times\sqrt{\lambda_1^2\lambda_2^2\lambda_3^2\ldots\lambda_n^2+\lambda_2^2\lambda_3^2\ldots\lambda_n^2+\cdots.+\lambda_n^2+1}}$, where $\lambda_i = f^{(i)\prime}(x_0)$ and x_0 is the initial condition produced by the encoding map. The mean and variance of the error is plotted in Fig. 6.1 by dashed lines. We see that using UPOs of order 7 for predicting SNRs is as precise as using direct slopes, when the iteration number is less than 6.

6.2. Gaussian Map

As a second example, we determine the functionality of the Gaussian map for doing computation and estimate the robustness of the resulting functions against noise. The Gaussian map is studied in detail in [69]. Again, in this example we assume that an additive noise perturbs the dynamics as follows:

$$x_{n+1} = e^{-bx_n} + c + D\varepsilon(t) \tag{14}$$

The phenomenon of chaos is observed in this map at some parameter values [69]. In this work I set b = 6.5 and *c = -0.54* in order to make the Gaussian map chaotic. The chaotic attractor of the Gaussian map lies in [-0.28, 0.5]. Similar to the logistic map example, a simple digital-to-analog converter with 10 binary digital inputs, two inputs for data and eight inputs for control, will be used as the encoding map. As the first step of the 3-step computing method, the combination of two binary data inputs and 8 binary control inputs are, with an initial value in [-0.28, 0.5), as follows:

$$x_0 = -0.28+0.78 \times (0.\, I_1 I_2 C_1 C_2 \ldots C_8)_{base2} \tag{15}$$

where I_1,I_2 are the two binary data inputs, and $C_1,C_2,\ldots,C_8$ are the eight control inputs. Notice that the coefficient value, -0.28, and the additive value, 0.78, are inserted to insure that the initial condition is situated inside the attractor.

At the second step of the algorithm, we let Gaussian map undergo different numbers of iterations in order to determine the instruction set at each iteration number.

As the third and last stage of the computing model, the final state of the Gaussian map is decoded to obtain the output:

$$output = \begin{cases} 0 & if\ x \le 0 \\ 1 & if\ x > 0 \end{cases} \tag{16}$$

There is an important difference between the logistic map and the Gaussian map examples. When the bifurcation value of the logistic map is 4, for any symbolic sequence $X_0, X_1,\ldots,X_p$ there is a unique UPO of length *p+1* that has the same symbolic itinerary. This one-to-one relationship between any possible symbolic sequence and a unique UPO describes any other one-humped map, where the attractor is between [0, *b*] and the critical point x_c is mapped to *b* [68]. Therefore the collection of all UPOs of length *p+1* can model the behavior of the chaotic map over the next *p* iterations. But the Gaussian map does not have this property and there are some neighborhoods of initial conditions with admissible symbolic itineraries of length *p* for which there is no UPO of length *p+1* with the same symbolic itinerary. But we know that, since the UPOs are dense over the chaotic attractor, there is therefore at least one UPO that comes inside the neighborhood and which can model this portion of the attractor during the next *p* iterations. Therefore we can easily overcome the problem by using higher order UPOs, such as *p+2* or *p+3*, to model the next *p* iterations of the map. All we need to do is to compute the pre-images of the critical map to find the neighborhood of initial conditions that symbolically behave the same during limited iterations of the map. Then we compute the UPOs until we can find at least one UPO in any neighborhood. This collection of UPOs can be used to model the chaotic map over a limited number of iterations. In the Gaussian map example, we observe that UPOs of length eight are enough to model the attractor during any iteration up to six iterations. By use of this model we directly compute the instruction set of the chaotic logistic map when it undergoes *p* iterations. The results are listed in Table III for different values of *p*, $1 \le p \le 6$. The format of data in Table III is the same as the format in Table I. We observe that, in a noise-free simulation, this technique determines the instruction set of the chaotic system precisely. Also simulation results illustrate that after modeling the Gaussian map by period-8

UPOs, the robustness of the instructions against noise are predicted with very high precision.

Table I: Instruction set of the logistic map for different iteration numbers.

P	Instruction Set
1	{(6,129,39.39dB,39.9dB), (7,255,26.74dB,26.11dB), (14,0,26.67dB,26.31dB)}
2	{(5,255,32.95dB,34.2dB), (9,123,30.73dB,31.8dB), (10,0,35.48dB,33.7dB), (11,52,42.72dB,48.79dB), (13,207,44.89dB,46.3998dB)}
3	{(2,89,50.46dB,57.09dB), (3,53,38.59dB,38.79dB), (4,165,52.28dB,52.59dB), (5,255,45.29dB,46.89dB), (6133,36.32dB,42.2dB), (10,0,47.52dB,45.79dB), (11,20,46.80dB,47.79dB), (12,211,37.62dB,42dB), (13,233,45.92dB,51.89dB)}
4	{(1,228,50.83dB,50.39dB), (3,200,52.73dB,58.89dB), (5,93,40.41dB,45.19dB), (6,126,44.13dB,46.79dB), (7,114,49.45dB,59.49dB), (8,24,49.45dB,59.89dB), (9,20,48.56dB,59.89dB), (10,17,40.40dB,42.59dB), (11,8,47.01dB,51.29dB), (12,59,50.89dB,62.2dB), (13,62,49.98dB,61.89dB), (14,144,53.41dB,54.89dB)}
5	{(1,177,54.97dB,59.69dB), (2,98,50.07dB,57.39dB), (3,106,47.72dB,51.89dB), (4,43,50.83dB,53.09dB), (5,170,57.96dB,63.39dB), (6,35,54.85dB,60.99dB), (7,29,62.30dB,67.79dB), (8,80,54.69dB,57.79dB), (10,85,59.64dB,64.49dB), (11,195,47.40dB,48.69dB), (12,146,48.20dB,47.79dB),(13,58,47.30dB,48.19dB), (14,228,63.97dB,70.39dB), (15,128,55.34dB,58.19dB)}
6	{(0,110,60.66dB,62.69dB), (1,106,58.20dB,58.99dB), (2,232,61.11dB,62.59dB), (3,140,58.56dB,69.69dB), (4,66,59.67dB,71.59dB), (5,68,55.76dB,62.19dB), (6,35,66.87dB,69.19dB), (7,173,55.50dB,57.89dB), (8,56,56.42dB,63.49dB), (9,53,53.76dB,61.29dB), (10,92,57.01dB,63.09dB), (11,46,55.98dB,60.39dB), (12,118,60.55dB,65.19dB), (13,210,59.97dB,60.49dB),(14,123,55.1dB,64.69dB), (15,126,54.21dB,62.39dB) }

Table II: Truth table of a typical two input, one output function.

Data inputs	Output
00	O_0
01	O_1
10	O_2
11	O_3

Table III: Instruction set of the gaussian map for different iteration numbers.

P	Instruction Set
1	{(7,133,18.90dB,19.7dB), (15,0,45.79dB,45.89dB)}
2	{(9,0,23.62dB,25.50dB), (12,192,33.15dB,34.20dB), (13,118,28.42dB,28.80dB)}
3	{(6,0,26.47dB,26.40dB), (7,88,36.24dB,37.2dB), (11,246,25.50dB,25.80dB), (15,147,36.4dB,36.5dB)}
4	{(6,191,34.45dB,35.5dB), (7,246,40.37dB,40.89dB), (10,118,55.71dB,56.69dB), (11,88,29.73dB,33.60dB), (14,138,35.51dB,36.4dB), (15,0,51.85dB,51.79dB)}
5	{(4,89,37.62dB,39.89dB), (5,106,36.81dB,38.6dB), (9,187,41.931B,42.29dB), (12,47,35.42dB,43.19dB), (13,30,40.96dB,40.39dB), (15,0,36.88dB,37.3dB)}
6	{(2,6,45.40dB,44.99dB), (3,22,45.96dB,48.69dB), (7,45,38.81dB,49.39dB), (8,244,44.09dB,47.19dB), (9,255,62.52dB,62.39dB), (10,223,41.67dB,51.49dB), (14,213,47.67dB,48.79dB), (15,79,39.77dB,36.1dB)}

In this chapter I have demonstrated how chaotic computation could be explained, modeled, and predicted in terms of the dynamics of the underlying chaotic systems. Unstable periodic orbits of the chaotic system were used first to model it and then to approximate it. These periodic orbits and the polygonal approximations based on them can be used for obtaining the computational functionality (the instruction set) of the system. In this way I have elucidated the deep connection between the structure of the system dynamics and the system's ability to perform computation. This connection intimately depends on the periodic orbit structure of the system.

Low-period periodic orbits are experimentally extractable from time series. This contributes practical importance to our ability to explain chaos computing in terms of basic periodic orbits; e.g., it enables us to predict and determine the instruction set that a chaotic system can implement and the stability of those instructions against noise just by having access to a time series from the chaotic system. The next chapter focuses on this subject and I demonstrate given a time series from an unknown chaotic system how I can determine and estimate the functionality of the system in doing computation.

CHAPTER 7

DERIVING INSTRUCTION SET FROM A TIME SERIES

In this chapter I assume I am given only a time series from a chaotic system, e.g., z_n, n = 1,2,3,... No further information about the underlying chaotic dynamics is provided. The aim is to obtain the functions that can be constructed from the chaotic dynamics and to understand the robustness of these functions against noise.

To do this, (A) recover a generating partition from the time series to be able to define its symbolic itineraries. (B) For a chaotic system which is iterated *p* times, find a suitable collection of UPOs as described in chapter 5 (C) Find the neighborhood of each UPO. Each UPO will then represent its neighborhood in an approximation of the chaotic system. (D) Find the eigenvalues of the UPOs. (E) Design local predictors for the time series to predict how close the orbits get to the neighborhood boundaries. This is required for estimating robustness. By extracting all of this dynamic information from the time series, I can obtain the library of functions for the underlying chaotic system as well as their robustness, using techniques introduced in chapter 4 and [70] as detailed below [71,72].

7.1. Extracting a Generating Partition from the Time Series

To estimate a generating partition from the time series, different methods and approaches have been introduced. [73,74] Here I introduce a novel technique using the topological entropy to locate the generating partition. The topological entropy is a basic measure of how much flexibility there is in the dynamics, how many different kinds of patterns it can produce, and how much the past of the process constrains its future behavior.

To begin, partition the state space of the attractor and label each partition with an unique symbol. The passage of the system through its state space then generates strings of symbols. The topological entropy is defined as following:

$$h(A) = \lim_{N \to \infty} \frac{\log\left|W_N(A)\right|}{N} \tag{17}$$

where N is the size of a given string using a particular partitioning A of the attractor, $W_N(A)$ is the collection of all possible N-strings which appear in the attractor, and the vertical bars indicate the size of this collection.

By applying a generating partition, a partitioning scheme which maximizes the topological entropy, to create symbolic itineraries for the orbits, we preserve the unpredictability of the chaotic dynamics. I use this definition to find the generating partition from the time series. Changing partitions to find the maximal topological entropy rate is an optimization problem:

$$h(A^*) = \underset{over\ A}{max}\left(\lim_{N \to \infty} \frac{\log\left|W_N(A)\right|}{N}\right) \tag{18}$$

where A^* is the generating partition.

In the example provided in chapter 8, I use a hill-climbing algorithm to maximize the topological entropy. However this technique is not durable for higher order systems or for systems with more complicated (e.g., higher-dimensional and/or fractal) boundaries. In these cases more sophisticated techniques are required. [73,74]

After constructing a generating partition, we can assign symbolic itineraries to orbits and, based on that symbolization, we can define the logic and the logical outputs.

7.2. Extracting UPOs from Time Series

A chaotic system is composed of unstable periodic orbits, and it is known that a collection of short period UPOs is enough to obtain a very precise approximation of the chaotic system [64,65,66]. In [70] it is demonstrated that, by approximating the chaotic system with an appropriate collection of short-period unstable periodic orbits, the functionality and robustness of the chaotic system in computation can be obtained. Ref. [70] assumes that the dynamic equations are given, so UPOs can be computed analytically or numerically. In this work I assume that I do not have access to the underlying dynamics; however the importance of using UPOs is that they can be easily extracted from a time series. Here to extract UPOs from a time series, I follow the classic technique introduced by Cvitanovic in [63]. To find unstable periodic orbits of length *p*, I monitor the time series z_n and evaluate if $|z_n - z_{n+p}| < \varepsilon$, where ε is some small number defining the neighborhood of the UPO. Here I use the Euclidian norm to measure the distance between z_n and z_{n+p}. If the inequality holds for a specific *n*, say n_0, then the series $z_{n_0}, z_{n_0+1}, z_{n_0+2}, \dots, z_{n_0+p-1}$ will be recorded as a UPO. Internally, the algorithm uses a parameter σ, which is the minimum distance for two observed UPOs to be recorded as distinct; otherwise they will be recorded as a single UPO.

7.3. Extracting the Neighborhoods of the UPOs

In chapter 4 polygonal approximation is used for modeling a chaotic system, where each face of the approximation was identified and approximated by a UPO. Each face of the approximation is composed of an unstable fixed point or unstable periodic point plus all close-by points with qualitatively similar Jacobians; i.e., whether the Jacobian flips or does not flip along the unstable eigendirection. The projection of each face of the polygonal approximation onto the x-axis results in a neighborhood around each periodic point. In that neighborhood all the initial conditions symbolically behave the same as the periodic orbit for some minimum number of iterations. In section and [70], since the dynamical equations were in hand, these neighborhoods could be computed analytically or numerically from the equations. But here I need to derive them from the time series. To find the set of initial conditions that symbolically behaves the same as a UPO with a symbolic itinerary $c_0,c_1,c_2,\ldots,c_{p-1}$, I trace the itineraries of candidate initial conditions (which we can select from any point in our time series) and, if any possess the same itinerary as the UPO, I add them to our set. The closure of this set is one face of the polygonal approximation as modeled by the UPO.

After extracting all the faces of the polygonal approximation for the underlying chaotic system, I examine the approximation to ensure that it covers the entire attractor. As mentioned before, it may in some situations be necessary to use higher period UPOs to achieve this coverage.

At this point we can obtain the functionality of the system for doing computation. The technique I use here is the same is introduced in chapter 4 and

[70]. To summarize, the technique for obtaining the computational functionality of a chaotic system is as follows: when the chaotic system iterates *p* times, approximate the chaotic system by its UPOs of length *p+1* and determine the neighborhood of each UPO by tracing initial conditions to determine their symbolic behavior. The last symbol of the itinerary determined by the corresponding UPO is the computational output for those specific data and control inputs. By applying this technique for all combinations of data and control inputs, the computational functionality of chaotic system can be directly obtained.

7.4. Extracting the Eigenvalues and Estimating Robustness

To estimate the robustness against noise of the computational functionality of the system, we need the eigenvalues of the UPOs at each point of the periodic orbit and also the minimum distance of the final state of the orbit (which represents the output of the computation) from the partition boundaries, as described in chapter 4.

To compute the eigenvalues, I obtain a tangent linear map in the vicinity of each UPO by a least squares fit. The eigenvalues of this tangent map approximate the eigenvalues of the chaotic map. More specifically, to compute the eigenvalues at a UPO s_i, find all samples in the time series so that

$$\left| s_i - z_j \right| \leq r \tag{19}$$

where the Euclidian norm is used for measuring the distance between states. The matrix **H** representing the linear map is obtained by least mean square fitting such that

$$\mathbf{H} \times \left| s_i - z_j \right| \approx \left| s_{i+1} - z_{j+1} \right| \tag{20}$$

The eigenvalues of H are the desired eigenvalues at the point s_i.

For simplicity the dimension of the underlying chaotic system and the embedding dimension are both unity. Thus the slope of the tangent line at the point s_i, denoted by λ, replaces the matrix **H**. This is the same value that we need for Eq. 10.

7.5. Forecasting Chaotic Orbits to Compute the Minimum Distance from Partition Boundaries

To forecast a chaotic orbit starting from a given initial condition, different techniques have been introduced. [75]

To design chaotic systems from a given time series, I use local constant predictors. The main idea of local methods is to predict subsequent samples of a time series solely by use of nearby samples in a training time series. Nearby samples are defined as the samples that fall into some neighborhood of the state that we wish to forecast. I use the Euclidian norm as a metric to measure the distance between states in order to detect whether the training sample falls into the neighborhood or not. The size of the neighborhood is a predictor algorithm parameter that is adjusted during each simulation.

In local constant models the prediction is accomplished by averaging or integrating the behavior of the nearby trajectories. In the averaging method one notes the subsequent iterates of points in the neighborhood and averages them to obtain the predicted sample. In the integrating method instead of averaging the iterates, one measures the differences between the neighborhood points and their iterates at the next time step. Then I average the differences and add this value to the state in question to obtain the predicted value. Here I use the integrating method, since it is accurate for extrapolation as well as interpolation of the training data for forecasting [76].

There are two methods for predicting *p* steps ahead, direct prediction and iterated prediction. In the direct method a model is built to directly predict the state *p* steps in the future for the given time series. On the other hand, iterated prediction jumps one-step ahead *p* times. Iterated prediction is usually used because of its superior short-term accuracy. However, one should use this method with caution since medium- to long-term forecasts can be worse because of accumulated error in the input vector [76]. The choice is critically dependent upon the numerical details of the iteration algorithm.

In this case I implement the forecasting method as follows:

1- Use the time series $\{z_n\}, n = 0, 1, 2, \dots, 10^6$ as the training data set.

2- For predicting *p* steps ahead, find all the nearby training samples that fall inside a neighborhood around the current state. Use the Euclidian norm to determine whether the training samples are inside a neighborhood of size ε or not.

3- Measure the differences between the current value of the nearby training samples and their values *p* time steps ahead. Then average the differences and add this value to the current state to predict the value *p* steps ahead.

7.6. Putting It All Together

Chapters 7.1-7.5 extract all the information that we need in order to design a chaos-based computing system and to estimate both its functionality and its robustness against noise from a time series. In this section I examine all this extracted information in order to discover the computational properties of the underlying chaotic dynamics as well as its computational robustness. I summarize the process as follows:

1- Extract the generating partition (7.1).

2- Extract all UPOs of length p+1 (or, if needed, slightly higher length) for modeling the p iteration of the map (7.2).

3- Find the neighborhood around each UPO where the UPO approximates all points in this neighborhood (7.3).

4- Encode the data and control inputs via an encoding map to develop an initial condition for the underlying chaotic system. For each produced initial condition, determine in which UPO neighborhood it falls. We know the effect of iterating this UPO p times; therefore the chosen initial condition's final symbol is also known without iteration. For any control input, repeat this procedure with different combinations of data inputs to estimate the type of function that the underlying chaotic system will construct with a given control system. To compute the complete instruction set of the underlying chaotic system, repeat the procedure for different control inputs.

5- Find the slope (eigenvalue) of the tangent map at each point of the UPO (7.4). The result λ is used in Eq. 10 for estimating robustness against noise.

6- For any initial condition produced by the encoding map, forecast the final state of the underlying chaotic system after p iterations (7.5). Compute the minimum distance of this final state from the partition boundary. This gives us y, which is used in Eq. 10.

7- For any initial condition, estimate the robustness of the orbit against noise as was discussed in chapter 5. To compute a robustness measure for a specific function, apply the procedure for all orbits of the function, starting from different initial conditions produced by the encoding map and choose the highest required SNR, as obtained from individual orbits.

CHAPTER 8

EXAMPLE FOR DERIVING THE INSTRUCTION SET FROM A TIME SERIES

As an example, here I assume I am given a time series of length 10^5 generated from evolution of the logistic map (after transients have died out). However, for the purposes of this exercise, I use no prior knowledge about the underlying dynamics and all the required information will be extracted solely from the time series.

To derive the instruction set of the chaotic system and the robustness of these functions against noise, I follow the 5 stage algorithm introduced in chapter 7. I repeat the algorithm for different iteration numbers p = 1, 2, ..., 6.

1- To extract the generating partition from the given time series, I start from a random partition and compute the topological entropy obtained by the use of this initial partition. Then I apply a hill-climbing optimization algorithm to maximize the topological entropy by change of the initial partition. The optimization technique converges to the partition boundary x_b=0.495605, which is very near to the real generating partition of the logistic map which is x_b=0.5. Notice that, since the generating partition is not a function of iteration number of the chaotic system, there is no need to repeat this step again and again for different repetitions of the chaotic system.

2- Using the technique introduced in chapter 7.2, I extract all the UPOs of length *p+1*. I choose the neighborhood and distance parameters (defined in chapter 7.2) as $\varepsilon = 0.00003$ and $\sigma = 0.000031$.

3- Applying the technique explained in chapter 7.3, I obtain the neighborhoods around each UPO of length *p+1* such that the UPO approximates the behavior of all the points in its neighborhood during evolution. Examine the total approximation and, if necessary, increase the length to $p+2$ or higher and return to step 2. In the example shown in Table III, although it was not necessary to achieve the desired function set, I used UPOs of length 7 to obtain the highest possible robustness.

4- Knowing that the time series does not include any transient (non-stationary) behavior, by monitoring the time series I observe that the chaotic attractor is between x=0 and x=1. Therefore the encoding map should encode the data and control inputs to a point in this interval. The encoding map I use here is where $I_1,$ I_2 are two binary data inputs, and $C_1, C_2, ...,C_8$ are eight control inputs. Determine in which UPO neighborhood the encoding map places each initial condition and the characteristic If chaotic map iterates for p times, the $(p+1)^{th}$ symbol of the symbolic itinerary of UPO is the output of computation for this specific set of data and control inputs. By applying this technique for all combinations of data and control inputs, instruction set of the chaotic system can be directly obtained.

5- Using the technique introduced in chapter 7.4 I compute eigenvalues for all UPOs extracted from the time series.

6- For any initial condition produced by the encoding map, I forecast the state of the chaotic system starting at this initial condition and iterated p times. Then I compute the minimum distance y from the partition boundary. This distance is used to compute the robustness of the orbit as in chapter 7.5.

7- To compute a robustness measure for a specific function, I apply the procedure introduced in chapter 4 to all orbits of the function, starting from different initial conditions produced by the encoding map and choosing the highest required SNR obtained from individual orbits. A chaotic system can construct different functions by applying different control inputs.

The obtained instruction set is presented in Table IV for different values of *p*. In the table each instruction set consists of 4-tuples, the first element being the type of function that the logistic map constructs. The format that I use for identifying each of these functions is the same as I used for Table I. The first element represents function number based on Table II. The second element of the 4-tuple is the control inputs that construct this sample function. There are 8 binary digital control inputs to the system, so the control inputs are numbered from 0 to 255. To evaluate the accuracy of our method in obtaining the functionality obtainable from a chaotic map, I have applied the computed control inputs to the logistic map. They construct the same functions that were predicted based on the periodic orbit approximation. The third element of each 4-tuple is the estimated SNR using the UPO approximation. To examine the precision of these SNRs, I experimentally compute the SNR for all functions, as reported as the forth element in each 4-tuple. To compute these experimental SNRs, I change the noise intensity and statistically compute the probability that the desired function is constructed. I choose the noise intensity that results in 99.9% success in constructing the desired function and use it to compute the SNR. I observe that the estimated robustness of the functions against noise in terms of SNR is very near to the experimental results.

Here I explain how one can design a chaos computing system from a given chaotic time series without having access to the underlying dynamical equations. Two key things that enable us to do this are: 1- chaos computing is directly connected to the dynamics of chaotic systems in terms of the short-period UPOs of the system; 2- These short-period UPOs and their robustness against noise are easily extractable from the time series. After extracting these UPOs and modeling the chaotic system with a combination of these UPOs, one can extract the instruction set of the underlying chaotic system and the robustness of these instructions against noise.

Table IV. Instruction set of the logistic map for different iteration numbers obtained from a time series.

P	Instruction Set
1	{(6,128,39.22dB,40.39dB), (7,255,24.30dB,25.8dB), (14,1,24.20dB,26.1dB)}
2	{(5,255,35.64dB,34.4dB), (9,128,30.73dB,31.6dB), (10,1,30.35dB,34.2dB), (11,48,46.01dB,46.09dB), (13,207,46.05dB,45.99dB)}
3	{(2,91,51.14dB,53.29dB), (3,54,37.85dB,38.79dB), (4,164,52.10dB,52.69dB), (5,255,48.17dB,46.79dB), (6,128,38.84dB,40.69dB), (10,1,36.71dB,46.79dB), (11,18,46.5107dB,47.39dB), (12,203,38.55dB,39.19dB), (13,23,45.9dB,48.69dB)}
4	{(1,228,50.23dB,50.49dB), (3,199,56.76dB,56.99dB), (5,88,40.70dB,42.69dB), (6,127,44.91dB,46.19dB), (7,112,54.68dB,54.49dB), (8,27,49.88dB,51.99dB), (9,19,55.87dB,55.99dB), (10,168,41.18dB,42.89dB), (11,7,50.02dB,52.59dB), (12,57,56.66dB,56.89dB), (13,25,47.74dB,54.29dB), (14,14,54.6dB,54.79dB)}
5	{(1,176,56.85dB,58.29dB), (2,213,51.04dB,52.69dB), (3,109,48.08dB,47.99dB), (4,160,51.36dB,52.79dB), (5,170,62.62dB,62.49dB), (6,34,58.72dB,59.89dB), (7,29,66.32dB,67.19dB), (8,80,56.47dB,58.19dB), (10,86,60.70dB,62.79dB), (11,195,43.62dB,48.79dB), (12,147,47.86dB,47.89dB), (13,60,43.2dB,48.29dB), (14,227,65.79dB,67.59dB), (15,128,56.99dB,58.19dB),}
6	{(0,110,60.66dB,63.09dB), (1,106,58.20dB,59.19dB), (2,232,61.11dB,62.59dB), (3,140,58.56dB,69.89dB), (4,66,59.67dB,72.29dB), (5,68,55.76dB,62.39dB), (6,35,66.87dB,69.59dB), (7,173,55.50dB,57.19dB), (8,56,58.70dB,63.69dB), (9,52,56.37dB,57.89dB), (10,187,59.1dB,59.69dB), (11,46,55.98dB,60.39dB), (12,118,60.55dB,65.69dB), (13,210,59.97dB,60.29dB), (14,123,55.09dB,64.9dB), (15,126,54.21dB,62.49dB)}

CHAPTER 9

COMPUTER ARCHITECTURE FOR CHAOS COMPUTING

9.1. Need for a New Architecture for Chaos-based Computers

The Chaogate (chaos-based logic blocks) presented in chapter 3 can be used to construct all the basic logic gates, e.g., two-input AND gates, OR gates, etc. But in the real world much more sophisticated forms of computation are desired and demanded. The science called *Computer Architecture and Design* collects such basic blocks and connects and organizes them to obtain a computing machine that is capable of executing programs and performing sophisticated computations. Since chaos computing contributes many novel features and methods for computation (e.g., dynamic reconfiguration) conventional computer architectures and designs are not applicable to a chaos-based computer, although some of their methods and approaches can be modified and adapted to use in designing chaos-based computers. Therefore a new generation of computer architecture and design is required for chaos-based computers in order to manipulate the novel morphing capability of single chaotic logic gates and to transform them into a morphable chaos-based computer.

In this chapter I develop methods and techniques for designing chaos-based system out of Chaogate, then, I design a chaos-based computing system that can morph to build different instruction sets and even processors. Finally I develop a simulator to simulate the designed chaos-based computing system.

9.2. From One Chaogate to a Lattice of Chaogates

9.2.1. One Chaogate

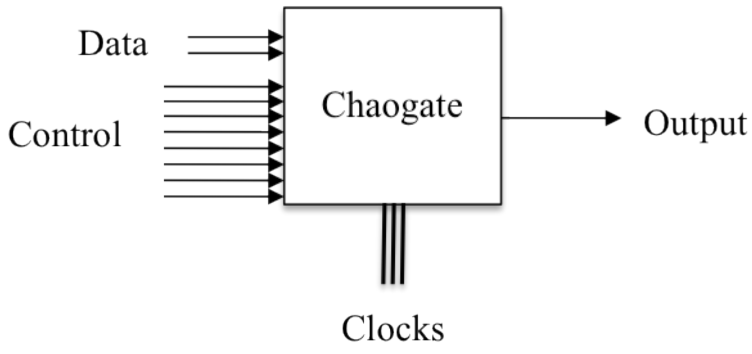


Fig. 9.1 Schematic view of a Chaogate is illustrated. There are three types of inputs, the data, the instruction, and the clocks, and one output, representing the output of computation.

A diagram for a single abstract Chaogate is depicted in Fig. 9.1. By *abstract* I refer to the fact that at this stage of design I don't take into account the details inside the block, instead I consider the Chaogate as a black box, and the only things that matter is the instruction set of the single Chaogate (the correspondence between control inputs and the type of function the Chaogate builds) and the *timing* of the Chaogate. *The timing* describes (1) the time intervals, when I need to feed the inputs to the Chaogate, (2) the evolution time of the Chaogate, and (3) the time interval, when the outputs are ready to be read at the output of the Chaogate. This *abstraction* is an important concept in computer architecture to keep different stages of design separate from each other and to hide the unnecessary details of each stage. At each stage of design, the abstract model contains just the necessary information that is needed and required at that stage. In chapters 4 and 5 I have studied the functionality and robustness of the Chaogate in detail and I have derived the instruction set and the timing of the Chaogate. These are the only information that we need to know about a Chaogate to be able to arrange a series of them in a lattice, organize them, and build a computer out of them.

There are three types of inputs to the Chaogate: Data, control, and Clocks. There is one output line that carries the output of the computation. As described in chapter 3, the Chaogate needs three clocks because of the internal three-stage computing procedure of Chaogate, the initial condition setting, the evolution, and the output production stage. Each clock trigs each stage of the computing. The frequencies and the duty cycles of these clocks depend on the type of technology used for implementing Chaogate the and the implementation details of the Chaogate. This information should be handed from Chaogate circuit design to the chaos-computer architecture stage. Here this mentioned information is available and I continue based on it.

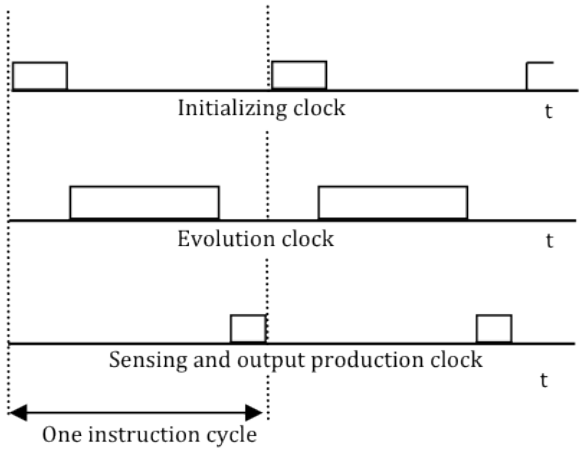


Fig. 9.2. Three clocks required for operation of a Chaogate

These three clocks are illustrated in Fig. 9.2. The Data and Instruction inputs should be ready and kept fixed when the first clock, Initializing clock, is on (the binary symbol of clock is 1). The second clock initiates the chaotic evolution of the Chaogate, and the third clock trigs the output production mechanism of the Chaogate.

User feeds the Chaogate with the data input and appropriate 8-bit control inputs to instruct the Chaogate to do the desired operation on the data. Notice the need for 8-bit control input comes from the random-process analysis of the Chaogate and the fact that to have a universal 2-bit computing engine, one needs to have 8-control bits [37]. However, one may reduce the number of control bits to less than 8 to obtain a reduced set of functions. But in either case, usually the required number of control bits to instruct a Chaogate exceeds the number of required bits to address and count the available functions inside a Chaogate. As an example, in introduced Chaogate in chapter 3, 8 control bits were used to instruct the Chaogate to build 16 different digital functions. But we know that 4 bits are enough to address 16 different digital functions. As discussed in chapter 3, this need to extra number of bits returns back to the fact that different control inputs may result in the same digital function and that is why the number of different control inputs exceeds the number of available functions in a Chaogate. The bottom-line is, here, these 8 control bits are carrying less than 8 bits of information, and this excessive use of wires and inputs is not desirable in computer architecture. For example, the Chaogate will need 8 control pins for programming, or when the user is writing a program to run it on a

Chaogate, he will need to write the long 8-bit operation code to instruct the Chaogate.

The solution I present is using a simple micro-programmed control unit to hold the 8-bit control bits. This control unit is composed of an addressable memory, called control memory. This control memory can be considered as a 2-D array, where rows are the 8-bit controls for programming the Chaogate. The number of rows equals to the number of available functions in the Chaogate or the number of functions a user needs to have. For example, here, the Chaogate is able to build all 16 2-bit digital functions; therefore, the number of rows will be 16. Furthermore, I store the control bits for function number 0 in row number 0 of the array, control bits of function number 1 in row number 1, and the same for all other functions up to function number 15, which is stored in row number 15. Notice the function naming is the same as the one introduced in Table II of chapter 6. The 4-bit address lines of this memory element is used by the user to address one row of the memory, and the 8-bit output of the memory is connected to the Chaogate control inputs. Now the user can program the Chaogate using just 4-bits of information. The schematic of the Chaogate and the micro-programmed control unit is depicted in Fig. 9.3. Now for programming the Chaogate just 4 bits, named operation code, is required.

This architecture enables us to implement all combinational digital functions that accept two bits of information and produces one bit output. These functions are like OR, NANAD, NOR functions, working on single bit operands.

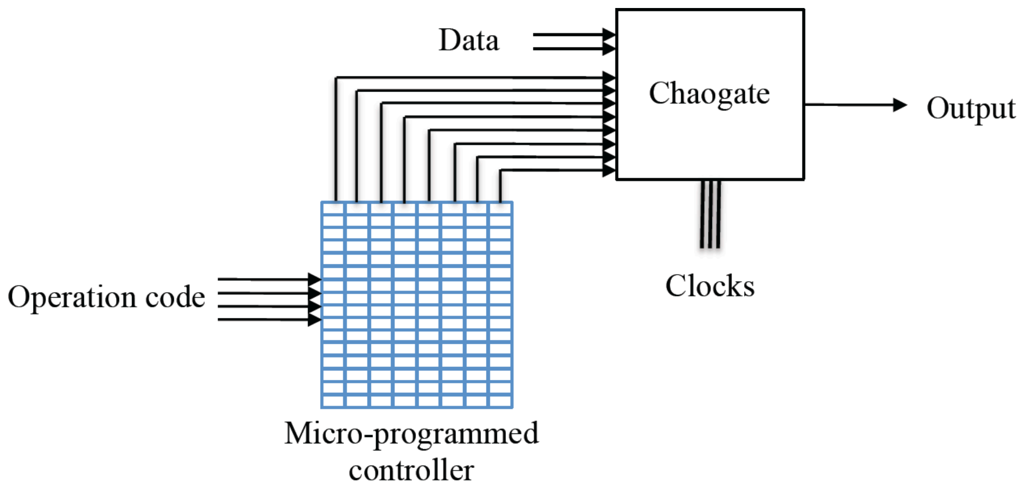


Fig. 9.3. Chaogate controlled by a micro-programmed controller. This architecture reduces the size of operation code (or equivalently required pin number for programming) from 8 to 4.

9.2.2. A Series of Chaogates Arranged in One Column

So far the operand size was one bit. The Chaogate accepts two one-bit operands and carries out computation like AND, NOR, or XOR on them. To build a computing system capable of performing bit-wise operations like AND or XOR on longer operands a column of Chaogates is needed. I propose the architecture illustrated in Fig. 9.4 for this purpose. Assume the goal is to perform simple bit-wise operations on 4-bit operands. Similar to single bit operands, bit-wise operations on operands longer than one bit is performed on pair of corresponding bits. Therefore for 4-bit operands, 4 Chaogates is needed, where the first Chaogate performs the operation on the first pair of corresponding bits, second Chaogate performs the operation on the second pair of bits, and so on and so forth.

The next question is about programming this column of Chaogates to perform desired operation. I propose an extended version of micro-programmed controller depicted and explained in Fig. 9.3 to program this column of Chaogates. Now each row of this extended micro-programmed controller contains the control bits for all Chaogates. For example, here, since there are four Chaogates in the column and each Chaogate needs 8 control bits for programming; each row of the controller contains 32 bits of control bits. The first eight bits control the first Chaogate, the next 8 bits the second Chaogate and so on and so forth. Similar to single-Chaogate architecture, an operation code addresses and selects one specific row to program the column of the Chaogates. The number of rows of the micro-programmed controller depends on the number of functions (operations) in the instruction set of the single-column lattice of Chaogates. For example, to have 8 different functions, there should be 8 different rows of control bits in the controller and the operation code needs to be 3-bit, to be able to address each row of the controller. The proposed architecture is illustrated in Fig. 9.4. Notice that to reduce the complexity of the picture and to prevent unnecessary complications, the 8-bit wires from micro-programmed controller to each Chaogate is replaced with a thick line.

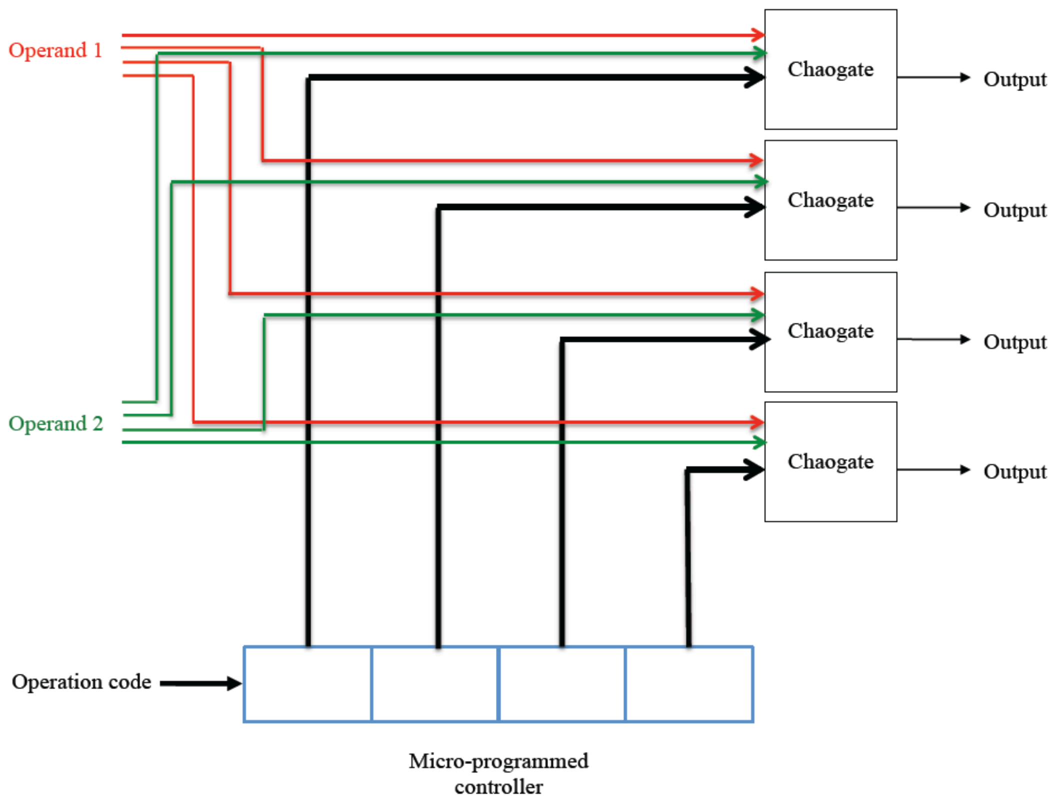


Fig. 9.4. Proposed architecture for single-column lattice of Chaogates. This computing system can perform bit-wise operations like AND, XOR, or NOR on 4-bit operands. To implement such functions, micro-programmed controller needs to contain the corresponding control bits and the user needs to use appropriate operation code to address the rows containing control bits for desired function.

9.2.3. A Lattice of Chaogates Arranged in Rows and Columns

The single-column architecture introduced in Fig. 9.4 can perform any function that is realizable in one layer of logic gates. AND, OR, XOR operating on multi-bit operands are examples of such functions. However, for implementing other functions, like addition or subtraction we need higher number of layers of Chaogates. To expand the architecture from single layer (single column) to multi layer (multi column), two main things need to be addressed: (1) Flow of data from one layer to the next layers (connectivity) (2) controlling and programming such a multi-layer architecture.

To answer these questions I propose a pipelined architecture, in which data and control inputs flow from one side of the lattice to the other side of the lattice, layer by layer, and the rate of flow is one layer at each instruction cycle. Notice that one instruction cycle is the summation of initial condition setting clock cycle, evolution clock cycle, and the output production clock cycle, as is shown in Fig. 9.2. This architecture has important advantages like locality of connections, ease of control, and parallelism of computations through deep pipelining. These advantages will be explained in detail.

In our proposed pipelined architecture data inputs of each layer are selected outputs of Chaogates in the previous layer of architecture. The exception here is the first layer of lattice, in which the data inputs are the data operands. Fig. 9.5 shows how an input to a Chaogate is selected from outputs of Chaogates in previous layer using multiplexers. Here to select an input to the Chaogate from 4 outputs of the previous Chaogates, a 4 to 1 multiplexer is used. The selection is controlled using two select bits. Here I assume there are 4 Chaogates in the previous layer, however it can be any arbitrary number, e.g. *z*. In this case we need a multiplexer of size *z* to 1, and the number of select bits should be the smallest integer not less than $log_2(z)$. Notice that for each data input to a Chaogate one multiplexer is needed. I am assuming the Chaogate is 2-input, as a result two multiplexers for each Chaogate is needed. In Fig 9.5 To simplify the picture in Fig. 9.5 and for better clarification, just one Chaogate in layer number two is depicted and the other Chaogates of the layer are omitted. The other Chaogates will have the same multiplexing system to select out outputs of previous layer as their inputs.

One of the main advantages of this pipelined architecture is the locality of the connections. The inputs to each layer are the outputs of previous layer and the outputs of each layer are connected just to the next layer. Such locality of connections reduces the complexity of the design and removes the need for long wire running across the IC chip. Long wires are not desirable in VLSI because they reduce the system speed, and introduce inductive effects.

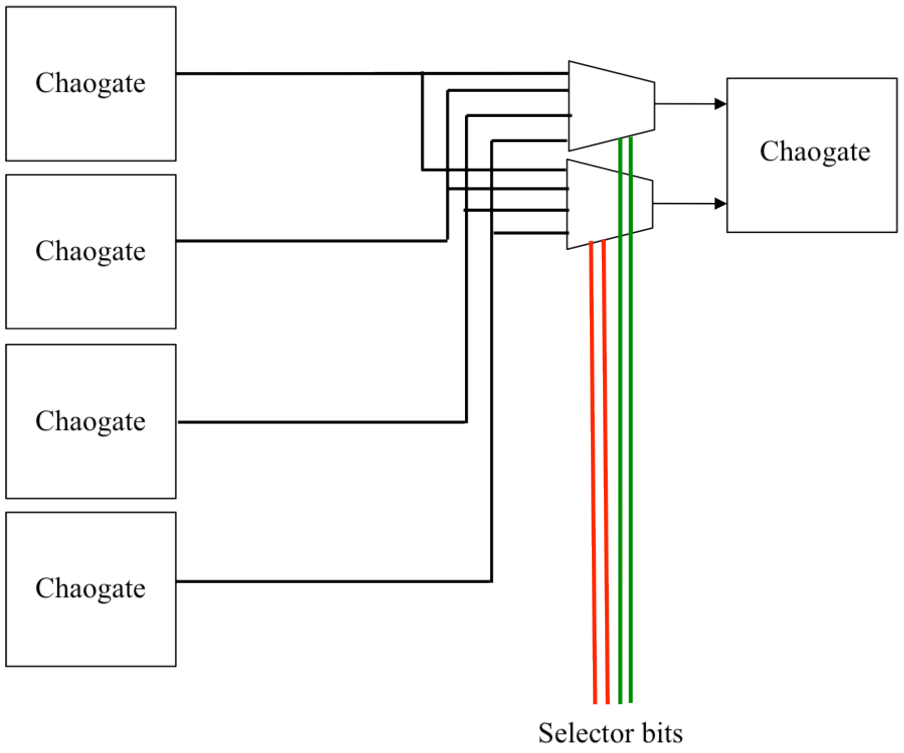


Fig. 9.5. Multiplexers are used to select inputs to a Chaogate from outputs of previous layer.

Notice that in conventional pipelined design we need a register between any two layers of gates to insure that the output of fist layers are stored and preserved until the next layer reads them. In our proposed architecture, there is no need for these intermediate registers for data because the output of the Chaogate has been internally stored inside the Chaogate itself. As an example, in our Chua circuit based implementation [61], there is a sample and hold capacitor in output production circuit. This circuit holds the final state of the chaotic circuit and the output will remain fixed and stable until the next cycle. This embedded register inside a Chaogate is enough to keep the output of a Chaogate stable so that the next layer can use it in initial condition setting phase of next cycle.

In our proposed architecture, the instructions (operation codes) themselves flow in the pipeline along with the data. This pipelined architecture for flow of instruction is illustrated in Fig. 9.6. Here I introduce a distributed micro-programmed controller. Each layer of Chaogates has its own micro-programmed controller. Notice that in the second stage the micro-programmed controller is connected to the selector lines of the input multiplexers as well as the Chaogates. Each row of controller contains the control bits for programming the Chaogates of that stage plus selector bits for multiplexors for selecting the desired inputs to the Chaogates. For simplifying the picture, the data lines are omitted in Fig. 9.6. Also the 8 wires between the controller and the Chaogate is reduced to one single line, and also the selector wires between controller and the multiplexer is reduced to one single line too. Notice that to pipeline the instruction along the lattice a register is placed in between each micro-programmed controller. This register allows the flow of the instruction stage by stage along with the flow of the data. It takes one instruction cycle for data to shift from one column of Chaogates to the right column. The instruction needs to be shifted to the right at the same rate.

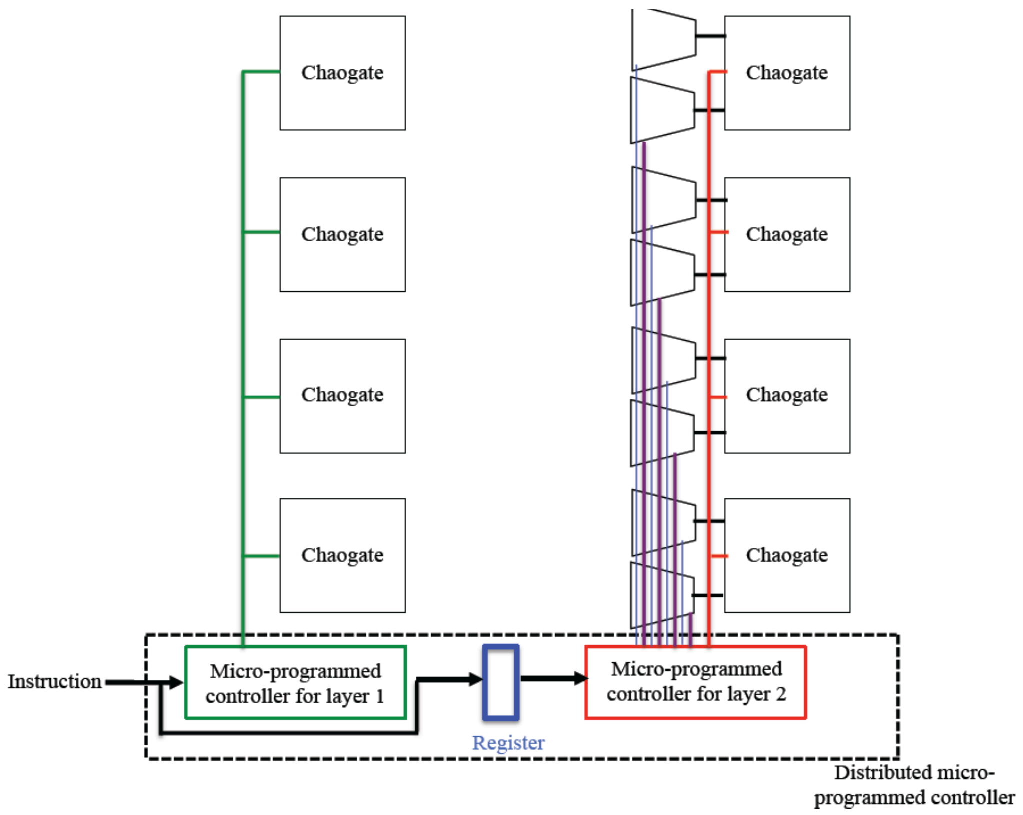

Chaogate
Chaogate
Chaogate
Chaogate
Chaogate
Chaogate
Chaogate
Chaogate
Instruction
Micro-programmed controller for layer 1
Register
Micro-programmed controller for layer 2
Distributed micro-programmed controller

Fig. 9.6. A distributed micro-programmed controller programs the lattice of Chaogates. There is one micro-programmed controller for each column of Chagates . Also a register is placed between any micro-programmed controlled and its consequent micro-programmed controlled at the next column of Chaogates for flow of the instructions. The instruction and the data flow along the lattice at the same time.

One may come up with an architecture that reconfigures the whole lattice of Chaogates to implement one single instruction at a time. Our proposed pipelined architecture is more efficient than these architectures because of its parallelism and higher throughput of instructions. When the first instruction (operation code) and the data is fed to the first layer, the instruction addresses one row of the controller of first column, and the content of that row reconfigures the Chaogates of the first column to build the first layer of the logic circuit for implementing the instruction. At the second cycle of instruction, the second set of data and instruction is fed to the first layer, and as a result the first layer of Chaogates is reconfigured to be the first layer of circuit implementation for the second instruction. Meanwhile, the first instruction is shifted to the second stage of Chaogates, reconfiguring them to be the second layer of the logic circuit implementation of the first instruction. Also the controller selects the appropriate outputs of the first stage to be used as the inputs to the second stage of Chaogates during initializing stage of second instruction cycle. Notice these outputs are stored inside the Chaogate using sample and hold circuits and buffers described earlier. The content of these sample and hold circuits will remain fixed until the output production phase of the second instruction cycle. This process happens all along the lattice. As a result, a Chaogate lattice of size $m \times n$, will hold and implement n different instructions at the same time, one instruction at a column of the lattice. Such pipelining enables us to reach the execution rate of one instruction at each instruction cycle. If I had allocated the whole lattice for implementing one instruction at a time, the rate of execution would be one instruction at *n* instruction cycle, where *n* is the number of columns (layers) of the lattice.

A very important feature of proposed architecture is that the distributed micro-programmed architecture can be loaded with desired bits, since it's nothing more than a SRAM element. This enables us to change the instruction set of the computer.

In a conventional computer controlled by a micro-programmed controller, *read only memory* (ROM) is used to implement the controller. The reason is the computing system has a fixed instruction set, which is already loaded to the controller. But in our proposed architecture for chaos computing, the lattice of Chaogates can implement any digital circuit that fits (in the sense of size) in it. The number of these possible circuits is so high that I am not able to load the instruction set for all of them in the controller. Even if I could, the size of micro-programmed controller and the size of operation code necessary for addressing such a huge micro-programmed controller would be so large that it makes the system inefficient. To have some idea about the approximate number of possible logic circuits that our proposed architecture can build, let's assume the size of lattice is $m{\times}n$, and each Chaogate accepts two inputs and produces one output. Each One Chaogate is able to build 16 different two-input, one-output functions. Also a Chaogate that is placed in second or latter columns of the lattice can be wired and connected to Chaogates of previous column in m^2 different ways. As a result, the lattice is able to build $(16)^{(mn)}(m^2)^{(m(n-1))}$ different logic circuits. Notice that a group of these logic circuits may implement the same function, so the total number of different implemented functions is less than this number, however, this approximation still suggests that the computational capacity of our proposed architecture is really high. As an example, if I assume *m=n=4,* the lattice of this size, can build 5.1923e+33 different logic circuits! Definitely I cannot micro-program all of these logic circuits and instructions in a controller. Instead what I do is for each class of applications I select a manageable subset of these circuits and functions, named instruction set, whose are suitable for that specific application, and load it to the micro-programmed controller. In other words I can have different instructions sets, each tailored and suitable for different applications and needs and all of them loadable and implementable on the same hardware. For example, I can have one instruction set suitable for intense floating point computations, the other for digital signal processing, and another for a graphic processing, etc.

Being able to load and change the instruction set of the processor has very profound advantages over conventional computers. Instruction set architecture of a computer is an important step in computer design, which involves deciding a set of instructions that optimizes the performance of the computer in processing. This optimization is usually measured and defined against benchmark programs. As a result, for any class of programs, a different instruction set suits well. In conventional computers, instruction set of a computer is fixed, the reason is the hardware and the implementation of instructions are hardwired and fixed. But this hardware is used to run different classes of applications. Some users use their computer for graphics applications, another user for signal processing, and someone else for statistical calculations. Our proposed architecture can address and solve the problem. Our chaos-based computer can come with a library of different instruction sets. Each user, based on his needs, can load suitable instruction set to its chaos-based computer to get the maximum performance out of the hardware.

Furthermore, a user can design its own custom instruction set and load it to the chaos-based processor to exactly implement its own application. This idea of loading different instruction sets is really similar to the idea of software and software engineering. Software comes as a package; the user installs the software on the computer and runs it. Similarly, here the user installs (loads) the instruction set on the computer and runs the custom made computer. Furthermore, our approach may open the doors for creation of hardware level software. For example, software can be implemented in hardware level through loading appropriate bits to the micro-programmed controller. The result will be a programmed hardware that dedicatedly runs the software in hardware layer and as a result the application will be faster and reliable than the case it runs as a software running on a generic processor.

Furthermore, our proposed architecture can implement and emulate different types of processors, DSPs, and microcontrollers and work like them. For example, one can load the instruction set of an specific microcontroller to the chaos based computer, and the chaos based compute will morph to be the exact microcontroller and the user can run the programs developed for the microcontroller.

Notice that here I have introduced two different types of programming. (1) Loading an specific instruction set to the micro-programmed controller, (2) running an instruction from the loaded instruction set. After loading an instruction set to micro-programmed controller, the computer starts to read the program and run it. The program is composed of a series of instructions that are already loaded to controller. Any encountered instruction in the program, instructs the controller to *dynamically* program the lattice of Chaogates to be an exact implementation of the instruction. This is the instruction-level programming. The other type of programming is to load different instruction sets to the programmer. To load an instruction set, the computer needs halt processing for loading a new instruction set to the micro-programmed controller, and afterward it restarts processing based on the new loaded instruction set.

Notice that in this thesis I am not deriving and presenting a sophisticated library of different instruction sets, instead I introduce new computer architecture for Chaogates that has flexible instruction set and different instruction sets can be loaded to it. Deriving a library of different instruction sets for the introduced hardware is a huge project by itself, however hardware description languages like VHDL or Verilog may ease the process and can automatically generate different logic circuit implementations for each instruction of desired.

9.3. Hardware Simulation

To test the proposed architecture a software for simulating the hardware is developed to demonstrate the performance of the computing system. This software simulates the proposed architecture wire by wire in details and it models the flow of signals.

This hardware simulator is a critical step in chaos computing, first it demonstrates how and how well the single chaos based logic blocks can be combined to build a processor, and second it bridges the software simulations to the physical hardware fabrications.

In this project C++ language is used to develop the simulator. Object oriented features of the C++ enables us to represent the architecture in terms of its basic blocks like wires, chaos based logic blocks, pins, registers, etc. A C++ class is defined for any type of component used in the architecture. For any instance of the defined component type (class), which is used in the architecture, an object will be declared. There are two main inputs to the software: (1) the instruction set, which is a binary stream and is loaded to the distributed micro-programmed controller. (2) The program, which is a sequence of instructions to be executed. These instructions belong to the instruction set that is already loaded to the micro-programmed controller.

The initial idea was to design and simulate a simple 4-bit processor. During designing the architecture, I exceeded the initial specification (designing a simple 4-bit processor), and instead I proposed a computing system that can execute any instruction set, or any processor, that fits in the lattice. The main criteria for fitting is if the digital circuit implementations of instructions in an instruction set fit in the lattice of Chaogates, the system can execute that instruction set, or simply *if it fits, it runs!*

I develop a hardware simulator for simulating a $m \times n$ lattice of Chaogates controlled by a distributed micro-programmed controller. *m* and *n* are arbitrary and are set as the parameters of the simulator. In simulation, I load an instruction set to this chaos-based computing system. Then I feed the program to the chaos-based computer for execution. This program is consists of the instructions from the loaded instruction set. The instructions reconfigure the lattice column by column while they flow through the lattice.

9.4. Proposed Chaos-based Computing System Versus FPGA

The idea of reconfigurable computing is not new. The first reconfigurable computing device was a PROM, which dates back to 1970 [77]. PROMs were originally intended to use as computer memory, however engineers started to use them for building simple digital functions. PROMs were composed of a lattice of AND gates and OR gates, with fuses in between. By burning different fuses one can implement a digital circuit. *Programmable logic arrays* (PLAs) were the first reconfigurable devices that were originally invented and commercialized for reconfigurable hardware [77].

Field programmable gate array (FPGA) is the state of the art technology in conventional reconfigurable computing [77]. It's composed of a lattice of logic blocks with programmable connection system running between blocks. These logic blocks and the connections are programmable and the FPGA can implement any logic design that fits inside. The programming is mainly performed using SRAM control bits for connections and look up tables for building logic blocks. These look up tables are again a type of SRAM memory. To program an FPGA it needs to be connected to a computer for loading the configuration bits to the FPGA. By loading these configuration bits the FPGA is programmed and is ready to use. However, to reprogram the FPGA again, if the technology allows, the FPGA needs to be stopped from execution of program, and then again it should be connected to a computer for loading the new configuration bits. In other words, the FPGA hardware and the implemented functions inside are frozen after loading the programming bits.

The main advantage of our proposed reconfigurable computing system over conventional reconfigurable systems is that our system can reconfigure and reprogram itself at any instruction cycle. In other words, any encountered instruction reprograms the hardware to be the exact implementation of that instruction. The hardware, the lattice of Chaogates, is continually reconfigured to be the exact optimal implementation for the encountered instruction.

FPGAs are usually efficient only if a fixed computation needs to be carried out on a long stream of data [77]. In such cases the FPGA is reconfigured to be an implementation of this fixed computation. This computation is programmed and frozen in the FPGA and there will be no way to change or reconfigure it afterward. But in our proposed computing system, we can observe speed up in the computation, no matter the computation is fixed or continuously varying. Since the hardware is reconfigurable by encountering any new instruction, the hardware can dynamically change itself to build and implement the new type of computation.

CHAPTER 10

CHAOS EXCITED LINEAR PREDICTOR CODING FOR SPEECH CODING AND PRODUCTION

Chaos is a random-seeming behavior generated by deterministic systems [78]. In addition to numerous physical systems, chaotic activity has been reported in many physiological systems [79,80,81] and pathological systems [82,83]. Also different engineering applications have been introduced for chaotic systems, e.g., chaos-based computation [70,84] and chaos-based communication [85,86].

Chaos has also been widely observed in nature [78]. It is thought [78] that such chaos enables a natural system to have a wider and more flexible range of behaviors than might be the case for a linear system. A case in point is the avian vocal system. In reference [87] it is demonstrated that, in addition to central neural control, the intrinsic nonlinearly oscillatory dynamics of the avian vocal organ expands the range and complexity of possible sounds. The syrinx of a bird can produce a sequence of oscillatory states that are spectrally and temporally complex in response to the slow variation of respiratory or syringeal parameters. In similar research, the significance of nonlinear phenomena in mammalian vocal production for generating highly complex vocalizations without requiring equivalently complex neural control mechanisms is argued [88]. Also chaotic vibrations are observed in vocal folds [89,90,91,92,93,94,95] and experimental studies of excised larynges [96,97,98] and nonlinear dynamical analysis of human voice [99,100] have demonstrated the existence of chaos in the human voice production system.

The speech waveform is an acoustic sound pressure wave that originates from voluntary movements of anatomical structures, which make up the human speech production system [101]. A basic simplified acoustic block diagram of human speech production is shown in Fig. 10.1.

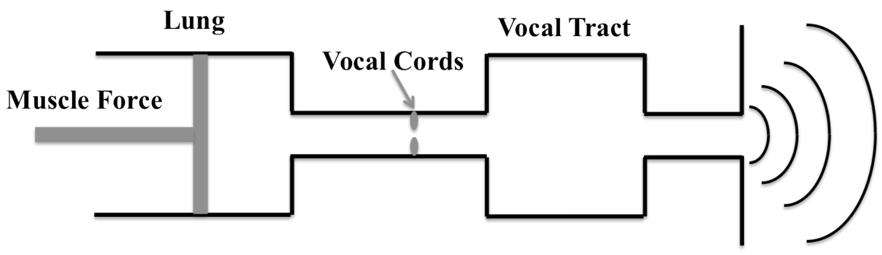


Fig. 10.1. A simple block diagram of the human speech production system.

The entire combination of all these speech production cavities is referred to as the vocal tract and comprises the main acoustic filter. The vocal tract provides resonance to human speech by changing its shape and dimension. The vocal tract filter is excited by the organs below it, the vocal cords, lungs, etc., and is loaded at its main output by a radiation impedance at the lips. The resonant structure of the vocal tract selects different resonant frequencies from the input excitation thereby producing different sounds [101]. There are two elementary types of excitations, voiced and unvoiced. Voiced sounds are produced by forcing air through the glottis or an opening between the vocal folds. The tension of the vocal cords is adjusted so that they vibrate in oscillatory fashion. The periodic interruption of the subglottal airflow results in quasi-periodic puffs of air that excite the vocal tract [101]. Unvoiced excitations, producing unvoiced sounds, are generated by forming a constriction at some point along the vocal tract and forcing air through the constriction to produce *turbulence* [101].

A trivial but not very effective method to transmit voice over a communications network is to record the waveform of the voice and then to transport the waveform. But this method generates too much communication load on the network and requires a very high bandwidth, making it infeasible in many applications. An alternative method is to extract a *model* for generating the waveform and to transport the *model* once to the receiver, letting the receiver synthesize the original waveform from the model and a reduced set of transmitted data. Acoustic models are suitable for understanding the operation of the speech production system; however to code and to synthesize speech, DSP models are required. A common DSP model for the basic acoustic level block diagram of Fig. 10.1 is illustrated in Fig. 10.2.

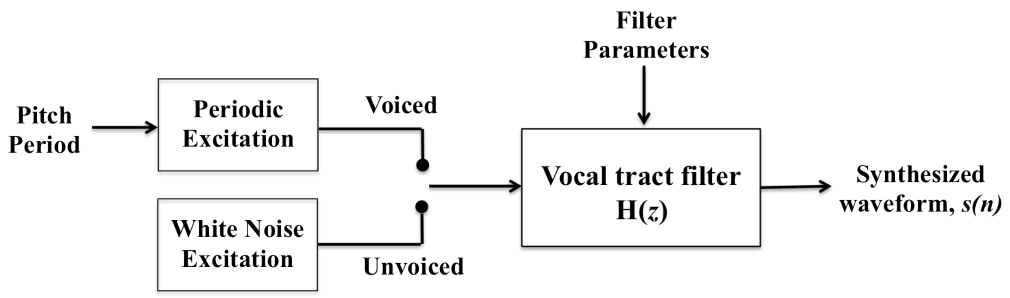


Fig. 10.2. DSP block diagram for the speech production system shown in Fig. 10.1 [101].

In this block diagram, the vocal tract is replaced with a linear filter, *H(z)*, and the acoustic excitations are replaced by a train of periodic excitations for the voiced sounds and a random noise sequence for unvoiced sounds [101,102,103,104]. Linear predictive coding (*LPC*), which is the basis for many speech coding techniques, can be used to extract an estimation for the filter *H(z)*. LPC is a very simple but effective method for coding voice and it assumes that each sample of a voice can be predicted with a linear combination of its past samples. The linear coefficients are the parameters of the filter. These few coefficients can be sent to receiver as an estimation for the filter. The remaining question is how to excite the filter. For unvoiced sounds the excitation is a random-like signal. However for voiced sounds, although they are periodic, there are slight cycle-to-cycle variations in the periodicity of the excitations as well as in their amplitude. Sending the excitation waveform to the receiver is not an option because the excitation waveform takes just as many bits as the original speech waveform, so that this would not provide any compression. Various attempts have been made to encode the excitation waveform in an efficient way. The most successful methods use a codebook, or table of typical excitation waveforms, which is set up by the system designers. In operation, the sender applies all possible excitations to the filter in the unvoiced sound case, or usesthese excitations to perturb the periodic excitation in the voiced sound case, and then compares the synthesized waveform with the original waveform and chooses the excitation waveform that results in the best approximation of the original signal. This principal for determining the optimal excitation signal is called *Analysis by Synthesis* (AbS), signifying that the encoding (analysis) is performed by perceptually optimizing the decoded (synthesis) signal in a closed loop. Since the receiver has the same codebook, the sender sends the index of the optimal excitation waveform to the receiver and the receiver excites the filter by use of this excitation signal to synthesize the speech signal.

This stored codebook introduces a bottleneck in improving the CELP algorithm, since a codebook cannot be arbitrarily large because it will occupy too much memory. It also slows down the CELP algorithm because it requires a memory fetch to read the long excitation sequences from memory [101,102].

10.1 CELP

For our purposes, a *wide-sense stationary* frame of speech, *s(n),* will be taken as one whose mean and autocorrelation do not change over time. It can ideally be characterized by a pole-zero system transfer function of the form [101]:

$$\Theta(z) = \Theta_0 \frac{1 + \sum_{i=1}^{L} b(i) z^{-i}}{a(i) z^{-i}} \tag{21}$$

However, for analytical reasons, the transfer function, $\Theta(z)$, is approximated by an all pole transfer function [101]:

$$\hat{\Theta}(z) = \Theta_0 \frac{1}{1 - \sum_{i=1}^{M} \hat{a}(i) z^{-i}} \tag{22}$$

The estimates $\hat{a}(i)$ are the coefficients of the linear prediction (LP) model and constitute a parametric representation of the filter. In the time domain we have:

$$s(n) = \sum_{i=1}^{I} a(i) s(n-i) + \Theta_0 e^{'}(z) \tag{23}$$

Thus the name *linear prediction* comes from the fact that *s*(*n*) can be predicted using a linear combination of its past values driven by a phase-altered version, *e'(n)*, of the excitation signal, *e(n)* [101]. To obtain optimal values of the parameters $\hat{a}(i)$, the root mean square criterion is used. In this method I minimize the expected value of the squared error [101,102]:

$$\min E\{\hat{e}^2(n)\}, \hat{e}(n) = s(n) - \hat{s}(n) = s(n) - \sum_{i=1}^{M} \hat{a}(i)s(n-i) \quad (24)$$

and it results in the normal equation:

$$\sum_{i=1}^{M} \hat{a}(i) r_s(\eta - i) = r_s(\eta) \quad (25)$$

where r_s is the temporal autocorrelation of *s(n)*. Different techniques have been introduced for solving normal equations; here the well-known Levinson recursion is used to solve the equation and to obtain the coefficients of the filter [101].

After computing the LP filter and reverse filtering the speech waveform, the signal $e^{'}(n)$ is obtained. Notice that this signal is not just the excitation signal; it is more like a residue signal, the difference between the LP coefficient-weighted sum of past values of the signal and the original signal, $\Theta_0 e^{'}(z) = s(n) - \sum_{i=1}^{I} a(i)s(n-i)$. The main part of this residue is the excitation; however the model error is included too.

An estimation of the pitch period can be obtained by computing the autocorrelation of this residue signal. If the peak value of the computed autocorrelation is less than some threshold, I conclude that the speech signal is unvoiced, else the index value of the peak represents the pitch period of the periodic excitation. This estimation of the pitch period will be improved by exhaustive search around this initial estimation.

A schematic of this CELP algorithm is presented in Fig. 10.3. First LP analysis is used to compute the LP synthesis filter. To shape the coding noise, a perceptual weighting filter is used to adjust the error between the synthesized speech and the original speech to emphasize differences of physiological relevance:

$$w(z) = \frac{\hat{A}(z)}{\hat{A}\left(\frac{z}{c}\right)}$$

$$\hat{A}\left(\frac{Z}{C}\right) = 1 - \hat{a}_1\left(\frac{z}{c}\right)^{-1} - \cdots - \hat{a}_M\left(\frac{z}{c}\right)^{-M} \tag{26}$$

and $\hat{a}_i$ are the LP coefficients.

To synthesize the excitations, a pitch synthesis filter, $\Theta_p$is used:

$$\Theta_p = \frac{1}{1-bz^{-P}} \tag{27}$$

and the time-domain output of this filter, which is the excitation to the LP filter, is:

$$e(n) = \Theta_o \rho_k(n) + be(n-p) \tag{28}$$

Here Θ_0 is the gain, *k* is the index of excitation waveform in the codebook, $\rho_k(n)$ is the *k*th sample of the excitation waveform *k* in the codebook, *b* is a parameter controlling how strongly past excitations influence the current value, 0<*b*<1.4, and *p* is the pitch period. These parameters need to be selected so that the energy of the perceptually weighted error between the speech and synthetic speech is minimized. The optimal value of *p* is obtained by exhaustive search around the initial estimation of pitch period and *b* is obtained based on *p*. Θ_0 and *k* are chosen by exhaustive search of the Gaussian codebook to minimize the energy of the error.

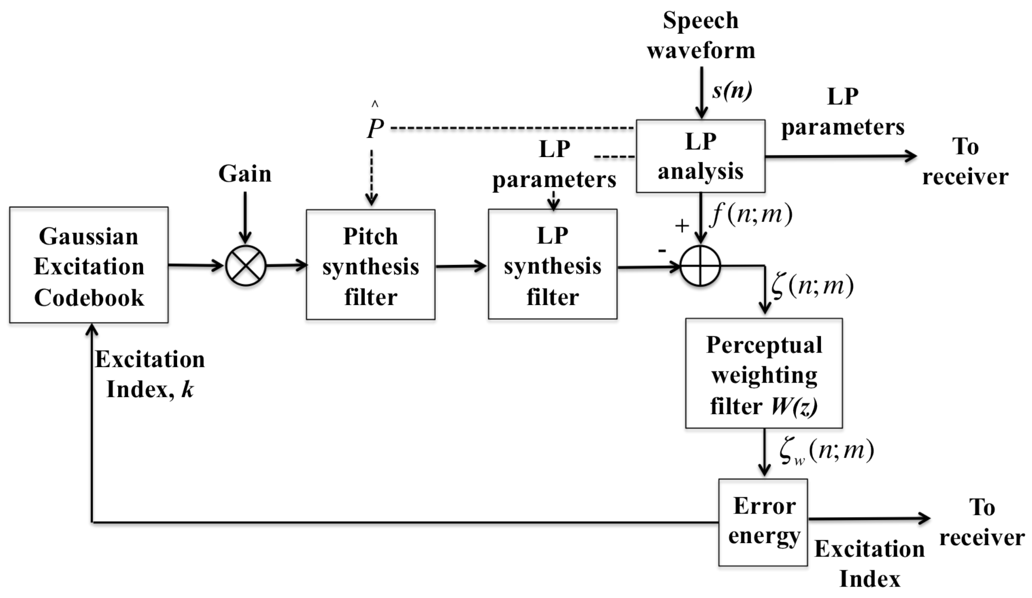


Fig. 10.3. Schematic of the CELP algorithm.

10.2 Chaotic Excited Linear Predictor

The selection of a suitable random excitation is one of the key factors in the performance of the CELP algorithm. In the CELP algorithm described above, a codebook that is a collection of Gaussian random waveforms is used to provide the method with a suitable suite of random sequences to minimize the error signal. The CELP algorithm is commonly used in mobile systems where memory is a restriction. Thus the codebook cannot be arbitrarily large. This limits the number of random waveforms that can be stored in the codebook. The other problem with codebooks relates to the fetching and reading times from a device's memory. Reading from memory is always slow in comparison to processing. Since the CELP method reads all the random waveforms from the codebook and applies them to the filter one by one to find the optimal one, the codebook access bandwidth and latency produce a bottleneck in the speed of the algorithm.

Here I demonstrate a *chaos-excited linear predictor* or *ChELP* algorithm in which I substitute the random-seeming chaotic orbits generated from a simple chaotic map for the random waveforms stored in a codebook [105]. The presence of nonlinearity and chaotic behavior in human speech production system has often been reported and studied [89,90,91,92,93,94,95,96,97,98, 99,100], however chaotic dynamics has not been widely exploited in speech coding and artificial speech production systems. A chaotic map can be as simple as the logistic map, which requires just two multiplications and one subtraction to produce each waveform sample:

$$x_{n+1} = rx_n(1 - x_n) \tag{29}$$

Notice that there is no need to store the random-seeming iterations of the map in a codebook; instead the processor can generate them while it is coding the speech. Additionally the generation of the orbit requires so little computation that, practically speaking, it will be faster than reading the orbit from the memory.

Different random-seeming waveforms can be generated from a chaotic map just by changing the initial conditions. Therefore the initial conditions can be used as indices for the excitation waveform. In the sender, when the ChELP algorithm finds a suitable waveform, it would send the initial condition along the other ChELP parameters to the receiver. The receiver would then use the initial condition to reproduce the same time series. Note that in speech processing the speech is encoded and synthesized frame by frame, and thus the excitations are short waveforms as well. Therefore the chaotic orbit will not be long enough so that noise, in this case the difference in rounding errors from the sending processor to the receiving processor, dramatically changes the orbit.

Since in chaos-excited linear predictor coding the chaotic orbits are not stored and memory is not an issue any more, chaos-excited linear predictor coding can exploit and search a higher number of chaotic orbits to find better random-like excitations, in the sense of less error.

In the human speech production system there is no random number generator; instead the vocal tract bends to produce turbulence, which is a random-seeming excitation. This is very similar to the approach I take in the chaos-excited linear predictor, and therefore I can claim that the approach I introduce here is more biologically oriented than the original CELP.

To test the idea, replace the codebook in Fig. 10.3 with a simple logistic map. The index to identify and to generate the random-like excitations is the initial condition of the logistic map. (Here I have repeatedly used the terms random-seeming or random-like to denote the output of a chaotic map. In reality this output is highly deterministic. Nevertheless it shares many qualities in common with random numbers; so much so that many random number generators are in fact based upon chaotic generating functions. In what follows I will simply use the word random, but always with the caveat explained here.) In the logistic map there is a tuning parameter, r. By setting $r = 4$, I get the maximum amount of randomness from the map. The simulation results are shown in Fig. 10.4. Three phrases, “chaos computing”, “applied chaos lab”, “nonlinear speech processing”, denoted by S1, S2, and S3, and spoken by two different speakers, are the benchmark. I choose the power of the error signal, which is weighted by the perceptual filter introduced in Eq. (26), as the measure for evaluating the performance. White bars represent the power of the error signal when sentences are coded and synthesized by use of a codebook containing 512 Gaussian random waveforms, and grey bars represents the power of error signal when 512 orbits of the Logistic map are used for coding. The power of the error signal is in the same range as that of the Gaussian codebook. As we see, a very simple and primitive chaotic map can perform almost as well as the state of the art codebook of Gaussian random waveforms.

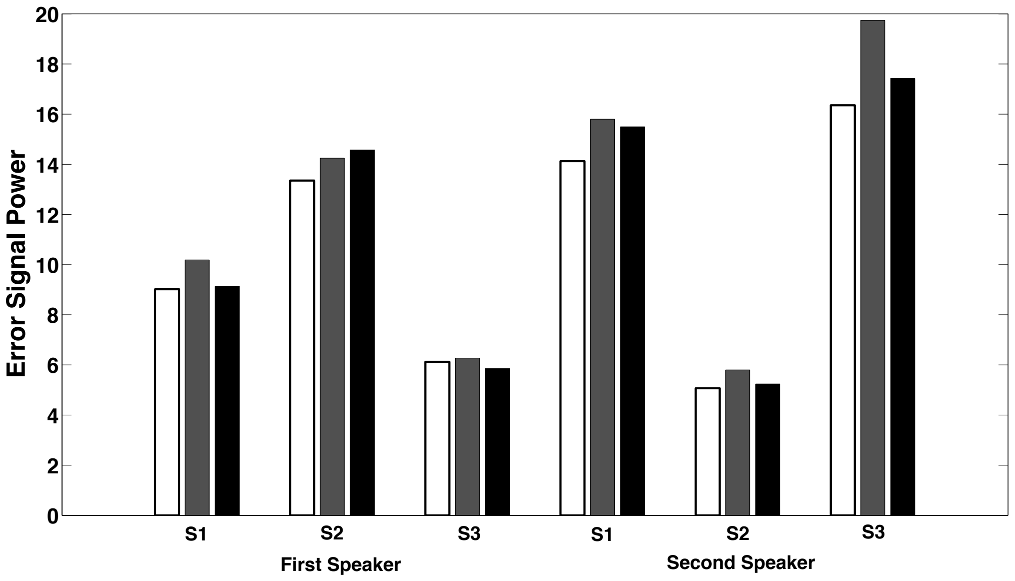


Fig. 10.4. Three sentences, S1, S2, and S3, spoken by two different speakers, are used for evaluating the performance. White, grey, and black bars represent the power of the perceptually-weighted error signal when a Gaussian code book, the Logistic map, and a coupled map lattice are used for excitation, respectively.

To investigate the interrelationship between the amount of chaotic behavior in a map and the performance of ChELP, in Fig. 10.5 I plot the power of perceptually weighted error signal of the ChELP algorithm in synthesizing the speech signal versus the Lyapunov exponent of the map. The Lyapunov exponent quantifies the degree of chaos in the system. To obtain this plot I change the bifurcation parameter of the logistic map, and for each bifurcation value, I compute the Lyapunov exponent and the error of the ChELP in modeling speech. This error is plotted versus the Lyapunov exponent. Notice that at some values of the Lyapunov exponent I have more than one data point. The reason is multiple bifurcation parameters can result in the same Lyapunov exponent and as a result I have more than one error for such values of the Lyapunov exponent. The overall trend is that as the Lyapunov exponent increases, the error of ChELP in synthesizing speech reduces. This result motivates us to investigate more highly chaotic maps; to do this I examine a *coupled map lattice* (CML) to produce the excitations.

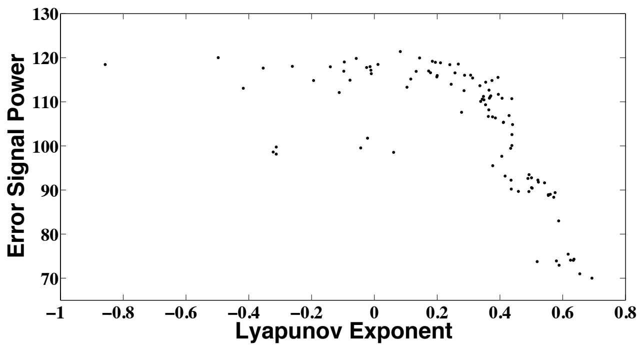


Fig. 10.5. Power of perceptually weighted error signal of ChELP algorithm versus different values of Lyapunov exponent. As the Lyapunov exponent increases, the algorithmic error reduces.

A CML is an array of normal chaotic maps coupled together [106]. Here by increase of lattice size the dimensionality of the chaotic orbits of individual nodes increases [106]. The dynamics of the CML I study in this research is:

$$u_{n+1}(i,j) = (1-\varepsilon) f(u_n(i,j)) + \frac{1}{4}\varepsilon \sum_{nn\, i^{'},j^{'}} f(u_n(i^{'},j^{'})) \tag{30}$$

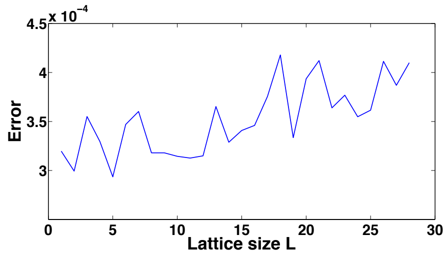
where (i’, j’) are nearest neighbors of (i, j). The function f(x) is the chaotic map, and in this research it is simply the logistic map introduced in Eq. (24). is coupling strength between nodes, here I simply set it to be 0.4. I use torus topology for the CML, therefore boundary conditions will not be an issue here. The orbits of any node can be used as the excitations. However the sender and receiver should use the same node for generating the excitations.

For generating high dimensional excitations I use CMLs of different sizes. A CML of size 8x 8 improves the performance of ChELP and makes it as efficient as Gaussian noise excited linear predictor. The power of error of 8 x 8 CML-excited ChELP is depicted in Fig. 10.4 by black bars. Since each node is a simple iterated map, simulating a lattice of 64 of such iterated maps is not computationally intensive.

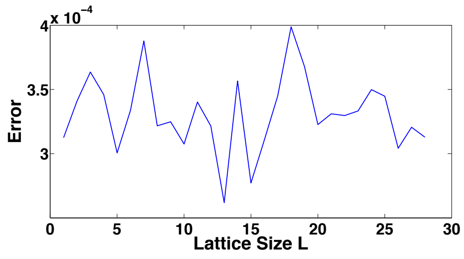
As described earlier, the initial condition of the chaotic system needs to be sent to the receiver as an *index* to produce the correct excitation. The problem with a CML is that, for an $L \times L$ CML-based ChELP, I need $L \times L$ individual initial conditions for all the different nodes in the lattice. This larger number of initial conditions adds an overhead to the bandwidth of the communication system and reduces the compression ratio of the algorithm. The solution I use here is take a single number as a *seed index* and to use a function to generate different initial conditions of the nodes in CML from that seed index. Both sender and receiver share the same function and thus by sending the single seed index the receiver will be able to initialize the CML in the exact way that the sender did and therefore the CML will provide the same excitation for the algorithm. An ideal function for this purpose can be a pseudorandom number generator, where the seed index is fed to the function as the initial seed. The first random numbers produced by the generator can be used as the initial conditions for the nodes of the lattice. Unfortunately good pseudorandom number generators are not fast and require too many CPU clock cycles for practical use and thus will reduce the performance of the CML ChELP in terms of speed. However our studies show that even a very simple and basic generator function is enough to produce different initial conditions for the nodes from a single seed index. The function I use in this research to compute the initial condition of node (*i*, *j*) in a CML of size from given seed index value, *k*: is

$$x_0(i,j,k) = \frac{i \times j \times k}{L \times L \times k_{\max}} \tag{31}$$

where $k=1,2,\ldots,k_{max}$ and $i,\ j=1,2,\ldots,L$. By use of this function I can initialize different nodes to numbers between 0 and 1. This function is very efficient for small CMLs, but when the size increases, correlations between different nodes of the CML appear. Worse, excitations initiated by nearby index values are also highly correlated, which means that subsequent excitations of the lattice will not be independent. As a result the CML fails to provide the algorithm with a diverse set of excitations, and therefore the performance reduces. Fig. 10.6(a) shows the mean squared error versus *L*. Increasing *L* increases the error. Based on our further investigations it turned out that by use of a small sized CML that is initialized by use of function in Eq. (31) I can get almost the same amount of performance when a large sized CML initialized by use of a pseudorandom number generator is used. The mean squared error of CML CELP algorithm versus lattice size *L*, when the lattice is initialized by use of a pseudorandom number generator is illustrated in Fig. 10.6(b). Based on these performances, I conclude that a small CML initialized by use of function in Eq. (31) is the optimal choice for producing the excitation. It is small, therefore it requires fewer CPU clock for simulating the lattice, and by use of a very simple function for initializing it provides the algorithm with highly diverse excitations, which results in a very low error in coding.



a



b

Figure 10.6. (a) The error of CML versus lattice size when the function in Eq. (31) is used for initializing the CML. (b) The error of CML versus lattice size when a pseudorandom number generator is used for initializing the CML.

Here chaotic excited linear predictor method for speech coding and synthesizing was introduced. In this method instead of stored random waveforms a chaotic map was used to generate random-seeming waveforms upon demand. I started from a simple chaotic map and the initial condition of the map is used as an index to produce different waveforms and the same index is sent to the receiver to generate and use the same waveform. To compare the performance, I synthesize a sample speech signal once by use of a codebook filled with Gaussians random numbers, and then I code and synthesize the same waveform by use of the chaotic orbits of the simple chaotic map. We observed that the performance of the new method is almost as good as the conventional CELP. For further improvement of the method I used a CML for producing excitations. The CML is initialized by use of a single index and a simple function, and the index is sent to the receiver to produce the same excitation. The root mean square error of this CML based CELP is the same and even slightly less than that of the original CELP method. The main advantages of chaotic excitation linear predictor are: 1- Since the waveforms are generated online there is no memory limitations and a more number of waveforms can be used to find better excitations resulting in better performance. 2- The waveforms are not read from a memory, and they are generated by a just a few basic machine instructions, therefore the new method can be faster than the previous method. 3- The new method is more biological oriented. 4- CMLs are known to model the turbulence and we already know the turbulence in our speech production system produces the random like excitations. Therefore the current work introduces the possibility of direct estimation of random like excitations in speech waveform by use of a CML.

CHAPTER 11

LOGICAL STOCHSTIC RESONANCE AND SYNTHETIC BILOGY

11.1 Logical Stochastic Resonance

Logiscal Stochastic Resonance (LSR) is a new dynamics-based direction in computation [107,108,109,110,111]. In the field of electrical engineering, with the present tendency to scale down each element in the circuit toward the nanometer region, noise has become an element that cannot be eliminated or neglected. Noise is relevant to both circuit characterization and functionality. For instance, noise immunity in an electrical circuit has become the recurring objective of significant research efforts in this field. Similarly, in biology, when working in nano-scale dimensions and with a small number of elements, small fluctuations may greatlly affect the system behavior. In traditional circuits, noise can cause logic gates to fail and not behave according to truth tables. The common approach is to find a solution that reduces the noise intensity in order to obtain as stable and predictable a performance as possible.

Consequently, it is counterintuitive that noise would enhance the stability and predictability of a circuit. Instead of conceiving noise as a disturbance, a possible approach to noise is to exploit it. In order to undergo LSR, a nonlinearity—such as a bistable potential function—and a noisy signal have to be present.

The typical logic gate-the core of a digital circuit-is a system that converts two inputs into a single output that corresponds to a particular logic function. An AND gate, for example, will output a “one” only if both inputs are “one”; on the other hand, an OR gate will output a “zero” only when both inputs are “zero”. The necessary bistability is related to the two possible output values that the logic gate can have: “zero” or “one”.

To explain the main idea of LSR, let’s imagine our logic gate as a bistable potential function. Because the potential function is bistable, its shape will have two steady states (we can define them as the left and right wells) and an unstable state (obviously between the two steady states). The difference between the potential function value at the unstable state and the potential function value at one of the two steady states is called a barrier. If the potential energy function is symmetric, the calculated barrier with respect to the left state is equal to the calculated barrier with respect to the right state. On the contrary, if the potential function is asymmetric, the two barriers will have different values. Moreover, if the noise intensity is comparable to the barrier value, the system will have the correct amount of energy to randomly overcome the barrier and to change its well. For the logic gate functionality, an asymmetric configuration is preferable. In accordance with the truth table, a deeper barrier characterizes the desired output value.

Now consider a noise intensity that allows the system to switch from the “wrong” stable state to the “correct” stable state (the one with a deeper barrier). The same noise intensity will not be sufficient to let the system to switch back in a reasonable time (because the “correct” well will have a deeper barrier, higher than the noise intensity value). Finally, the measured output state will be the one according to the truth table.

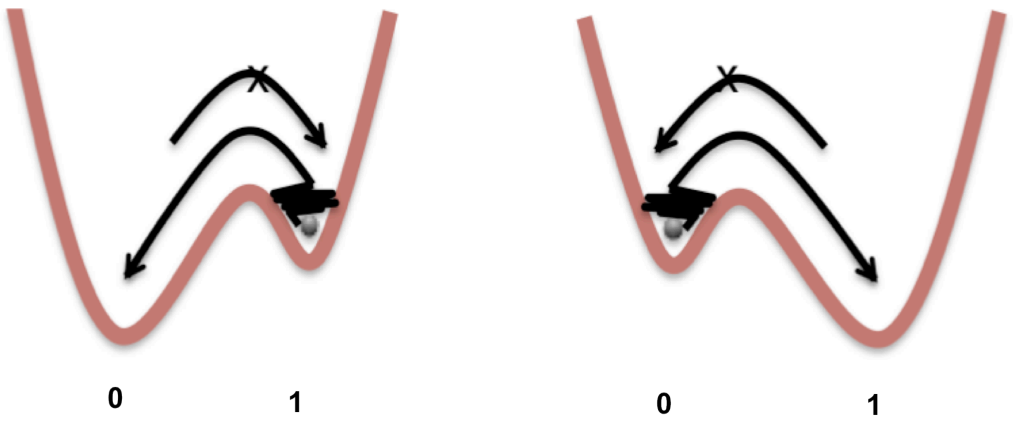


Fig. 11.1: The two asymmetric potential function configurations. In the left panel the “zero”output value is expected, while in the right panel the “one” output values is expected.

What has been explained since now is the behavior generated by the interplay between bistability and noise. LSR exploits this phenomenon in order to create a morphable logic gate, robust to noise. With the Logical Stochastic Resonance we are in the presence of a single system that can switch between two logic gates. To better understand LSR, consider a general SDE:

$$\dot{x} = F(x,a,b,...) + D_n\xi(t) \tag{32}$$

where *x* is the measured output, while *F* is related to the potential function represented in Fig. 11.1, and D_nx(t) is the term that represents the presence of noise. In particular, *F* depends on several parameters (for example *a* and *b* in Eq. (32)). To implement the LSR paradigm, the input signal is encoded by adjusting a parameter of the nonlinear function (for example adjusting *a*). As explained previously, in this work I am encoding the sum of our two logical inputs, this means that I will need only one parameter (as suppose it will be *a*), instead of two parameters as in the traditional case. At the same time, a tuning control parameter is needed to switch between the two logic gate functionalities; in our particular case, to switch between the AND gate to the OR gate (for example, *b* is the parameter).

Changing and tuning the parameters *a* and *b* means to adjust the potential function asymmetry (see Fig. 11.1) in a way that the system is able to exploit noise in order to jump from the “wrong” well to the “correct” well. With performance I define the ratio of the success in realizing the desired gate over the number of attempts. This ratio is the probability of realizing the desired gate.

Eq. (32) is a general form that describes the LSR paradigm. According to the particular implementation of the LSR, the function *F* in Eq. (32) will assume different forms, moreover all the variables and parameters will have different meanings.

Here I want to apply the LSR paradigm to a single gene network, where the noise is always present and the computing paradigm must be robust to the high level of noise.

11.2 Creating Logical Stochastic Resonance in an Engineered Gene Network

The LSR paradigm is adapted to a synthetic gene network derived from the bacteriophage λ [112,113,114,115]. Consider, first, a deterministic model describing the temporal evolution of the concentration of protein in a single-gene network from bacteriophage λ. Bacteria and their temperate phages, like Escherichia coli (E. coli) and λ, exist in symbiotic relationships. After the virus λ infects the bacteria, its evolution proceeds down one of two pathways: lytic (wherein the λ replicates its DNA autonomously, assembles virions and lyses the host) and lysogenic (wherein the phage DNA is incorporated into the host genome) [113,114]. Hence, the bacteriophage λ GRN displays bistability in the choice of one of the two pathways with the characteristics of its stable attractors adjustable by changing the system parameters. Such binary decision-making has also been demonstrated by a gene network with positive feedback loop [115]. Following this, LSR uses the data inputs to adjust the (relative) depths of the two (stable) wells of the potential energy function so that the well representing the desired output (as defined by the truth tables [109]) of the computation becomes deeper than the other well. In the bacteriophage λ GRN the two main (adjustable) parameters to implement LSR are α (related to the basal rate of production of the repressor CI), and γ (proportional to the degradation rate of CI) [115]. Hence, the logic inputs sets ((0,0), (0,1)/(1,0), and (1,1)) are encoded via α, and control inputs representing the type of computation (AND or OR gates), are encoded through γ. This leads us to a reconfigurable GRN-based logic device whose workings are underpinned by the interplay between its (intrinsic) nonlinearity and the noise [115,116]. The output of the computation can be decoded from the final state of the dynamical system; this is 0 or 1 depending on the potential well that the system settles into. Precise definitions of α and γ are provided later in this work. Our system is a DNA plasmid consisting of a promoter region PRM that regulates the cI gene. This promoter consists of three tandem operational sites, O_{R1}, O_{R2} (activated transcription) and O_{R3} (repressed transcription). These genetic elements provide a positive feedback loop. Bistability is reached in the system only when a correct mutual relation between the production and the degradation of protein is realized. Operatively, the logic gate inputs can be adjusted in several ways, e.g. via the bacteriophage response to UV light [117], while the gate reconfiguration can be obtained by varying the degradation term through its response to temperature changes [118,119].

The biochemical reactions that control λ phage are very well characterized [10,16]. They are, naturally, divided into fast and slow categories (table V). Then, defining the concentrations of network components as dynamical variables, x = [X], $x_2 = [X_2]$, $d_0 = [D]$, $d_1 = [D_1]$, $d_2 = [D_2D_1]$, and $d_3 = [D_3D_2D_1]$, it is possible to write the evolution of the concentration repressor CI for the monomer and dimer forms:

$$\begin{aligned} \dot{x} &= -2k_1x^2 + 2k_{-1}x_2 + nk_t p_0(d_1 + \beta d_2) - k_x x + \varepsilon d_0 \\ \dot{x}_2 &= k_1 x^2 - k_{-1}x_2 - k_y x_2 \end{aligned} \tag{33}$$

where the concentration of RNA polymerase, p_0, is assumed to remain constant over time, ε is the basal production rate of the repressor CI, and $K_i = k_i/k_{-i}$ are equilibrium rate constants (i=1, . . . , 4). To accurately model the evolution of the chemical species x, I can sum x and x2 to consider the total number of biomolecules. The system can be reduced by exploiting the fact that the dimerization reactions occur on a time scale that is much faster than the other reactions [115,116].

Table V: Biochemical reactions of the presented network are summarized. X, X2, and D denote the repressor, the repressor dimer, and the DNA promoter site, respectively; Di denotes dimer binding to the ORi site, and in order, each fast reaction is characterized by a rate constant: K_1, K_2, K_3 = σ1K_2, and $K_4 = \sigma_2 K_2$. σ_1 and σ_2 represent the binding strengths relative to the dimer O_{R1} strength. Slow reactions are the transcription, dilution and the degradation (with rates k_t, $k_y = k_x/20$ and k_x): P denotes the concentration of RNA polymerase, n is the number of repressor proteins per mRNA transcript. The dimer occupation of OR2 enhances the transcription rate of a factor β >1 and it appears only in the second slow biochemical reaction.

Fast Reactions	Slow Reactions
$X+X \Leftrightarrow X_2$	$D_1+P \rightarrow D_1+P+nX$
$D+X_2 \Leftrightarrow D_1$	$D_2D_1+P \rightarrow D_2D_1+P+nX$
$D_1+X_2 \Leftrightarrow D_2D_1$	$X \rightarrow f$
$D_2D_1+X_2 \Leftrightarrow D_3D_2D_1$	$X_2 \rightarrow f$

After considerable calculations I obtain, in terms of the dimensionless variables $\tilde{x} = x\sqrt{K_1 K_2}$, $\tilde{t} = \frac{trK_2}{4}$ and $r = \mathcal{E} d_T$ (I have suppressed the overbar on x):

$$\dot{x} = \frac{(\alpha-1)x^2 + \sigma_1(\alpha\beta-1)x^4 - \sigma_1\sigma_2 x^6}{(\tau+x)(1+x^2+\sigma_1 x^4+\sigma_1\sigma_2 x^6)} + \frac{1-\gamma x-\gamma_y x^2}{\tau+x} + D_n\xi(t) \tag{34}$$

where I set $\alpha = \frac{nk_t p_0 d_T}{r}$, $\gamma = \frac{k_x}{(\sqrt{K_1 K_2} r)}$, $\gamma_y = \frac{2k_y}{(rK_2)}$, $\tau = \frac{\sqrt{K_1 K_2}}{4K_1}$ and I have added a noise term representing, phenomenologically, the fluctuations affecting the system. In addition, I set (and retain throughout this work) the degree of transcriptional activation as β = 11, the equilibrium constant for *cI* dimerization as *K_1 =0.05 (nM)$^{-1}$*, the equilibrium constant for *cI* –*O_R* reaction as *K2 =0.33 (nM)−1*, the binding affinity for *cI* dimer to *O_{R2}* relative to *O_{R1}* as *σ_1* =2, and the binding affinity for *cI* dimer to *O_{R3}* relative to *O_{R1}* as *σ_2 =0.08* to maintain the connection to biologically accessible parameter ranges [114,119]. Here, I focus on the (additive) external noise that can stem from random variations in the (external) control parameters. *ξ(t)* is zero-mean Gaussian noise *(<ξ(t)> = 0)*, and I assume that random fluctuations have correlation time scale smaller than any other reaction time scale in the system, so that the noise can be taken to be delta correlated, i.e., <ξ(t)ξ(t')> = δ(t−t'), with *D_n* being the measure of the noise intensity. Equation (33) is the core of the computing model.

The potential function of the (deterministic, i.e. ξ(t) =0) system in Eq. (33), *U(x)* (fig. 11.2), is obtained analytically by integrating the right-hand side of (29) with α and γ the two accessible parameters (taken in the regime of bistability). The plotted curves of U(x) represent the most robust configuration in the limited range of parameters, α and γ, germane to the biological system in the bistable configuration. Several simulations have been made to exhaustively search (in the parameter space) the most robust configuration of parameters that yields the best logic gate performances. For the "conventional" LSR paradigm [109,120], I obtained (numerically) α=7.8, 9.1, and 10.4 (respectively, for (0, 0), (0, 1)/(1, 0), and (1, 1)), and γ =41.9 yields the AND gate and γ =36.5 the OR gate as the control input for programming the gate.

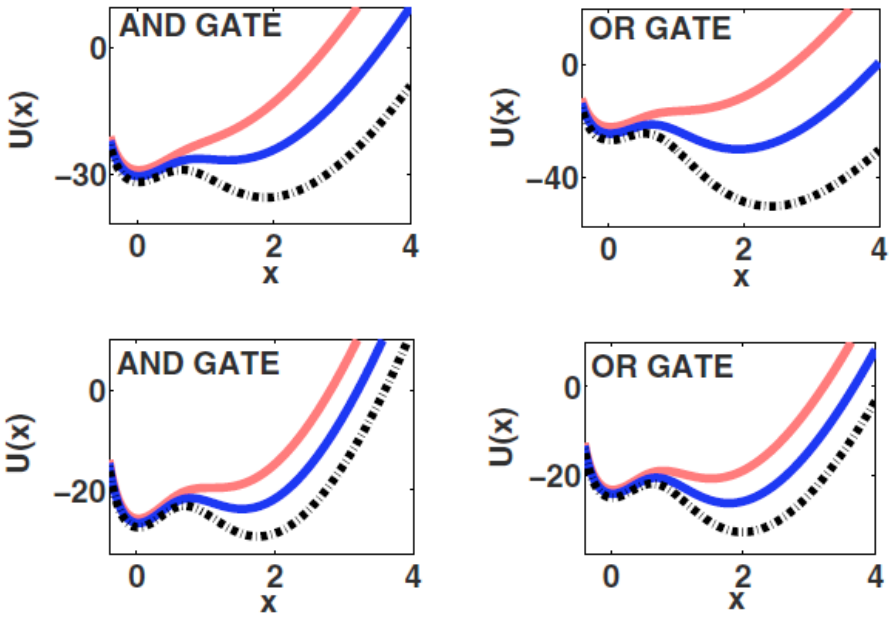


Fig. 11.2. Potential functions for different data inputs for the AND gate (left panels) and the OR gate (right panels), using the modified LSR paradigm (see text) (the two upper panels), and the “conventional” LSR paradigm [4] (the two lower panels). The red curve represents the (0,0) case, the blue curve represents the (0,1)/(1,0) cases, and the black curve is for (1,1) case. Values in the accessible parameter range, related to the most robust configuration (see text), have been chosen.

With these α and γ values, the stochastic differential Eq. (33) is solved via the Euler-Maruyama method on the dimensionless interval [0, 7000]. In simulations, it is observed that 7000 is longer than the mean escape time required to switch from the “wrong” to the “correct” (depending on the desired logic outcome) well; this time length also ensures the expected logical output for a large number of trials. However, the fundamental observation is that the desired logical output occurs consistently and robustly only in an optimal range of noise values, in line with the tenets of stochastic resonance [121,122]. In the absence of a noise floor, this model does not work correctly; orbits may be trapped in the wrong well. Increasing the noise intensity beyond its optimal range leads to random switching between wells and the output no longer conforms to the truth tables. To quantify this behavior with respect to noise in this (designed) logic gate I measured its performance as defined as the ratio of success in realizing the desired gate over the total number of attempts; this ratio is the probability of realizing the desired gate and is shown (for OR and AND gates) in Fig. 11.3. I note that the probabilities in the left panel of Fig. 11.3 do not take the value unity, in contrast to the results presented in [109]. This can be traced back to the structure of the potential function for this GRN model, which is bistable only in a restricted regime of parameter values [115]; hence this model does not yield enough dynamic range to realize a failure-free implementation of the LSR paradigm. Moreover, in our performance definition, for each noise value I have checked the agreement between the simulated logical outputs for all the three data inputs ((0, 0), (0, 1)/(1, 0), and (1, 1)) and the respective truth table values of the gate under study. If one of the outputs does not realize the desired gate, I mark that as a failure. If, for example, we consider one of the panels in Fig. 11.2 for each noise value the least robust potential configuration (among the three plotted) will have the highest influence on the performance quality of this considered gate. This procedure is repeated 500 times. Different combinations of parameters have been tried, but unsuccessfully, because of the restricted dynamic range implicit in the model and the biological properties that are endemic to bacteriophage λ. This has lead us to propose a new version of the LSR principle, via a manipulation of the “conventional” LSR principle [109], to achieve a bacteriophage λ configuration that is still biologically correct.

As detailed above, the LSR paradigm works in the range of α and γ parameters that induce bistability, and for all the distinct logic input sets (0,0), (0,1), (1,0) and (1,1). To “adjust” the bacteriophage λ to conform to the LSR paradigm implies limiting the parameter interval to a narrow region. In the enhanced LSR paradigm, the idea is to encode inputs as parameters of the GRN model so that the undesired well (almost) disappears and to take advantage of stochastic resonance for the cases where the unwanted well cannot be removed from the potential function, *U(x).* The second case usually happens when the inputs are (0,1)/(1,0). With this proposed model, we are still working in a parameter interval that is biologically meaningful, without restricting our study to the bistable region [115]. In other terms, it can happen that the (0,0) or (1,1) cases can be realized when *U(x)* is monostable as shown in Fig. 11.2 (red curve of the upper left panel). By “controlling” the second parameter, γ, we can deepen either well selectively; hence with the appropriate amount of noise, trajectories will switch to the deeper well and remain there. This updated model for computing is, underpinned by the (numerically obtained) data inputs α=6.3, 9.8, and 13.1. I note now that γ = 50 yields the AND gate and γ = 36 the OR gate. All numbers for the α definition and the γ values have been obtained through several simulations (as mentioned above). The potential functions for AND and OR gates for different data inputs are presented in Fig. 11.2 (lower panels) with the probability of realizing these gates, using the modified paradigm, shown in Fig. 11.3 (right panel). I note that the two gates are robust to noise in the same range of noise and amenable to the design of a morphable logic gate; in addition we observe a range of noise intensities for which P(logic) = 1. The enhanced LSR paradigm yields greater robustness to external fluctuations.

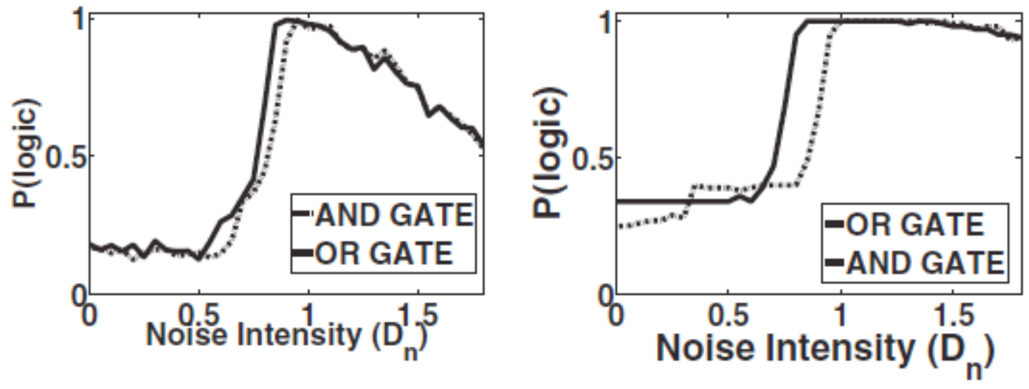


Fig. 11.3. Performance of logic gates OR and AND using the “conventional” LSR paradigm [109] (left), and the modified paradigm (see text) (right). α and γ values as in the text.

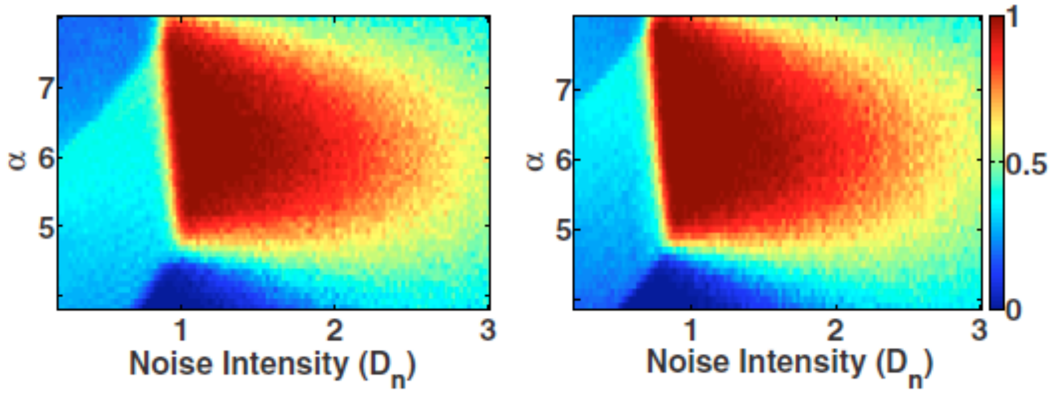


Fig. 11.4. Performance of logic gates AND (left) and OR (right) vs. noise intensity D_n, and α. γ values as in the text.

Finally I note that, to obtain the best performance in the logic gates, two possible solutions can be assessed: the change of noise intensity [123,124] or the variation of the parameter values, thereby adjusting the system dynamics to an optimal configuration, so that P(logic)$\sim$1 as desired; for a nonlinear system this is tantamount (as already noted earlier) to changing the transfer characteristic, thereby "tuning" the noise. In Fig. 11.4, the gate performance is plotted vs. noise intensities and α values (while γ =50 for the AND gate and γ =36 for the OR gate). For a fixed value of noise (for example the one mandated by nature) it is possible to select the "best" α value. It is interesting to note that (for our particular choice of model parameters) if noise intensity values are in the [0.7, 1] regime, there is a reasonably large range of α values for which P(logic) $\sim$1, as desired.

CHAPTER 12

DISCUSSIONS

In this research I introduced a different direction for computation and demonstrated how the library of patterns and behaviors of a chaotic system can be used in computation. Different digital functions can be built based on a chaotic system.

Here I introduced a new analytical analysis for chaos computing using periodic orbits and UPO theory. A direct connection (in terms of UPOs) between computational functionality of a chaotic system and the dynamics of the chaotic system was introduced. Based on the connection one can obtain the functionality and robustness of a chaotic system in doing computation from the dynamical equations of the system.

Furthermore, I showed how one can obtain and estimate the functionality and robustness of a chaotic system in doing computation just by using a time series without knowing the exact dynamical equation of the system. The main idea is reconstruct an UPO-based approximation for the chaotic system and use it for estimating the functionality and robustness. The simulation results showed that based on an UPO-based approximation one can easily obtain the functionality of the underlying chaotic system and estimate the robustness of these functions.

After studying single Chaogates, I proposed a new computer architecture to put the Chaogate together to build a sophisticated computing system out of them. The new computer has a flexible instruction set, in which the user can load his desired optimal one to the computer. Such flexibility enables the computer to meet the requirements of the application. For example, for statistical computation, an instruction set suitable for statistical calculation can be loaded to maximize the performance. For digital signal processing a suitable instruction set for DSP can be loaded. Furthermore, a user can have his own customized instruction set.

A software for simulating the hardware was developed to test the architecture and to demonstrate the capability of the hardware in running different instruction sets. C++ language was used to develop the simulator. Object oriented features of the C++ enables us to represent the architecture in terms of its basic blocks like wires, chaos based logic blocks, pins, registers, etc. A C++ class was defined for any type of component used in the architecture. For any instance of the defined component type (class) which is used in the architecture, an object was declared. The inputs to the hardware simulator were (1) Instruction set (2) program in terms of binary instructions.

This hardware simulator is a critical step in chaos computing, first it demonstrated how and how well the single chaos based logic blocks can be combined to build a processor, and second it bridges the software simulations to the physical hardware fabrications.

Apart from digital computation, it is explained how a chaotic system can be used in speech coding and synthesizing and it is demonstrated that such a chaos-based coder can be efficient than conventional methods like CELP. In our chaos based excited linear predictor coding method, chaotic systems were used for exciting a filter that models the vocal tract. The main advantages of chaos excited linear predictor are: 1- Since the waveforms are generated online there is no memory limitations and a more number of waveforms can be used to find better excitations resulting in better performance. 2- The waveforms are not read from a memory, and they are generated by a just a few basic machine instructions, therefore the new method can be faster than the previous method. 3- The new method is more biological oriented. 4- CMLs are known to model the turbulence and we already know the turbulence in our speech production system produces the random like excitations. Therefore the current work introduces the possibility of direct estimation of random like excitations in speech waveform by use of a CML.

As another approach for digital computation, I implemented and enhanced LSR in a GRN, specifically, the bacteriophage λ. The resultant computing device is able to work as an AND or OR gate interchangeably in the presence of noise. LSR on a GRN, that has the capability of being reconfigured, could be combined, in the near future, with other logic modules (done by different sets of input/output signals) to increase the computational power and functionality of an engineered GRN.

The research and the results of this thesis open new doors to us and new threads of research have already started following it. The main topics of future research areas are listed below.

Fabricating chaos-based processor is one of the main aims to reach after graduation. As a part of Ph.D. project, a hardware simulator software was presented for a chaos based computer. This hardware simulator is very beneficial in physical fabrication of the processor, because it defines and simulates the processor in terms of a group of basic blocks that have been already fabricated. So far different circuit implementations have been introduced for the chaos-based logic block [57,58,59,60,61] and a proof of concept VLSI implementation is presented [125]. However it might be required to introduce a new VLSI fabrication for the chaos based logic block, which is more compatible with the introduced computer architecture. There are hardware description languages like VHDL or Verilog, which can synthesize the architecture by use of provided library of layouts for the basic blocks, used in the architecture. The hardware simulator software I developed is very similar to a VHDL program that describes the architecture in terms of its blocks. Therefore having such a software in hand will make developing the VHDL program easy in this stage of the work. The output of the hardware description language will be a layout for fabricating a chaos based processor chip.

Another filed of research following this thesis is extending the UPO and time series analysis introduced in chapters 4-8 to higher dimensional chaotic maps and chaotic flows.

In chapters 4-6 it is demonstrated how the functionality and the robustness of a one-dimensional chaotic map in computation can be obtained from its UPOs. The techniques can be extended to be higher dimensional system and flows too. As to future work, the focus will be on improving, adapting, and changing the introduced techniques to derive the instruction set of a given higher dimensional chaotic map or chaotic flow from its dynamic equations and to estimate the robustness of these systems in performing computation.

Also in chapters 7 and 8 it was presented how one can obtain and estimate the instruction set and the robustness of the instruction set of a given 1-

D map from a given time series. With some modifications these techniques can be applied to higher dimensional chaotic maps and flows as well. In the future these techniques will be improved to be applicable to higher dimensional chaotic maps and flows as well.

Designing a new LSR based computer is another future project. Herethe idea of logical stochastic resonance is explained and it was adapted to a GRN model. The constructed logic block was able to build simple functions like AND or OR. To perform sophisticated computations a group of these blocks are required to work together. In chapter 9 the idea of designing a new architecture for a chaos-based computer was introduced. The exact same architecture will not be applicable to the LSR paradigm, because chaos computing and LSR have different instruction sets and timings. However a similar architecture can be developed for the LSR computing paradigm.

REFERENCES

[1] G. Ifrah, The Universal History of Numbers: From Prehistory to the Invention of the Computer, (John Wiley & Sons Inc, New York, 2000).

[2] A. S. Brooks, and C. C. Smith, The African Archaeological Review **5**, 65 (1987).

[3] Royal Belgian Institute of Natural Sciences, WWW document, (http://www.naturalsciences.be)

[4] D. Schmandt-Besserat, Documenta Praehistorica **26**, 21 (1999).

[5] G. Ifrah, The Universal History of Computing: From the Abacus to the Quantum Computer, (John Wiley & Sons, New York, 2001)

[6] Computer history museum, Mountain View, California, WWW document, (http://www.computerhistory.org/revolution/artifact/37/131)

[7] R. Bud and D. J. Warner, Instruments of science: an historical encyclopedia, (Garland, New York, 1998)

[8] Cultural China, WWW document, (http://kaleidoscope.cultural-china.com/en/137K6K6434.html)

[9] T. Freeth, Tony; Y. Bitsakis, X. Moussas, etc., Nature **444**, 587 (2006).

[10] National Archaeological Museum, Athens, WWW document, (http://odysseus.culture.gr/h/4/eh430.jsp?obj_id=5582)

[11] The astrolabes, WWW document, (http://astrolabes.org/)

[12] T. Oxley, The Celestial Planispheres Or Astronomical Charts, (Kessinger Publishing, Whitefish, 2010)

[13] J. Evans, History and practice of ancient astronomy, (Oxford University Press, Oxford, 1998)

[14] The Babbage engine, WWW document, (http://www.computerhistory.org/babbage/modernsequel/)

[15] The science museum, London, WWW document, (http://www.sciencemuseum.org.uk/)

[16] G. C. Chase, IEEE Annals of the History of Computing **2**, 204 (1980).

[17] MIT online museum, WWW document, (http://webmuseum.mit.edu)

[18] V. Bush, F. D. Gage and H. R. Stewart, Journal of the Franklin Institute **203**, 63 (1927).

[19] D. R. Hartree, nature **146**, 319 (1940).

[20] G. O'Regan, A Brief History of Computing, (Springer-Verlog, London, 2008).

[21] C. E. Shannon, Transactions of the American Institute of Electrical Engineers **57**, 713 (1938).

[22] S. Sinha and W. L. Ditto, Phys. Rev. Lett. **81**, 2156 (1998).

[23] S. Sinha and W. L. Ditto, Phys. Rev. E **60**, 363 (1999).

[24] A. Miliotis, K. Murali, S. Sinha, W. L. Ditto and M. L. Spano, Chaos, Solitons & Fractals **30**, 809 (2009) .

[25] K. Murali, A. Miliotis, W. L. Ditto and S. Sinha, Phys. Lett. A **373**, 1346 (2009).

[26] W. L. Ditto, A. Miliotis, K. Murali, S. Sinha and M. L. Spano, Chaos **20,** 037107 (2010).

[27] J. P. Crutchfield, W. L. Ditto and S. Sinha, Chaos **20**, 037101 (2010).

[28] H. R. Pourshaghaghi, R. Ahmadi, M.R. Jahed-Motlagh, B. Kia Int. J. of Bifurcation and Chaos **20**, 715 (2010).

[29] K. Murali, S. Sinha, and W. L. Ditto, Pramana Journal of Physics **64**, 433 (2005).

[30] J. Guckenheimer, and P. Holmes, Nonlinear Oscillations, Dynamical Systems, and Bifurcation of Vector Fields, (Springer-Verlag, New York, 1993).

[31] S. Wiggins, Introduction to Applied Nonlinear Dynamical Systems and Chaos, (Springer-Verlag, New York, 2003).

[32] P. A. Corning, complexity **7**, 18 (2002).

[33] P. Weiss, The living system: Determinism stratified. In: Beyond reductionism: New perspectives in the life sciences; A. Koestler, J. R. Smythies, Eds. (The Macmillan Co, New York, 1969).

[34] J. Goldstein, Emergence as a Construct: History and Issues. Emergence **11**, 49 (1999).

[35] R. C. Hilborn, Chaos and nonlinear dynamics, second edition, (Oxford university press, New York, 2000).

[36] C.E. Shannon, , Bell System Technical Journal **27**, 379 (1948).

[37] M.R. Jahed-Motlagh, B. Kia, W.L. Ditto, S. Sinha, Int. J. Bifur. Chaos **17**, 1955 (2007).

[38] R. L. Devaney, An introduction to chaotic dynamical systems, (Westview press, Boulder, 2003).

[39] E. Ott, Chaos in dynamical systems, (Cambridge university press, New York, 1993).

[40] H. D. I. Abarbanel, Analysis of observed chaotic data, (Springer-Verlag, New York, 1996).

41 E. Ott, C. Grebogi, and J. A. Yorke, Phys. Rev. Lett. **64**, 1196 (1990).

42 W. L. Ditto, S. N. Rauseo and M. L. Spano, Phys. Rev. Lett. **65**, 3211 (1990).

43 K. Pyragas, Continuous Control of Chaos by Self-controlling Feedback, Phys. Lett. A **170**, 421 (1992).

[44] B. R. Andrievskii and A. L. Fradkov, Automation and Remote Control **64**, 673 (2003).

[45] E. A. Jackson, and I. Grosu, Physica D **85**, 1 (1995)., vol. 85,

[46] J. Alvarez-Gallegos, J. Dynam. Control **4**, 277 (1994).

[47] A. Babloyantz, A. P. Krishchenko, and A. Nosov, Comput. Math. Appl. **34**, 355 (1997).

[48] L. Q. Chen, and Y. Z. Liu, Nonlin. Dynam. **20**, 309 (1999).

[49] C. Grebogi and Y. C. Lai, IEEE Trans. on circuits and systems-I **44**, 971 (1997).

[50] S. Sinha and D. Biwas, Phys. Rev. Lett. **71**, 2010 (1993).

[51] D. C. Dracopoulos and A. J. Jones, Neural Comput. Appl. **6**, 102 (1997).

[52] D. P. A. Greenwood, R. A. and Carrasco, IEE Proc. Commun. **147**, 285 (2000).

[53] K. Tanaka, T. Ikeda, H. O. Wang, IEEE Trans. Circ. Syst. I **45**, 1021 (1998).

[54] L. Pecora and T. Carroll, Phys. Rev. Lett. **64**, 821 (1990)

[55] N. F. Rulkov, M. M. Sushchik, and L. S. Tsimring, Phys. Rev. E **51**, 980 (1995).

[56] H. D. I. Abarbanel1, N. F. Rulkov, and M. M. Sushchik, Phys. Rev. E **53**, 4528 (1996).

[57] K. Murali, S. Sinha, W. L. Ditto, IJBC (Letters) **13**, 2669, (2003).

[58] K. Murali, S. Sinha, W. L. Ditto, Phys. Rev. E **68**, 016205, (2003).

[59] K. Murali, S. Sinha ,W.L. Ditto, Pramana-J. Phys. **64**, 433 (2005).

[60] M. R. Jahed-Motlagh, and B. Kia, IEEE Asia Pacific Conference on Circuits and Systems, 1826 (2007).

[61] H. R. Pourshaghaghi, B. Kia, W. L. Ditto, M. R. Jahed-Motlagh, Chaos, Solitons and Fractals **41**, 233 (2008).

[62] H. R. Pourshaghaghi, R. Ahmadi, M. R. Jahed Motlagh, B. Kia, IJBC **20**, 715 (2010).

[63] D. Auerbach, P. Cvitanovic, J.-P. Eckamnn, G. H. Gunaratne and I. Procaccia, Phys. Rev. Lett. **58**, 2387 (1987).

[64] P. Cvitanovic, Phys. Rev. Lett. **61**, 2729 (1988).

[65] P. Cvitanovic, Physica D **51**, 138 (1991).

[66] R. Artuso, E. Aurell, P. Cvitanovic, Nonlinearity **3**, 325 (1990).

[67] R. Badii, E. Brun, M. Finardi, L. Flepp, R. Holzner, J. Parisi, C. Reyl, and J. Simonet, Rev. Mod. Phys. **66**, 1389 (1994).

[68] R. Gilmore, The Topology of Chaos: Alice in Stretch and Squeeze Land, (John Wiley & Sons Inc., New York, 2002).

[69] V. Patidar, Electronic Journal of Theoretical Physics **3**, 29 (2006).

[70] B. Kia, M. L. Spano and W. L. Ditto, Phys. Rev. E **84**, 036207 (2011).

[71] "Chaotic computer design using time series", B. Kia, M. Spano, W. L. Ditto, submitted to Physical Review E.

[72] "Unstable periodic orbits and the effect of noise in chaos computing", B. Kia, W. L. Ditto, M. Spano, Submitted to Chaos (invited paper to special focus issue).

[73] F. Christiansen and A. Politi, Phys. Rev. E **51** 3811 (1995).

[74] Y. Hirata, K. Judd, D. Kilminster, Phys. Rev. E **70**, 016215 (2004).

[75] M. Casdagli, Physica D **35**, 335 (1989).

[76] J. McNames, in Proceedings of International Workshop on Advanced Black-Box Techniques for Nonlinear Modeling (Leuven, Belgium, July 8-10, 1998).

[77] C. Maxfield, The design warrior's guide to FPGAs: Devices, tools and flows, (Elsevier, Amsterdam, 2004).

[78] W. L. Ditto, and T. Munakata, Communications of the ACM **38**, 96 (1995).

[79] R. T. Sataloff and M. Hawkshaw, Chaos in Medicine: Source Readings, (Singular Publishing Group, San Diego, 2001).

[80] J. N. Weiss, A. Garfinkel, M. L. Spano and W. L. Ditto, J. Clin. Invest. **93**, 1355 (1994).

[81] M. Varela, R. Ruiz-Esteban, M. J. M. De Juan, Perspectives in Biology and Medicine **53**, 584 (2010).

[82] A. Garfinkel, J. N. Weiss, W. L. Ditto and M. L. Spano, Trends in Cardiovascular Medicine **5**, 76 (1995).

[83] A. Garfinkel, M. L. Spano, W. L. Ditto and J. Weiss, Science **257**, 1230 (1992).

[84] J. P. Crutchfield, W. L. Ditto and S. Sinha, Chaos **20**, 037101 (2010).

[85] E. Bollt, IJBC **13**, 269 (2003).

[86] E. Bollt and Y.C. Lai, Phys. Rev. E **58**, 1724(1998).

[87] M. S. Fee, B. Shraiman, B. Pesaran and P. P. Mitra, Nature **395**, 67 (1998).

[88] T. Fitch, H. Herzel, J. Neubauer, and M. Hauser, Anim. Behav. **63**, 407 (2002).

[89] J. Awrejcewicz, J Sound Vibrations **136**, 151 (1990).

[90] H. Herzel, Appl Mech Rev. **46**, 399 (1993).

[91] D. A. Berry, H. Herzel, I. R. Titze and K. Krischer, J Acoust Soc Am. **95**, 3595(1994).

[92] H. Herzel, D. Berry, I. Titze and I. Steinecke, Chaos **5**, 30 (1995).

[93] P. Mergell, H. Herzel, I. R. Titze, J Acoust Soc Am. **108**, 2996 (2000).

[94] Jiang JJ, Zhang Y, Stern J. J Acoust Soc Am. **110**, 2120 (2001).

[95] Y. Zhang, J. J. Jiang, Acoust Soc Am. **115**, 1266 (2004).

[96] D. A. Berry, H. Herzel, I. R. Titze and B. H. Story, J Voice **10**, 129 (1996).

[97] J. G. Svec, K. S. Harm and D. G. Miller, J Acoust Soc Am **106**, 1523 (1999).

[98] J. J. Jiang, Y. Zhang Y and C. N. Ford, J Acoust Soc Am **114**, 1 (2003).

[99] R. J. Bake, J Voice **4**, 185, (1990).

[100] A. Kumar and S. K. Mullick, J Acoust Soc Am. **100**, 615 (1996).

[101] J. R. Deller, J. H.L Hansen, and J. G. Proakis, Discrete- Time Processing of Speech Signals, (IEEE Press, New York, 1993).

[102] W. Kleijn, D. J. Krasinki, and R. H. Ketchum, IEEE Trans. on acoustic, speech. and signal processing **38**, 1330 **(**1990).

[103] I. McLoughlin, R. Chance, Electronic Letters **33**, 743 (1997).

[104] L. M. Dasilva, A. Alcaim, IEEE Signal Processing Letters **2**, 44 (1995).

[105] B. Kia, W. L. Ditto, M. Spano, Chaos for Speech Coding and Production, C.M. Travieso-Gonzalez and J.B. Alonso-Hernandez (Eds.), LNAI 7015, 270 (Springer, Heidelberg, 2011)

[106] T. Bohr, O. B. Christensen, Phys. Rev. Let. **63**, 2161 (1989).

[107] A. R. Bulsara, A. Dari, W. L. Ditto, K. Murali, and S. Sinha, Chem. Phys. **375**, 424 (2010).

[108] A. Dari, B. Kia, A. R. Bulsara, and W. L. Ditto, Europhys. Lett. **93**, 18001 (2011).

[109] K. Murali, S. Sinha, W. L. Ditto, and A. R. Bulsara, Phys. Rev. Lett. **102**, 104101 (2009).

[110] A. Dari, B. Kia, X. Wang, A. Bulsara, W. L. Ditto, Phys. Rev. E **83**, 041909 (2011).

[111] "Logical stochastic resonance with correlated internal and external noises in a synthetic biological logic block", A Dari, B. Kia, A. Bulsara, W. L. Ditto, Submitted to Chaos (invited paper to special focus issue).

[112] H. H. McAdams and L. Shapiro, Science **269**, 650 (1995).

[113] A. D. Johnson, A. R. Poteete, G. Lauer, R. T. Sauer, G. K. Ackers and M. Ptashne, Nature **294,** 217 (1981).

[114] M. Ptashne, A Genetic Switch: Phage λ and Higher Organisms, 2nd edition (Cell Press & Blackwell Scientific, Cambridge, Mass.) 1992.

[115] J. Hasty, J. Pradines, M. Dolnik M. and J. J. Collins, Proc. Natl. Acad. Sci. U.S.A. **97**, 2075 (2000).

[116] J. Hasty, F. Isaacs, M. Dolnik, D. McMillen and J. J. Collins, *Chaos* **11**, 207 (2001).

[117] V. K. Kumar, O. Lockerbie, S. D. Keil, P. Ruane, M. Platz and C. Martin, Photochem. Photobiol **80**, 15 (2004).

[118] S. Atsum and J. W. Little, Proc. Natl. Acad. Sci. U.S.A. **103**, 19045 (2006).

[119] F. J. Isaacs, J. Hasty, C. R. Cantor and J. J. Collins, Proc. Natl. Acad. Sci. U.S.A. **100**, 7714 (2003).

[120] D. Guerro, A. R. Bulsara, W. L. Ditto, S. Sinha, K. Murali and P. Mohanty, Nano. Lett. **10,** 1168 (2010).

[121] K. Wiesenfeld and F. Moss, Nature (London) **373**, 33 (1995).

[122] A. R. Bulsara and L. Gammaitoni, Phys.Today **49**, 39 (1996).

[123] T. Lu, M. Ferry, R. Weiss and J. J. Hasty, Phys. Biol. **5,** 036006 (2008).

[124] K. F. Murfy, R. M. Adams, X. Wang, G. Bal´azsi and J. J. Collins, Nucl. Acids Res. **38,** 2712 (2010).

[125] W. L. Ditto, A. Miliotis, K. Murali, S. Sinha, Review of Nonlinear dynamics and Complexity **3,** 1 (2010).