Correcting Errors in Private Communications Between Synchronized Chaotic Circuits

University of Missouri - St. Louis

Warren Li

May 2017

Contents

Abstract

In this paper, we analyze Chua's attractor and build two Chua circuits. These two circuits are then synchronized in a master/slave configuration. Proof of synchronization is obtained by plotting y against y_r, which denotes the transmitter and receiver respectively: the result is a 45° line. Afterwards, a bit stream was sent by using a digital potentiometer to modulate the parameter β. The message was recieved using an audio input, and the estimation of the message had an error rate of 0.5%. By incorporating an error-correcting cyclic code along with a Meggitt decoding algorithm, the transmission time increased by approximately one-third, and all remaining errors were resolved.

Chapter 1

Introduction

In the 1960s, Edward Lorenz was exploring a weather model. Specifically, he ran computer simulations on the following system of three nonlinear differential equations. This deterministic system was a simplified model of atmospheric convection, yet possessed complex dynamical behaviors.

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

Instead of restarting the program, Lorenz took a shortcut one day and ran the model at some later time step, using rounded values from a previous run as the initial input [1]. What he noticed were two time series that rapidly diverged. This sensitive dependence on initial conditions is a key feature of any chaotic system, and Lorenz described the implications of this phenomenon on long-range weather prediction in 1963 [2]. Interest in nonlinear dynamics expanded rapidly throughout the subsequent decade, as well as its applications to physics, chemistry, biology, and engineering.

In 1990, Pecora and Carroll demonstrated that two chaotic systems could synchronize, thus marking the early days of chaos communications [3]. Cuomo and Oppenheim published related findings in 1993 that expanded upon the previous literature [4]. They described an analog communication scheme using two synchronized circuits, which were the first electronic implementations of the Lorenz system. These circuits were then used in a signal-masking technique to add a high-power chaotic "mask" to the message. A digital scheme was also described.

In this paper, we look at digital signals based on parameter modulation of two synchronized Chua circuits. A practical implementation for sending bit streams

is explored. In addition, we describe the engineering process used in the signal transmission and signal estimation codes. Minor parameter adjustments are introduced to alleviate the number of errors. In the remaining sections, a cyclic error-correcting code is incorporated. Using the Meggitt decoding algorithm, one can recover the original message with greater accuracy.

The need for private communications has increased rapidly into the modern day and the unpredictability of chaos may be a novel way of providing additional security. Applications of coding theory may help mitigate the noise associated with sending signals through a channel, which may prove to be important for situations requiring high reliability.

Chapter 2

Exploring Chua's Attractor

2.1 Phase Portraits

We begin by analyzing Chua's attractor, which we will electronically implement in the next chapter. It is modeled by the following system of equations:

$$\begin{cases} \dot{x} = \alpha[y - x - g(x)] \\ \dot{y} = x - y + z \\ \dot{z} = -\beta y \end{cases} \tag{2.1}$$

Notice that α and β are parameters, and $g(x)$ is a piecewise linear function defined as follows:

$$\begin{cases} m_0 x + m_0 - m_1, \text{ if } x \leq -1 \\ m_1 x, \text{ if } -1 < x < 1 \\ m_0 x + m_1 - m_0, \text{ if } x \geq 1 \end{cases} \tag{2.2}$$

In the following analyses, we have fixed $\alpha = 10$, $m_0 = -8/7$, and $m_1 = -5/7$. In order to send private messages via a chaotic circuit, it is necessary to first identify the regions of chaotic behavior. To keep things simple, only β is varied throughout this chapter. However, the effects of changing α are briefly discussed at the end of this section. Lastly, the source code files used to generate the plots in this chapter are listed in Appendix B unless otherwise stated.

Using fourth-order Runge-Kutta numerical integration, one can obtain the solutions of the differential equations. They are plotted on a phase portrait, which shows the overall image of trajectories in phase space. As shown in Figure 2.1, for large values of β, we appear to get a stable spiral since the time series shows rapid

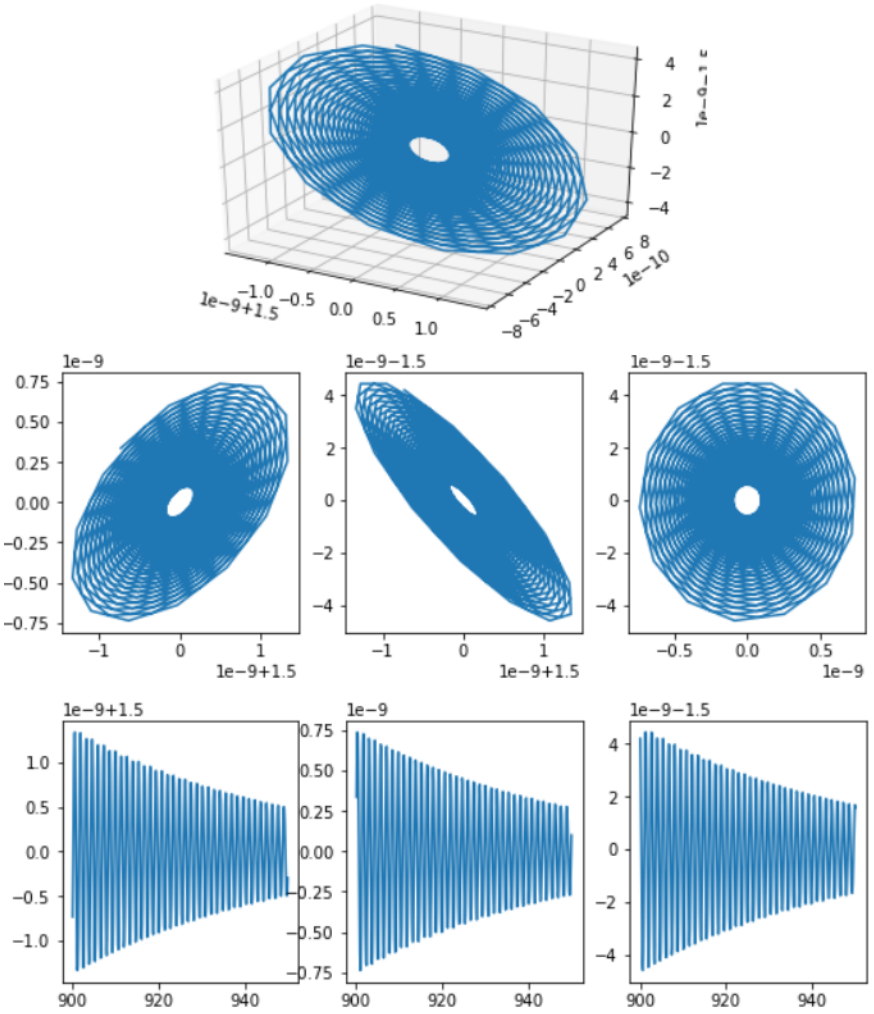


Figure 2.1: A phase portrait of a stable spiral when $\beta = 29$. From left to right, the middle row is a projection of the 3D-plot onto the xy-plane, xz-plane, and yz-plane. The bottom row shows the time series for x, y, and z respectively. 10000 iterations were performed, with 9000 transient points discarded. The time series only displays the first 500 points after the cutoff in order to avoid cluttering the plot.

decay. In addition, the magnitude of the scales are minuscule; almost one-billion times smaller when compared to our initial conditions $x_0 = 0.1$, $y_0 = 0.1$, and $z_0 = 0.6$. The transition into a spiral occurs somewhere between $\beta \approx 27.5$ and $\beta \approx 28.0$. However, if β is decreased below 27.5, the first signs of chaos appear.

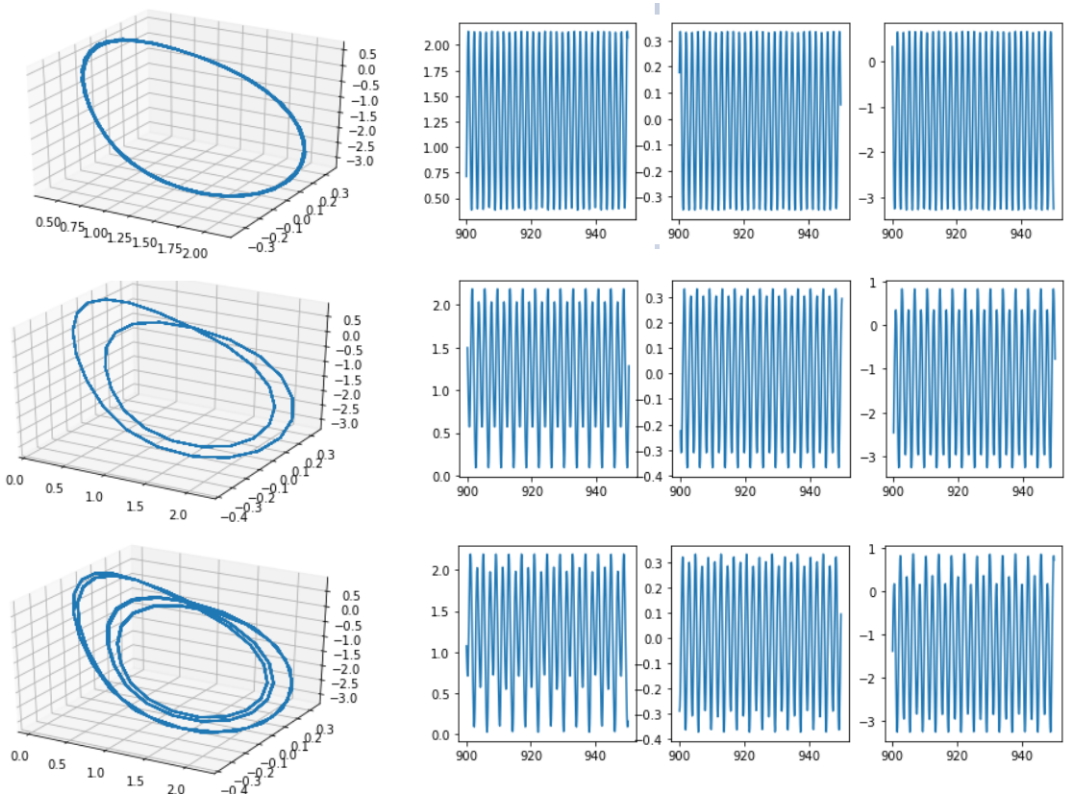


Figure 2.2: Phase portraits and time series for a period-1, period-2, and period-4 system from top to bottom. The parameter values are as follows: $\beta = 20$, $\beta = 18.5$, and $\beta = 18.2$. Although the top row appears to have a single period, the peaks and troughs are "wavy" and do not perfectly align. Yet, the phase portrait remains tightly clustered in a loop. Moreover, the middle plot appears to be less "dense" when compared with the other two portraits (thinner lines). This variation is similarly observed in later plots.

Based on the behavior depicted in Figure 2.2, we expect that Chua's attractor undergoes a period-doubling route to chaos. This is similar to many other systems such as the Lorenz or Rössler attractors. However, the "period-1" time series appears to have some inconsistencies. Consequently, we further investigate this parameter value and discuss the results in the upcoming sections.

As the parameter is lowered further, a single scroll in the attractor begins to manifest. This can be seen in Figure 2.3. The trajectories first fill in around the outer "ring," before gradually expanding inwards.

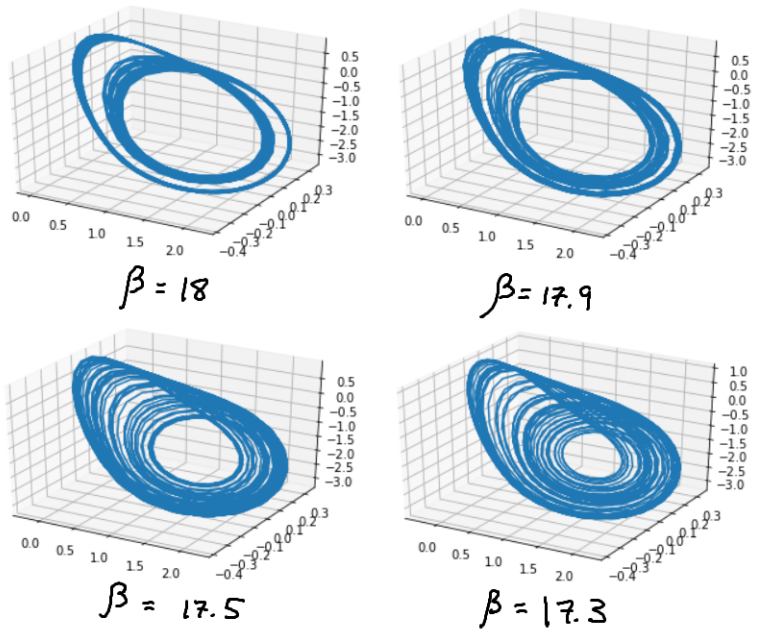


Figure 2.3: Appearance of the first scroll. The corresponding β values are labeled under each plot.

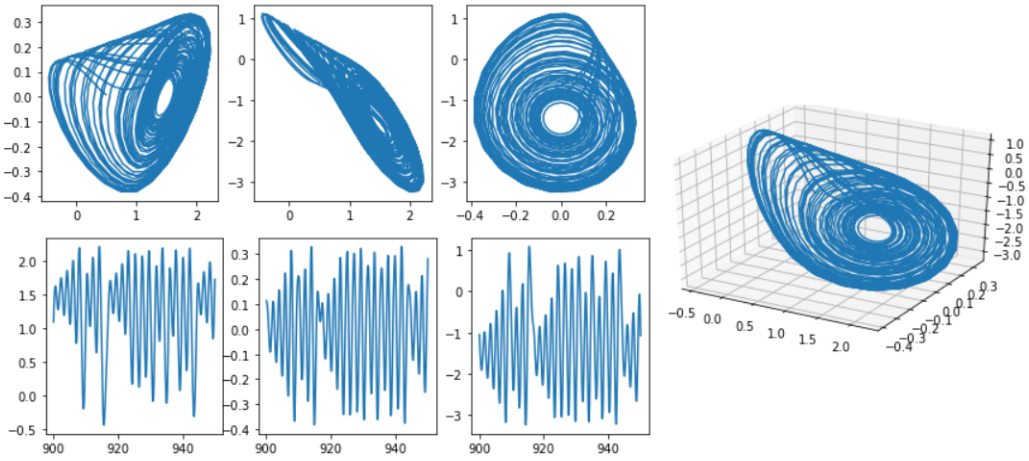


Figure 2.4: Phase portrait of a single scroll for $\beta = 17$. The top row shows the xy, xz, and yz projections from left to right. The bottom row are the time series, which clearly display chaotic behavior.

Around this point, the system starts behaving erratically. For small changes in β, the attractor switches between the single-scroll and double-scroll configurations. In addition, the orientation of the single scroll occasionally flips. During particular values, the trajectory tends to favor one scroll over the other, even though both are visibly present. A sampling of these phenomena are displayed in Figure 2.5.

This sensitivity continues throughout the transition into the double scroll, and no discernible pattern appears as β is reduced. With that said, there are a few general observations: below approximately $\beta = 16.5$, the single scroll ceases altogether. However, the spread of trajectories and proportion of time spent on each of the two scrolls continues to vary unpredictably until $\beta \approx 13.2$.

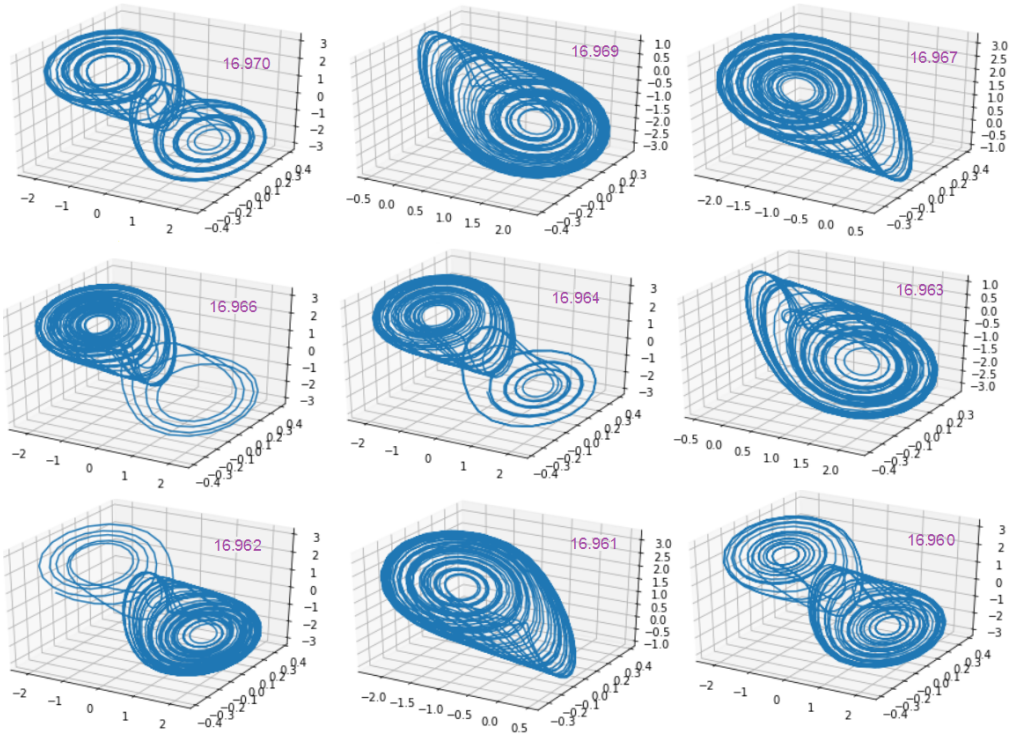


Figure 2.5: A sample of phase portraits between $\beta = 16.960$ and $\beta = 16.970$. The specific parameter corresponding to each plot is labeled on the upper-right corner. The attractor's appearance widely varies due to minor changes in β.

Fortunately, these differences are not noticeable when plotted for a physical circuit on an oscilloscope. Thus, we can operate in the double-scroll region. This is advantageous since there is a largest range of parameter values where chaos is maintained, but where the nature of the attractor is not prone to shift (to say, a

single scroll). However, it is still important to consider the remaining limit: at the lower extremes of β, an unstable spiral forms (Figure 2.7).

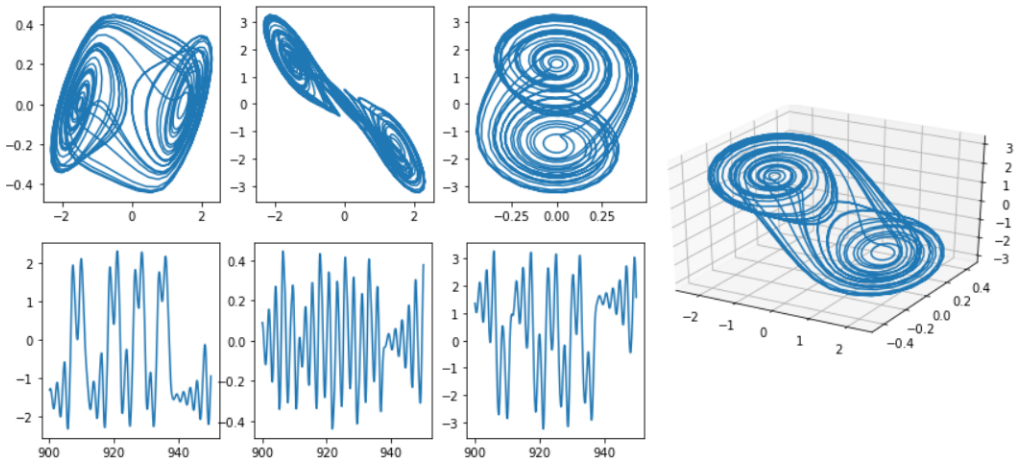


Figure 2.6: Phase portrait of the double-scroll attractor for $\beta = 14$. The time series also shows chaotic behavior like the single-scroll attractor, but has increased amplitude.

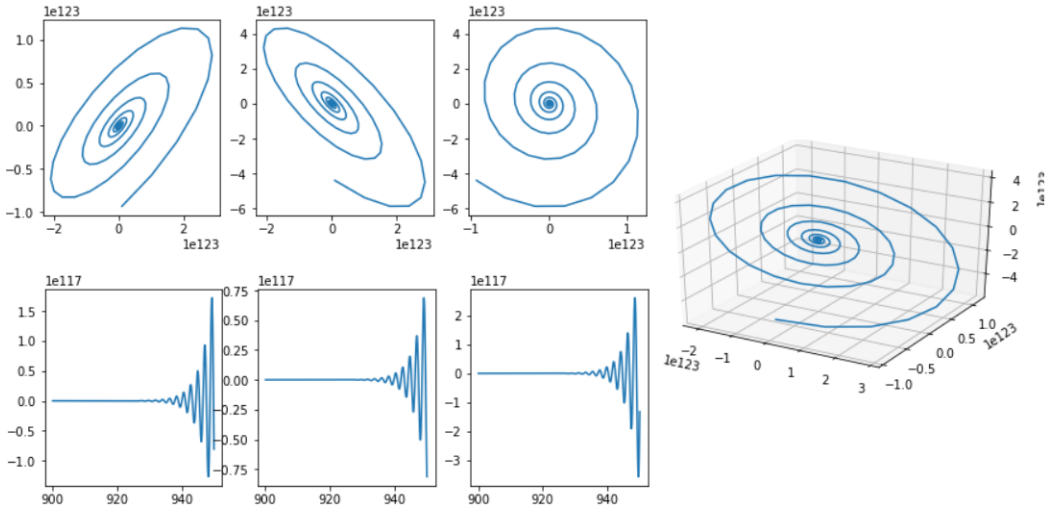


Figure 2.7: Phase portrait for an unstable spiral at $\beta = 13$. This can be discerned by observing that the time series exponentially increases and that the scale grows by orders of magnitude.

The α parameter is not included in our discussions since all of the behaviors observed by holding β constant and varying α have already been described. In fact, α and β appear to be inversely related. Increasing α tends to create an unstable spiral and decreasing α sufficiently results in a stable spiral. Also, when the circuit equations in Chapter 3 are nondimensionalized, the variable resistance corresponds to β.

2.2 Poincaré Sections

We can gain greater insight into the structure of an attractor by plotting Poincaré sections. This is essentially equivalent to cutting a cross-section through the attractor. In the following examples, we have chosen the plane $x = 1$ as our "slice". This concept is depicted in Figure 2.8.

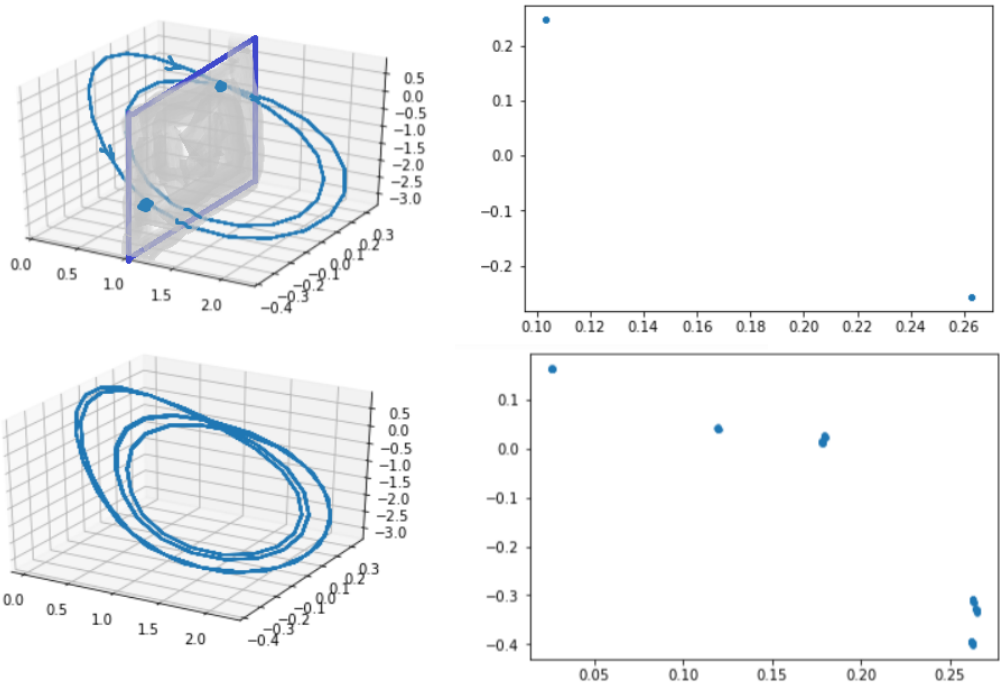


Figure 2.8: The top left is a figure of the attractor for $\beta = 18.5$ (period-2), with a plane intersecting the attractor at $x = 1$. The right-hand side shows the corresponding Poincaré section, which in this case, is simply two points. The bottom row shows the attractor and section for the period-4 system ($\beta = 18.2$).

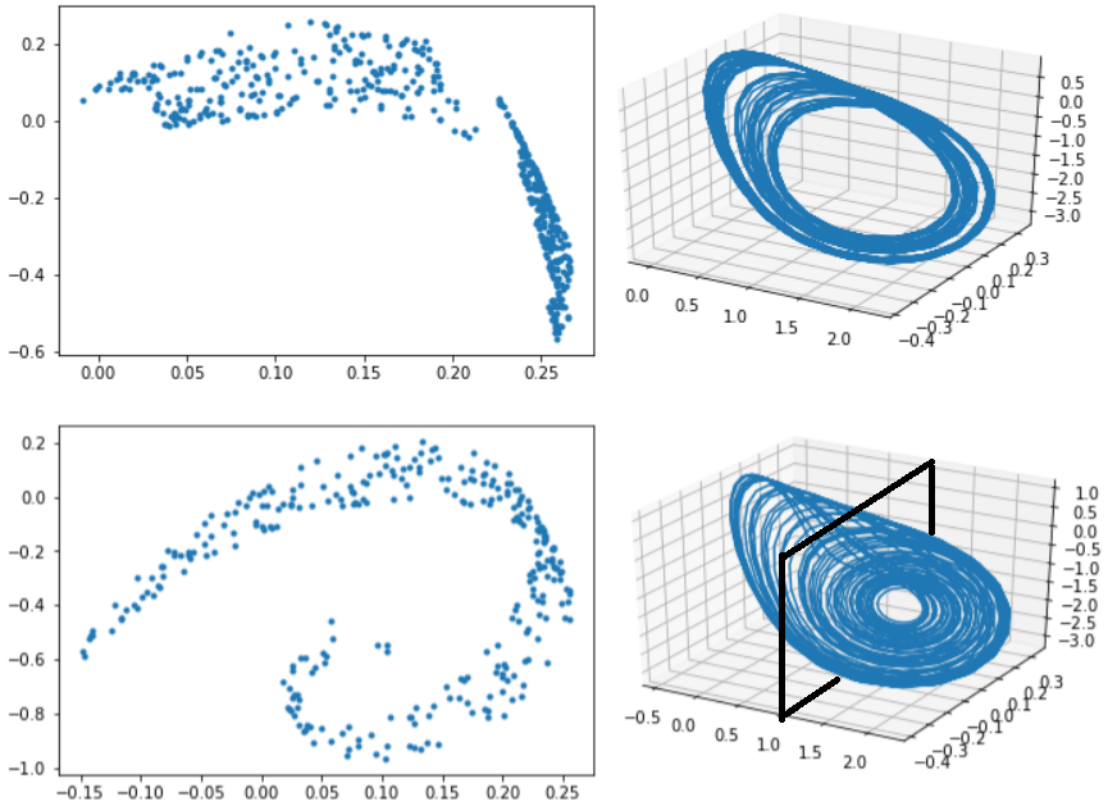


Figure 2.9: Poincaré sections for the single scroll. The top row corresponds to $\beta = 18$ and the bottom row corresponds to $\beta = 17$. Once again, the "slice" $x = 1$ is drawn for the lower-right attractor.

Notice that in Figure 2.9, there is a cluster of points on the right-hand side of the Poincaré section, which spirals inward as the single-scroll attractor is revealed. It is also important to note that by definition, only points crossing the $x = 1$ plane from a negative to positive direction are plotted. This explains why there are only two points in Figure 2.8 (top), even though there are technically four intersections with the plane.

These maps illustrate the folding and stretching mechanism behind chaotic attractors. For instance, the double-scroll attractor in Figure 2.10 (bottom) appears to be a continuation of the single scroll. Instead of spiraling in towards the center from one direction, the right-hand side of Figure 2.9 is "pushed" in and the tail is "pulled" out so that the end result appears to be a spiral being approached from two directions.

As described previously, there appears to be a sudden absence of points when $\beta = 16$. However, this seems to be the case because the points are grouped together in three regions (Figure 2.10, top). Even though the general behavior is still chaotic and the double-scroll attractor is apparent, perhaps some residual behavior from the single scroll remains. This may also be an analogue to other attractors where a system's chaotic behavior temporarily stops and enters into a period-3 region, before resuming period-doubling towards chaos. Yet in this situation, the chaos does not cease and is merely restricted.

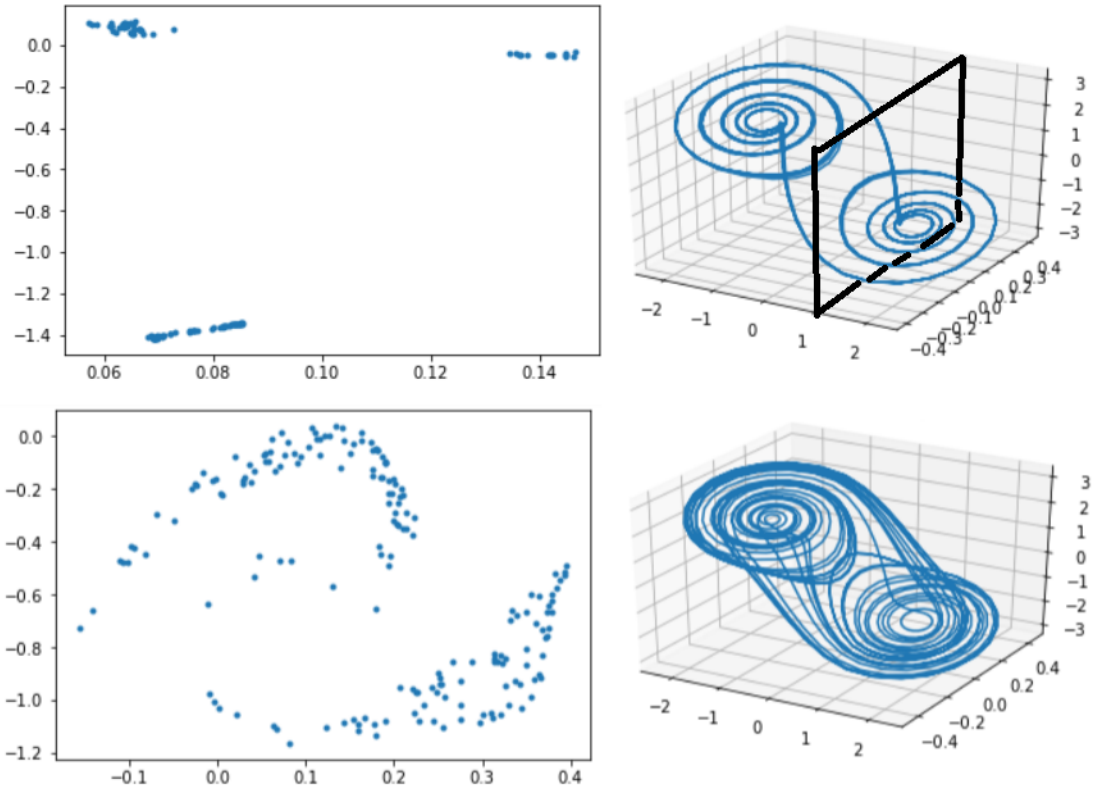


Figure 2.10: Poincaré sections for the double scroll. The above attractor corresponds to $\beta = 16$ and the one below refers to $\beta = 14$. There is an evident sparsity of points in the top Poincaré section. The lower Poincaré section however, more closely resembles the previous images seen in the single scroll.

2.3 Return Maps

In the remaining sections, we briefly discuss some common "tests" for chaos. If one finds the maximums of a time series and plots each subsequent peak with respect to the current peak, then the result is called a return map.

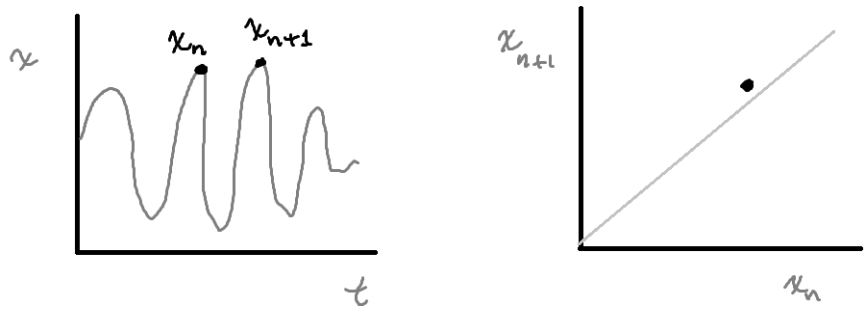


Figure 2.11: Two peaks of a time series are drawn on the left. On the right-side, x_{n+1} is plotted against x_n. In this figure, only one point is plotted. In actuality, this process should be repeated for all peaks.

If the data were truly random, then the return map would also appear to be a random distribution of points. However, a chaotic system will reveal some structure in the map. An example is illustrated in Figure 2.12 below.

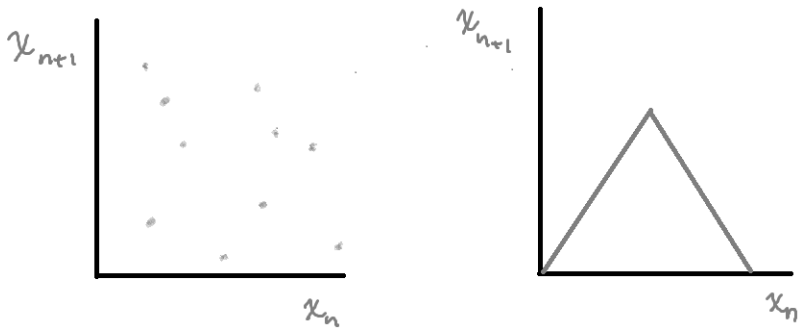


Figure 2.12: Two return maps. On the left is a return map for random data. The right side is a depiction of the tent map. This is a piecewise function defined as $f(x) = rx$ for $0 \leq x \leq 1/2$ and $f(x) = r - rx$ for $1/2 \leq x \leq 1$ [5].

In general, a system that undergoes a period-doubling route to chaos will have a single-peaked (unimodal) map. The following is an analysis of return maps for key parameter values previously explored. We begin with the period doubling region before discussing the single-scroll and double-scroll attractors.

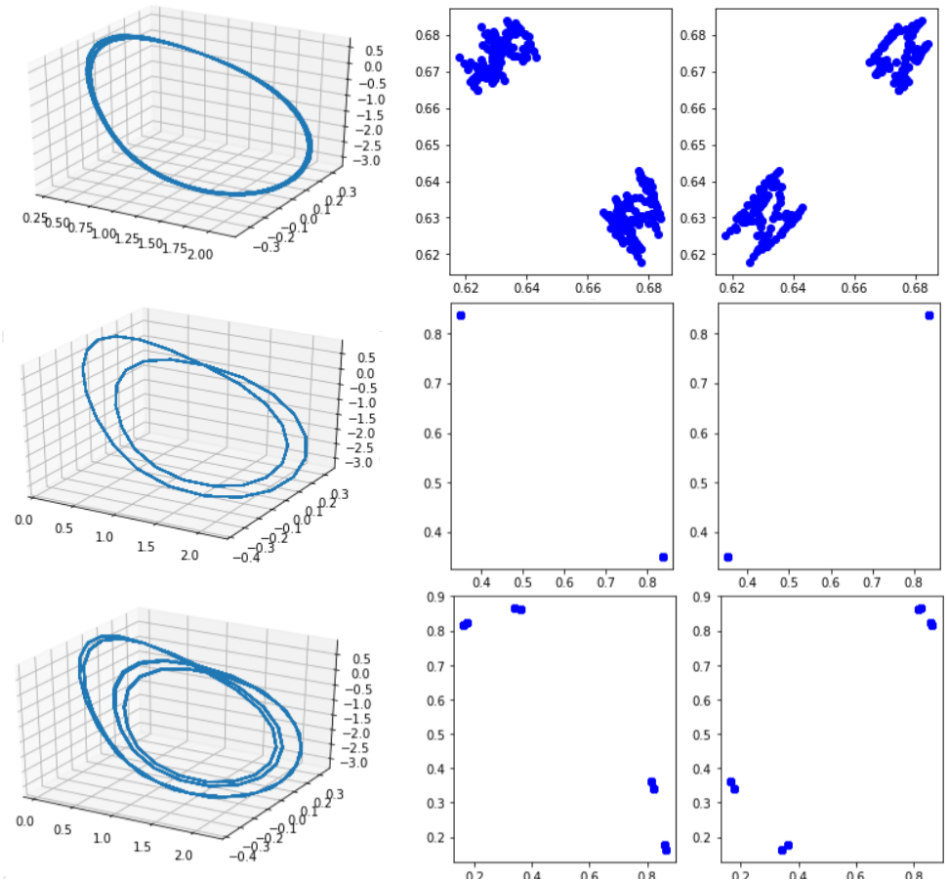


Figure 2.13: Images of the first and second iterated maps for $\beta = 19$, $\beta = 18.5$, and $\beta = 18.2$ from top to bottom. The second return map (rightmost column) plots x_{n+2} versus x_n. As discussed previously, upper plot seems to be cycling at one phase, although the return map shows a cluster of points. This suggests that zooming into the attractor may yield finer details, or that there is some other error such as computational precision. The period-2 attractor in the middle is consistent, with one point on each side of the 45° line. Likewise, the period-4 attractor mostly meets expectations, though it appears to be on the cusp of another period doubling.

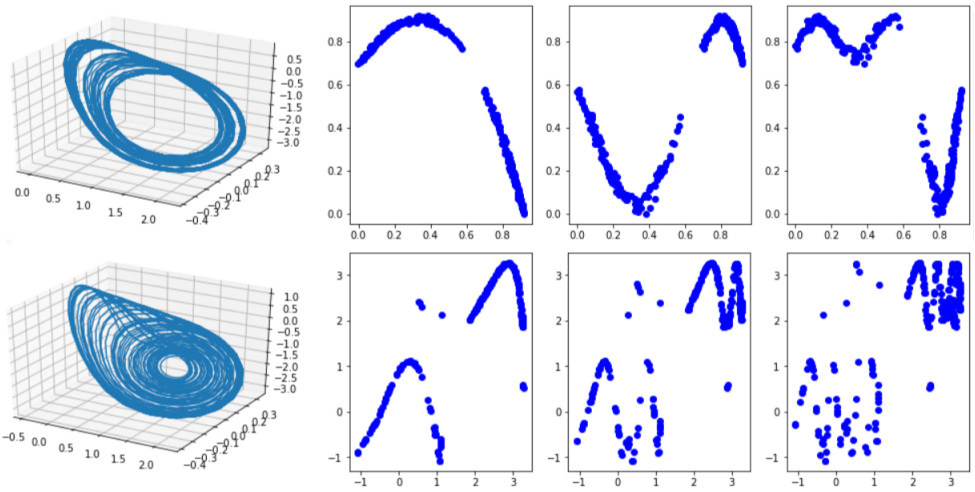


Figure 2.14: 1^{st}, 2^{nd}, and 3^{rd} return maps for $\beta = 18$ (upper) and $\beta = 17$ (lower). All return maps refer to the x variable. Note that there appears to be a slightly cutoff unimodal return map. This suggests that the system does indeed undergo a period-doubling route to chaos. However, this later turns into a bimodal map.

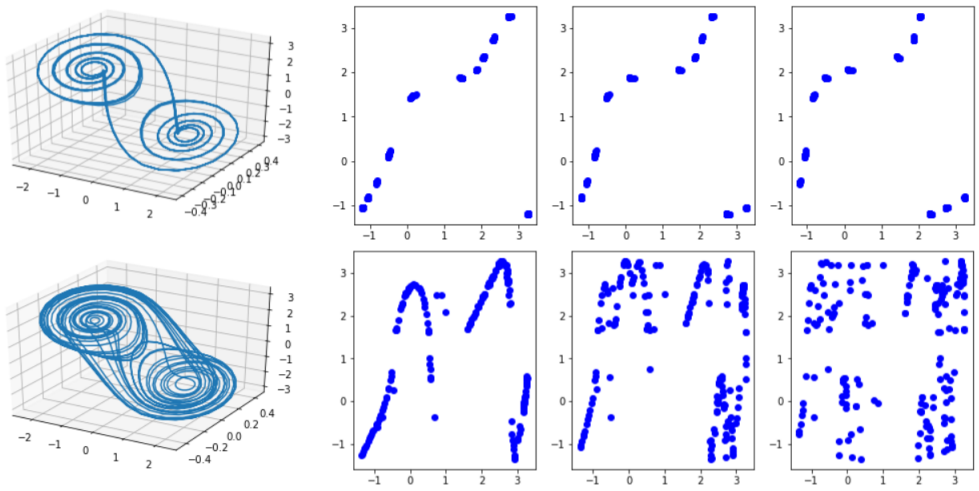


Figure 2.15: 1^{st}, 2^{nd}, and 3^{rd} return maps for $\beta = 16$ (upper) and $\beta = 14$ (lower). The "sparse" attractor is once again reflected in the return maps. Interestingly enough, all three iterated maps are almost identical. For $\beta = 14$, the bimodal plot is prominent, which may be due to the fact there are two joined, single scrolls.

A summary of this chapter is concisely displayed in the bifurcation diagram. Although the system in Figure 2.16 refers to the actual circuit parameters, the overall behavior is consistent.

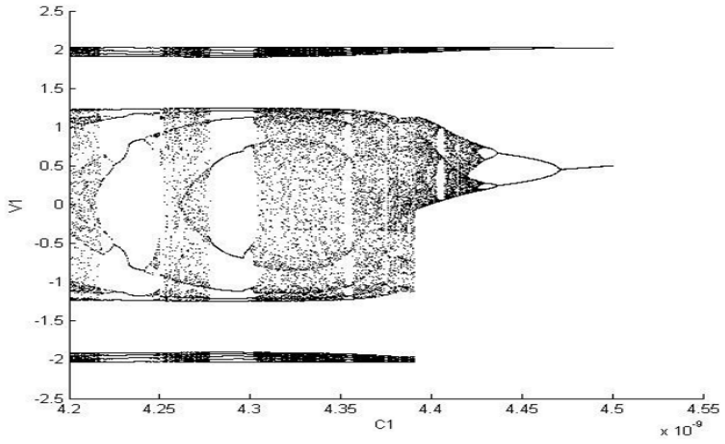


Figure 2.16: A bifurcation diagram of the Chua circuit. Notice that period doubling occurs as the parameter is decreased. However, upon entering chaos in the single scroll, the route to the double scroll is not as clear. Although there are gapse in the chaotic regions, it does not appear to be period-3 and may instead refer to the aforementioned examples of "sparse," less-chaotic attractors. Diagram obtained from `http://nldlab.gatech.edu`.

2.4 The Lyapunov Exponent

Sensitive dependence on initial conditions is a key characteristic of chaos. Consequently, two trajectories that are close together will diverge rapidly. This idea is encapsulated in Figure 2.17.

We can model this relationship as,

$$||\delta(t)|| \approx ||\delta_0|| e^{\lambda t} \tag{2.3}$$

where λ is known as the Lyapunov exponent.

This can then be rewritten as,

$$\ln\left(\frac{\delta(t)}{\delta_0}\right) \approx \lambda t. \tag{2.4}$$

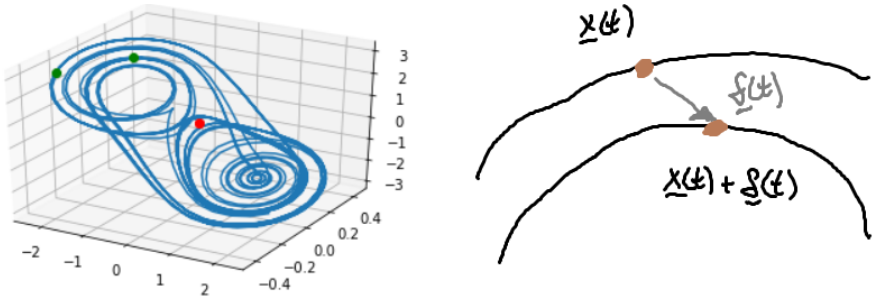


Figure 2.17: An attractor with two initial points in red seperated by a small distance slightly less than $\delta_0 = 0.01$. The final location of this pair is represented by the green points. Note that $\delta(t)$ has increased significantly. The right side depicts a magnified view of the error vector between trajectories.

Therefore, if we plot $\ln(\delta/\delta_0)$ against t, the slope of the line is λ.

Observe that the plot is not perfectly linear since there is some variation when moving along the attractor. Furthermore, δ must eventually level off since the separation between points is limited by the "diameter" of the attractor [5]. The results for the Chua system are plotted in Figure 2.18.

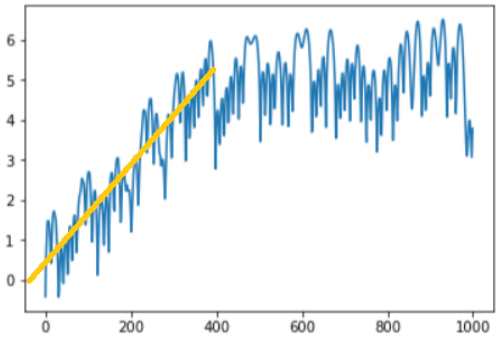


Figure 2.18: A plot of $\ln(\delta/\delta_0)$ versus time steps. The slope of the line represents the Lyapunov exponent λ, which is approximately 0.012. This is not quite the correct value since we did not average over a large sample of points on the same trajectory. However, this simple model still yields a positive exponent, and therefore indicates the exponential divergence of nearby trajectories.

Chapter 3

Experimental Setup

3.1 Circuit Implementation

3.1.1 Analysis of Chua's Circuit

The implementation of Chua's circuit is shown below.

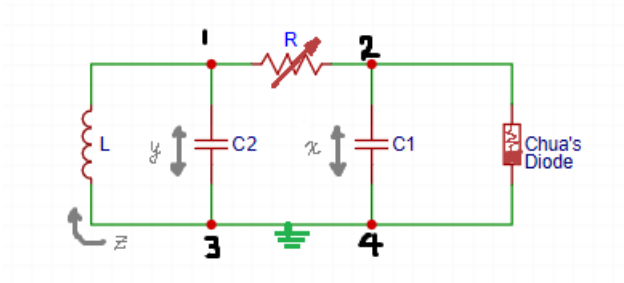


Figure 3.1: A schematic of the Chua circuit. Note that R is a variable resistor, which is needed to vary the parameters of Chua's equations.

It consists of two capacitors, an inductor, a resistor, and a component known as Chua's diode. The specifications are listed in Appendix A. The diode's IV curve is a piecewise function (Figure 3.2), which is necessary for nonlinear behavior to occur [6].

If we let the voltage across C_2 be x, the voltage across C_1 be y, and the current through the inductor be z, then applying KCL for nodes 1 and 2 in Figure 3.1 yields,

$$\frac{x-y}{R} + \frac{dx}{dt}C_1 + g(x) = 0 \tag{3.1}$$

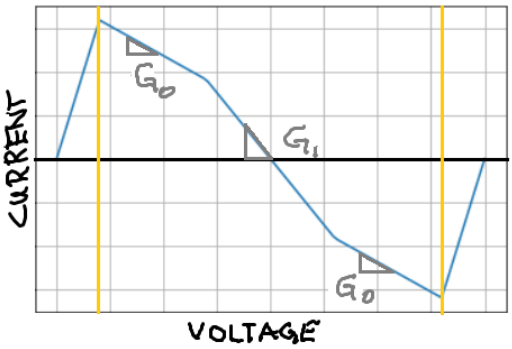


Figure 3.2: IV characteristic of a Chua diode. The circuit operates within the vertical lines, which is a piecewise function with three segments. Each respective slope is denoted by conductances G_0 and G_1.

$$\frac{y-x}{R} + \frac{dy}{dt}C_2 - z = 0. \tag{3.2}$$

Finally, KVL around the loop 1-3-1 gives the third equation,

$$L\frac{dz}{dt} + y = 0. \tag{3.3}$$

Rearranging terms results in the following:

$$\begin{cases} RC_1\dot{x} = y - x - Rg(x) \\ RC_2\dot{y} = x - y + Rz \\ L\dot{z} = -y \end{cases} \tag{3.4}$$

which closely models Equation 2.1.

3.1.2 Constructing a Piecewise IV Curve

In order to create a piecewise IV characteristic, we cascade two negative resistance converters (NRC). The basic unit is shown in Figure 3.3. Using the junction law at node 1:

$$i = \frac{v - v_0}{R_1}. \tag{3.5}$$

Applying the loop law around 1-3-0-1 and recognizing the voltage divider pattern gives,

$$v = v_d + \frac{R_3}{R_2 + R_3}v_0 \tag{3.6}$$

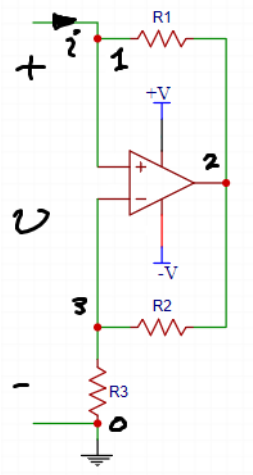


Figure 3.3: Circuit diagram of a negative resistance converter. Notice that the op amp has both positive and negative feedback. Interestingly enough, the NRC draws power into the circuit rather than dissipating energy like a resistor.

Recall the voltage transfer curve of an operational amplifier (Figure 3.4). It can be broken down into three regimes: the negative saturation region, linear region, and positive saturation region. Consequently, we will analyze each separately.

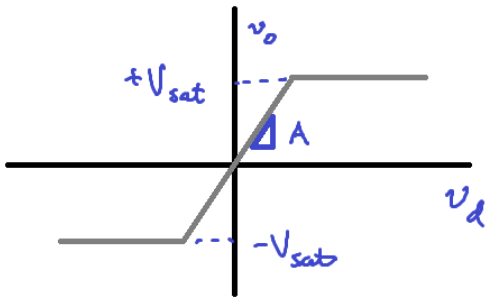


Figure 3.4: Transfer characteristic of an op amp. v_d is the differential voltage and v_0 is the output voltage. Although this curve may be shifted horizontally by some offset V_{OS}, we simply the calculations by assuming it is centered at zero. For a more complete discussion, see [7].

In the linear region, $v_0 = Av_d$. Thus,

$$i = \frac{1}{R_1}(v - Av_d) \tag{3.7}$$

$$v = v_d + \frac{R_3}{R_2 + R_3} A v_d \tag{3.8}$$

from Equations 3.5 and 3.6 respectively.

Combining these two results, one finds that,

$$i = \left[\frac{(1 - A)R_2 + R_3}{R_1[R_2 + (1 + A)R_3]} \right] v, \tag{3.9}$$

and when A is large,

$$i = \left[\frac{-R_2}{R_1 R_3} \right] v. \tag{3.10}$$

In this experiment, $R_1 = R_2$. As a result, $m_1 = -1/R_3$ where $i = m_1 v$. In the positive and negative saturation regions,

$$i = \frac{1}{R_1}(v \pm V_{sat}). \tag{3.11}$$

Therefore, the slope of the IV curve is given by,

$$\frac{di}{dv} = \frac{1}{R_1} = m_0. \tag{3.12}$$

Putting these results together shows a piecewise IV curve (Figure 3.5).

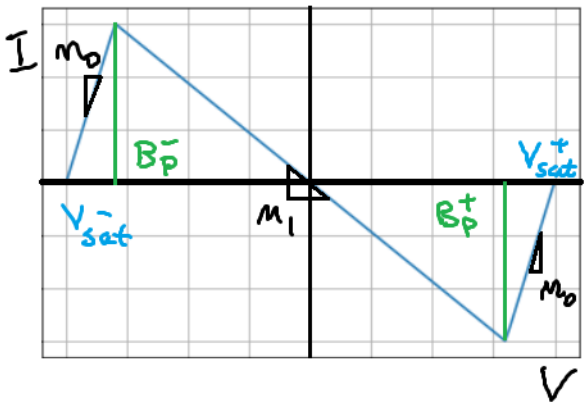


Figure 3.5: The IV characteristic of an NRC. The slope is negative in the linear region of operation, and positive in the saturation regions.

We can find the discontinuities (or voltage breakpoints), denoted as B_p^- and B_p^+, by analyzing the saturation regions. In the positive saturation region, $v_0 = V_{sat}$. Therefore, from Equation 3.5,

$$i = \frac{1}{R_1}(v - V_{sat}). \tag{3.13}$$

Positive saturation is reached when $v_d \geq \frac{V_{sat}}{A}$, and so, substituting into Equation 3.6,

$$v \geq \left[\frac{V_{sat}}{A}\right] + \left[\frac{R_3}{R_2 + R_3} v_0\right] \tag{3.14}$$

This means that

$$v \geq \frac{R_3}{R_2 + R_3} V_{sat} \tag{3.15}$$

as $A \to \infty$. A similar procedure can be performed for the negative saturation region. In summary,

$$B_p = \pm \frac{R_3}{R_2 + R_3} V_{sat}. \tag{3.16}$$

3.1.3 Building Chua's Diode

Consider two NRCs in parallel with varying resistors. Then, the total current passing through will be split as follows: $i_{tot} = i_1 + i_2$. This is equivalent to $i_{tot} = g_1(v) + g_2(v)$, where $g_1(v)$ and $g_2(v)$ are functions of current with respect to voltage. Thus, the resulting IV characteristic can be obtained by simply summing the two individual curves (Figure 3.7).

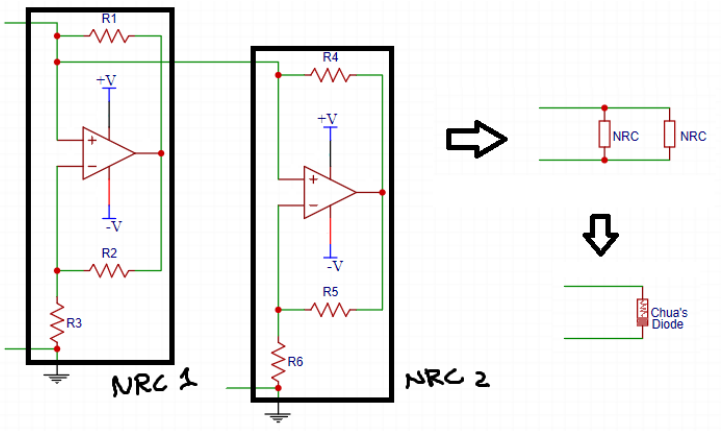


Figure 3.6: The lumped-element abstraction process. The model on the left can be simplified as two NRC "components" connected in parallel. This can be further abstracted into just one element: the Chua diode.

Given appropriate parameters, adding two 3-segment piecewise functions will result in a 5-segment piecewise function with the desired shape depicted previously (Figure 3.2). A summary of this section is illustrated in Figure 3.6 above.

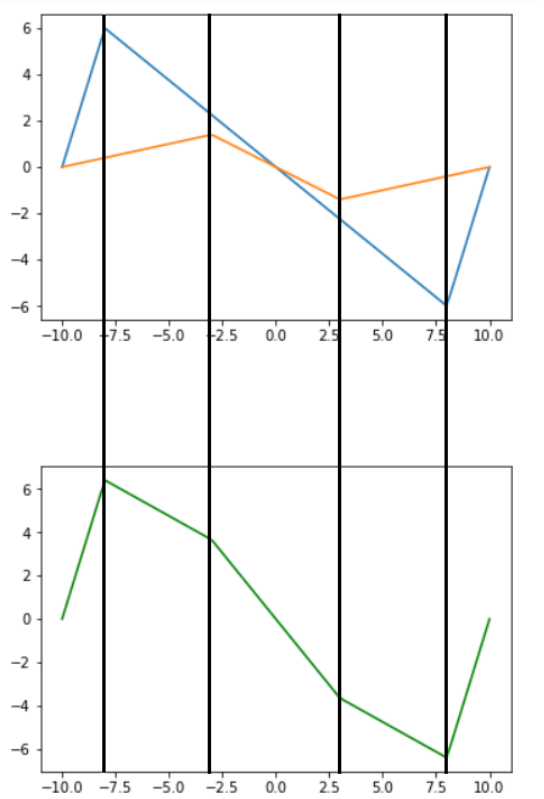


Figure 3.7: Piecewise addition of two NRC IV curves. The blue curve has parameters $m_0 = 3$, $m_1 = -2$, and $B_p = \pm 8$. The orange curve has parameters $m_0 = 0.2$, $m_1 = -0.5$, $B_p = \pm 3$. All units are arbitrary in this diagram.

3.2 Synchronization

3.2.1 The Mathematical Idea

It appears somewhat paradoxical that two chaotic systems can synchronize, given that nearby trajectories exponentially diverge. However, this can be theoretically shown by using a clever choice of coupling.

The Chua equations have been reproduced below:

$$\begin{cases} \dot{x} = \alpha[y - x - g(x)] \\ \dot{y} = x - y + z \\ \dot{z} = -\beta y \end{cases} \tag{3.17}$$

We refer to the previous set of equations as the "transmitter". Next, construct a nearly identical system, replacing x_r with x:

$$\begin{cases} \dot{x_r} = x \\ \dot{y_r} = x - y_r + z_r \\ \dot{z_r} = -\beta y_r \end{cases} \tag{3.18}$$

This result will represent the "receiver". Now, define the following error terms:

$$\begin{cases} e_x = x - x_r \\ e_y = y - y_r \\ e_z = z - z_r \end{cases} \tag{3.19}$$

Since $x = x_r$ by construction, $e_x = 0$. In order for these two systems to fully synchronize, it is necessary to also show that $e_y \to 0$ and $e_z \to 0$ as $t \to \infty$. The time derivatives of the errors are,

$$\begin{cases} \dot{e_y} = -(y - y_r) + (z - z_r) = -e_y + e_z \\ \dot{e_z} = -\beta(y - y_r) = -\beta e_y \end{cases} \tag{3.20}$$

and so,

$$\begin{cases} \dot{e_y} e_y = -e_y^2 + e_z e_y \\ \frac{1}{\beta}\dot{e_z} e_z = -e_y e_z \end{cases} \tag{3.21}$$

Combining these equations yields,

$$\dot{e_y} e_y + \frac{1}{\beta}\dot{e_z} e_z = -e_y^2 + e_z e_y - e_y e_z \tag{3.22}$$

$$\frac{d}{dt}\left(\frac{1}{2}\left[e_y^2 + \frac{1}{\beta}e_z^2\right]\right) = -e_y^2. \tag{3.23}$$

Define $V = e_y^2 + e_z^2/\beta$. Since V is a positive definite function and its time derivative $\dot{V}$ is always negative (except at $\mathbf{e} = \mathbf{0}$), this means that V is a Lyapunov function, and so $\mathbf{e} = \mathbf{0}$ is globally asymptotically stable [5]. For a formal definition and proof see reference [8].

3.2.2 Physical Realization and Proof

Translating the mathematics into a physical system requires the construction of two circuits. One for the transmitter and one for the receiver. The coupling is accomplished by connecting two associated nodes (Figure 3.8).

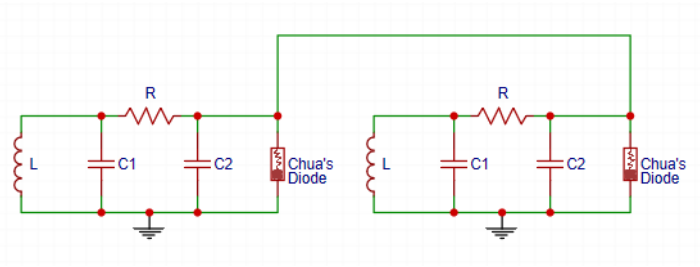


Figure 3.8: Synchronization of two Chua circuits. The two circuits are identical, including the component specifications. This is a bidirectional setup. That is, when the parameters of one circuit are perturbed, the other circuit closely follows and vice-versa.

As a check, one can plot y versus y_r on an oscilloscope. If the two circuits are synchronized, the plot should be a 45° line. Otherwise, chaos will appear.

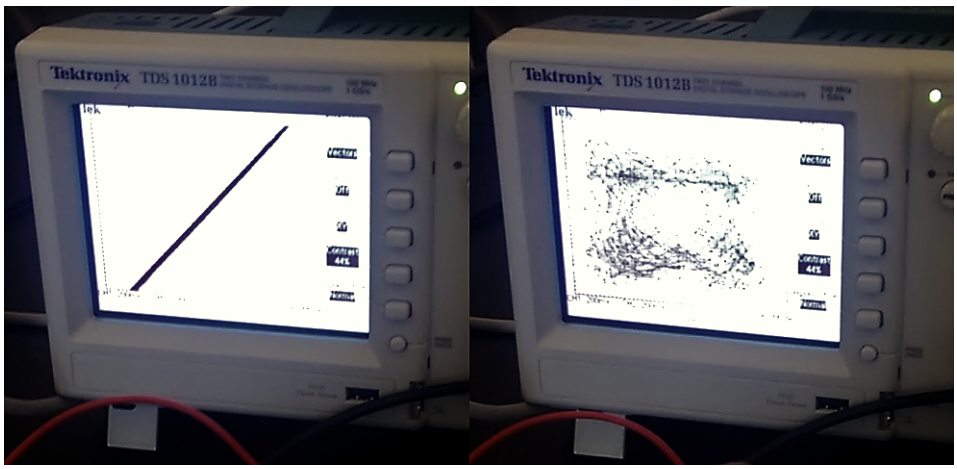


Figure 3.9: Plots of y vs. y_r. The left-hand image shows when the two circuits are synchronized. Even though the points tend to move around, they do so along a straight line. The right-hand side shows when the circuits are not synchronized (the wire between the two circuits is disconnected).

This phenomenon can also be simulated on the computer by adjusting the Runge-Kutta algorithm to implement the combined system described by Equations 3.17 and 3.18. The results are plotted in Figure 3.10 below, and qualitatively agree with the experimental plots (Figure 3.9).

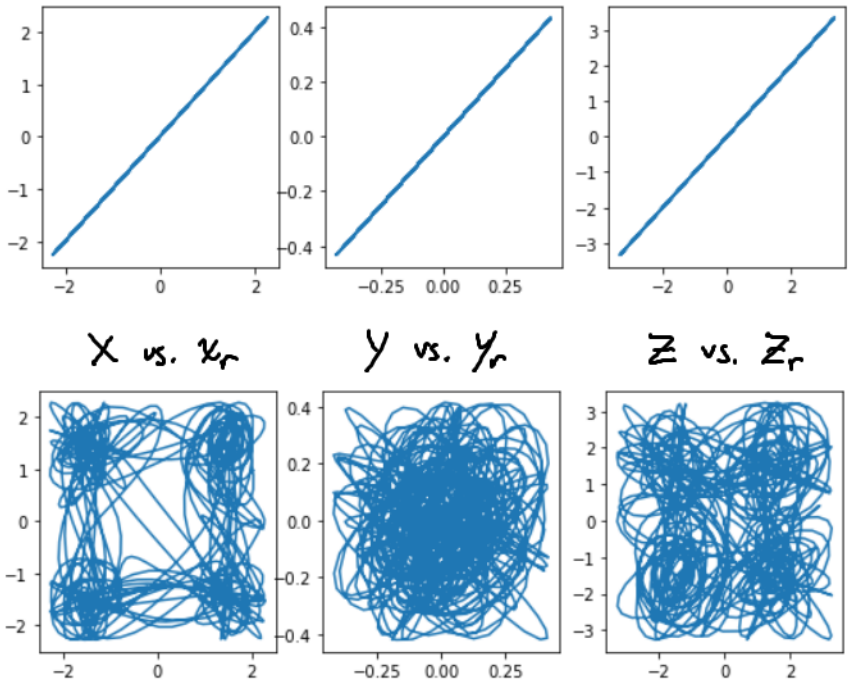


Figure 3.10: Plots of corresponding variables between two Chua systems using numerical integration for 10000 steps. The first 8000 transients are removed. The parameters are as follows: $x_0 = 0.1$, $y_0 = 0.1$, $z = 0.6$, $\alpha = 10.0$, $\beta = 15.0$. The receiver initial conditions are: $y_{r,0} = 0.2$ and $z_{r,0} = 0.8$.

This is remarkable because although only one variable x was "sent", we can reconstruct the entire system and recover the values of y and z as well.

3.2.3 Rate of Convergence

One can show that the rate of convergence of the error function decays exponentially. In order for this to hold, the previous Lyapunov function must satisfy the condition $\dot{V} \leq -kV$ for some $k > 0$ (see Equation 3.23).

From before,

$$V = e_y^2 + \frac{1}{\beta} e_z^2, \tag{3.24}$$

and so

$$\dot{V} = -2e_y^2 \tag{3.25}$$

after properly substituting in Equation 3.20. Consequently, since $\dot{e_z} < 0 \implies e_y > 0$ and $\dot{e_y} < 0 \implies e_y > e_z$,

$$\dot{V} \leq -2e_y^2 + \left(e_y^2 - \frac{1}{\beta} e_z^2\right) \implies -e_y^2 - \frac{1}{\beta} e_z^2, \tag{3.26}$$

and thus, there exists a $k = 1$ such that $\dot{V} \leq -kV$. This means that both $e_y(t) \to 0$ and $e_y(t) \to 0$ exponentially fast.

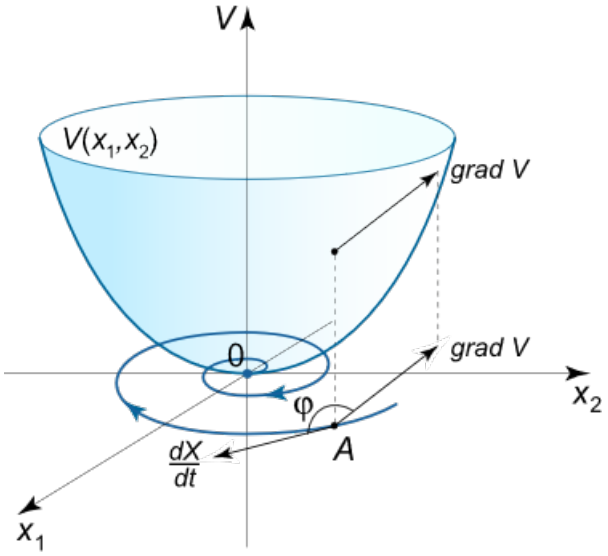


Figure 3.11: Diagram of a Lyapunov function. V is a positive definite surface. If $\dot{V}$ is negative, the phase trajectory spirals towards the origin and the system is stable. The combined rate of decay of the function is limited by the slowest rate along x_1 or x_2. Image credits: `https://www.math24.net/method-lyapunov-functions/`

An exponential rate of convergence is important when distinguishing digital signals. The transmission rate can be increased while maintaining a suitable signal-to-noise ratio. Figure 3.12 contains plots of x, y, and z with respect to time (corresponds to Figure 3.10). Observe that the approximate number of time steps required before synchronization is very short.

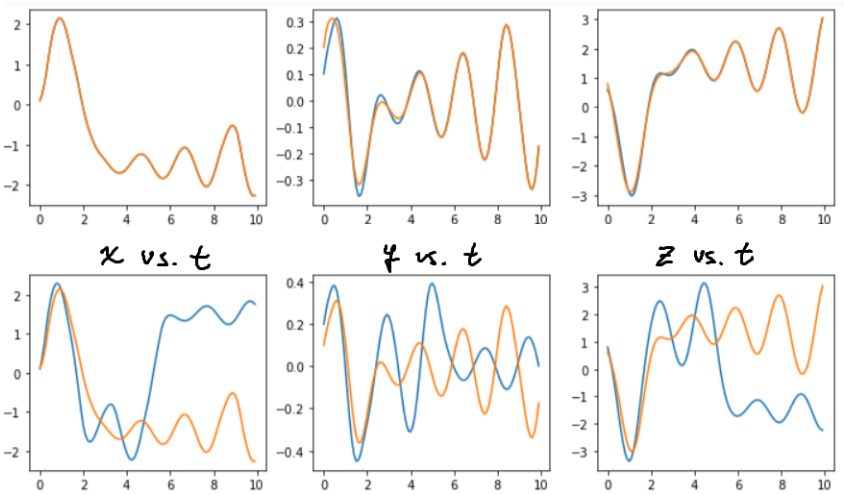


Figure 3.12: Time series plots for the first 100 steps out of 10000. The top shows rapid synchronization of two trajectories with different initial conditions. The bottom row confirms the divergence of trajectories in non-synchronized systems.

3.3 Signal Transmission

3.3.1 Parameter Modulation

By making a slight modification to the connection between the two circuits shown in Figure 3.8, one can create a master/slave configuration (Figure 3.13). In this setup, perturbing the slave circuit distorts the double-scroll attractor of the slave, but the master circuit remains unaffected. However, altering the master circuit will cause the slave circuit to follow. This phenomenon can be seen in Figure 3.14.

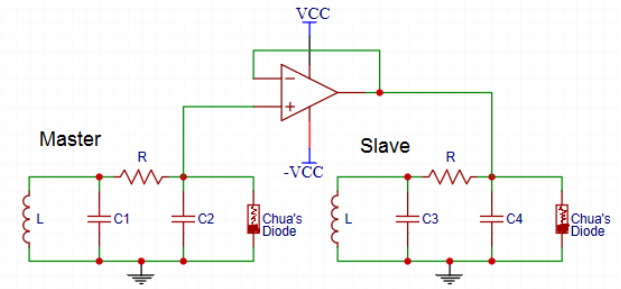


Figure 3.13: Master/slave configuration for circuit synchronization. The only difference is the inclusion of an op amp with negative feedback.

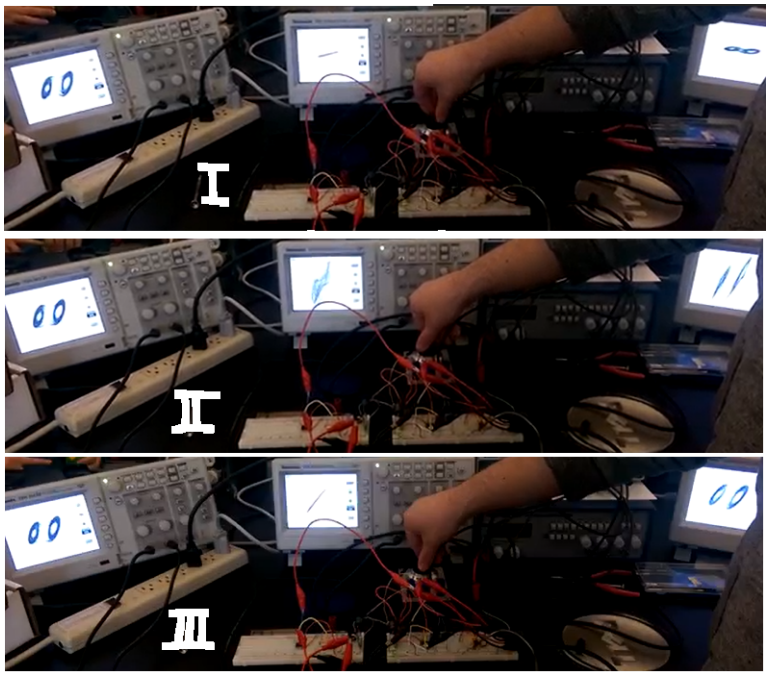


Figure 3.14: Oscilliscope plots of the master circuit xy phase plane (left), slave circuit xy phase plane (right), and synchronization (center). The resistance is varied using the ten-turn potentiometer dial. Notice that the master circuit is unchanged throughout. Furthermore, the slave circuit appears almost identical to the master circuit in III since the synchronization plot is linear at roughly 45°. However, the slave circuit in I and II is clearly distorted.

If we then look at the error e_y, which can be realized by connecting nodes y and y_r to a subtractor (Figure 3.15), a transmission scheme naturally follows. Assume that some parameter varies with time, say $\beta(t)$. Then, when β in both circuits match, the system will synchronize and the error will approach zero. Otherwise, we should expect to see chaos.

This idea of "parameter modulation" is described in multiple references [4,9], and allows one to send a stream of data bits if we consider '1' to be the chaotic region (parameter mismatch), and '0' to be the synchronized region (matching parameters). This procedure is illustrated in Figure 3.16. As a result, the master circuit and subtractor is associated with the receiver, while the slave circuit acts as the transmitter.

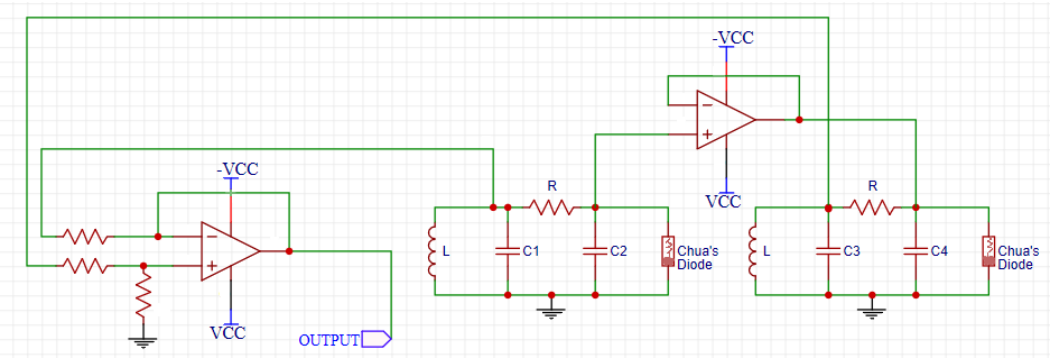


Figure 3.15: The differential op amp is shown to the left of Chua's circuits. Resistances were chosen such that the gain = 1 and the inputs take the difference between y_r and y.

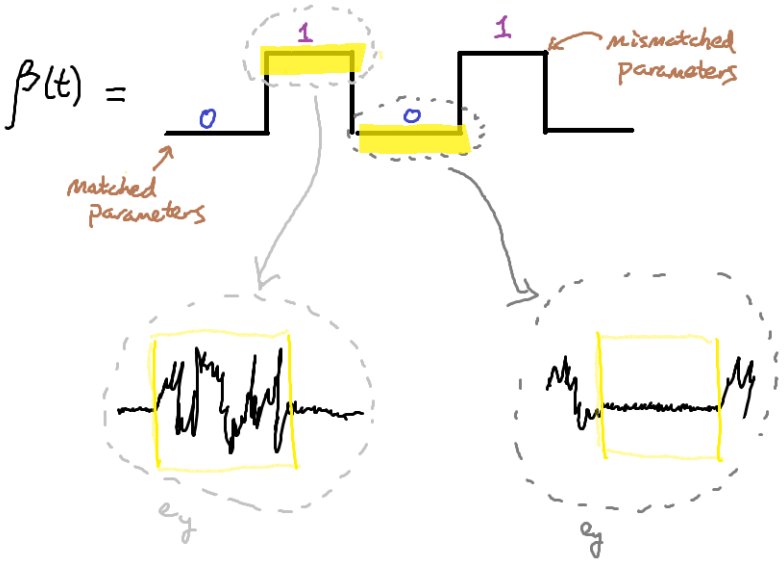


Figure 3.16: A graphical view of parameter modulation. The error is chaotic in regions of parameter mismatch, and zero elsewhere. In reality, there is always some noise since the parameters are never perfectly matched. However, within reasonable tolerances, the error signal can be filtered to retrieve the original bit stream.

3.3.2 A Digital Alternative

Unfortunately, using a mechanical device to manually adjust the resistance is impractical and imprecise. To resolve this issue, we use a digital potentiometer,

commonly shortened to "digipot". This component uses a Serial Peripheral Interface (SPI) bus. Thus, it relies on a short distance communication system to send data between the device and a microcontroller.

In this experiment, we use a Raspberry Pi to talk with the digipot. The circuit schematic is once again modified in Figure 3.17.

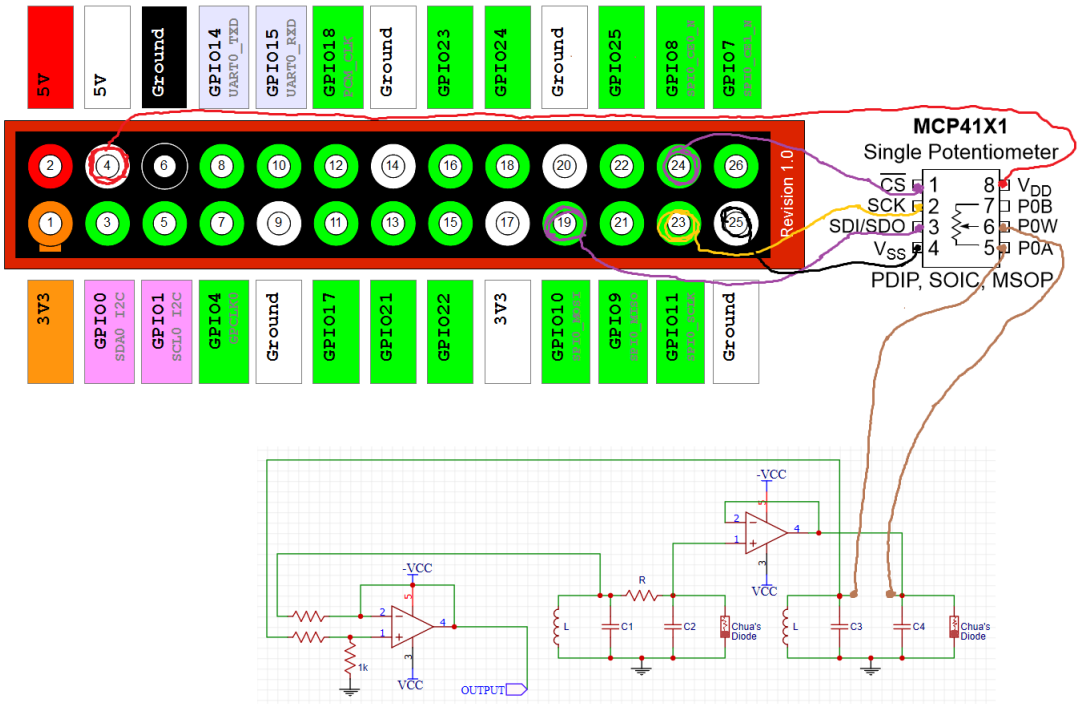


Figure 3.17: A graphical view of parameter modulation. The error is chaotic in regions of parameter mismatch and zero elsewhere. In reality, there is some noise since the parameters are never perfectly matched. However, within reasonable tolerances, the error signal can be filtered to retrieve the original bit stream. The top two images are from Raspberry Pi's website and Microchip's data sheet. Additional details on these devices can be found in references [10] and [11].

This chip allows us to rapidly switch the parameter value R, and control the rate and length of signals programmatically. An audio wire is also attached to the subtractor so that the waveform can be analyzed later. Lastly, this concludes the setup needed for a private chaotic communications system. A few photos of the completed build is shown in Figure 3.18. In the following chapter, we analyze the results of the transmission.

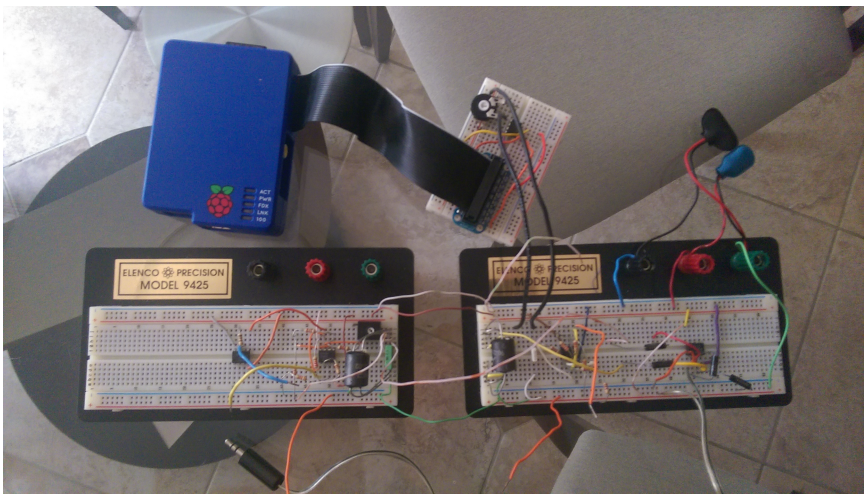


Figure 3.18: The actual completed circuit. Two Chua circuits are mirrored with the master on the left and the slave on the right. The op amp coupling is on the far left while the subtractor was built on the far right. Although the subtractor would normally be built with the master, this network is topologically equivalent and more visually symmetric (appealing).

Chapter 4

Discussion of Results

4.1 Signal Processing

4.1.1 Continuous Bit Stream

In this section and subsequent ones, we attempt to transmit and receive the first two stanzas of a poem. An excerpt is reproduced below:

I Wandered Lonely as a Cloud

I wandered lonely as a cloud
That floats on high o'er vales and hills,
When all at once I saw a crowd,
A host of golden daffodils;
Beside the lake, beneath the trees,
Fluttering and dancing in the breeze.

Continuous as the stars that shine
and twinkle on the Milky Way,
They stretched in never-ending line
along the margin of a bay:
Ten thousand saw I at a glance,
tossing their heads in sprightly dance.

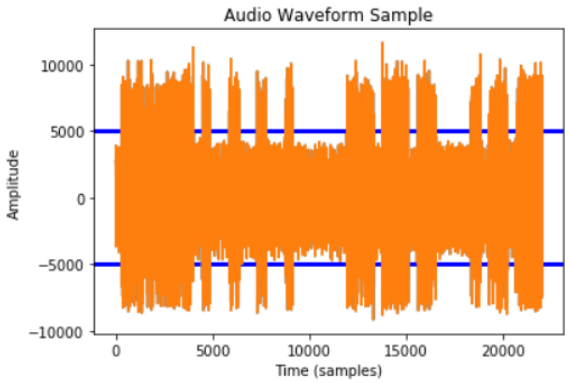
William Wordsworth (1770–1850)

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

Figure 4.1: The American Standard Code for Information Interchange (ASCII) encoding table. Values 32-126 correspond to "printable" characters. The "space" character is considered as a printable character rather than a control character such as "NULL" (00) or a line feed "\n" (10). Figure from `https://www.cs.cmu.edu/ pattis/ 15-1XX/common/handouts/ascii.html`.

First, each character is represented as a 7-digit binary number according to ASCII encoding (Figure 4.1). For instance, the letter "I" has an ASCII value of 73, which is then converted to '1001001'.

Each character's bit representation is then concatenated together into a single message string. Then, upon encountering a '1', the program adjusts the digipot value so that the circuits are out-of-sync, and for each '0', the parameters are readjusted to match. The string '1111111' was attached to the beginning and end of the message to indicate where the program should start and stop analyzing. Each bit was held for 0.01 seconds before the next bit was transmitted.




```
Sampling Rate: 44100 samples/sec
Samples: 1843200
Audio Length: 41.795918367346935 sec
Bit Transmission Rate: 0.01 sec
Samples per Bit: 441
```

Figure 4.2: A sample of the received audio waveform. There is a string of '1's marking the start of the transmission, followed by the message data itself. The blue bar denotes the threshold, and is set at an amplitude of 5000. This value was chosen because it safely excludes '0' values from being accidentally labeled as '1', while still retaining enough high amplitude points to detect a '1' region.

The received audio waveform is shown in Figure 4.2. Note that the digipot introduced a significant amount of noise. Thus, to clean the data, a threshold value was determined. Any values within the cutoff were set to 0, while values that exceeded this range were set to 1. The result is a binary digitized waveform, which may be seen in Figure 4.3 (left).

Although the reported sampling rate is 44100 samples/sec and the bit transmission rate was set to 0.01 sec, there is clearly some deviation that may be due to imprecision in the electronics. Consequently, we tried to correct this situation by altering the samples-per-bit value in the program calculations. The right-hand side of Figure 4.3 shows an improvement in the alignment, but this is short-lived. By the tenth character ('e'), it is evident that the timing is misaligned again (Figure 4.4).

Additional corrections included slowing down the transmission rate by a factor of 10. Unfortunately, similar desyncrhonization is observed. A linear shift was also applied after every 8 characters to account for the modified window sizes. This alleviated the problem for another 80 characters or so before abruptly becoming out-of-sync. Even though one could continue playing around with shifts for different regions of the transmission, this does not generalize well and requires tuning an excessive number of hyperparameters each time a different message is sent. Hence, the model was scrapped.

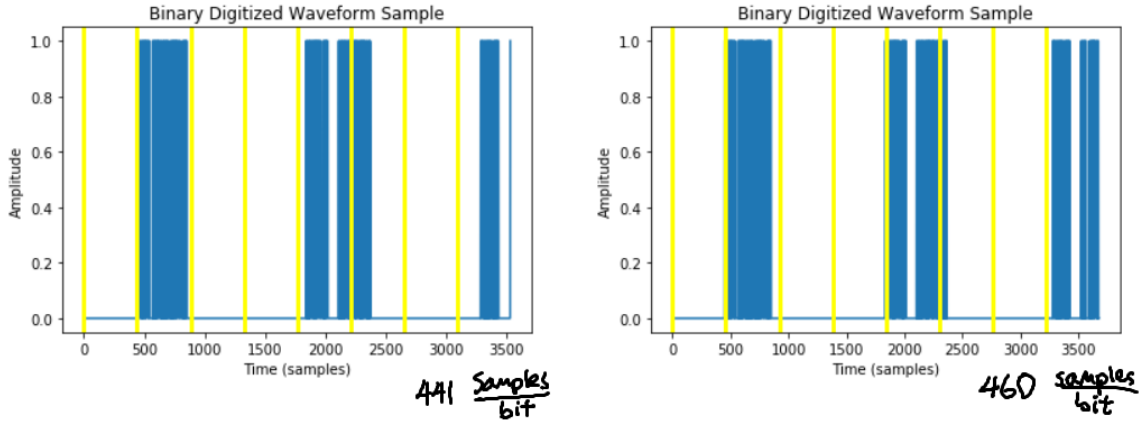


Figure 4.3: Two samples of the digitized audio waveform for the character 'I'. An additional '0' bit was prepended to the waveform in order to (unnecessarily) create a byte. The yellow vertical lines mark off each of the 8 bits. In the left plot, each bit consists of 441 samples. In the right plot, the "window" width is widened to 460 samples per bit. Notice that the right plot better represents the character's binary representation: '01001001'.

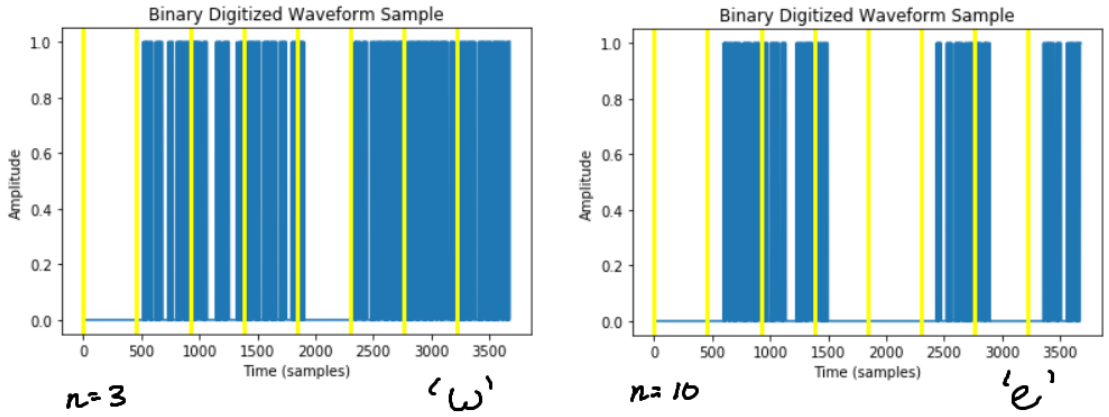


Figure 4.4: Two additional later samples of the digitized audio waveform with the adjusted samples-per-bit value. The left sample correctly represents the lowercase letter 'w', which has a binary representation of '01110111'. The program also correctly identifies the right sample as an 'e', which is '01100101' in binary. Yet, it is gradually going out-of-phase. After the twelfth character, the estimated message bears virtually no resemblance to the original transmission.

4.1.2 Gapped Bit Stream

To fix this issue, a long gap was introduced between character bytes. Then, a '1' bit was prepended to each byte in order for the program to detect the start of each byte. This modification is detailed in Figure 4.5.

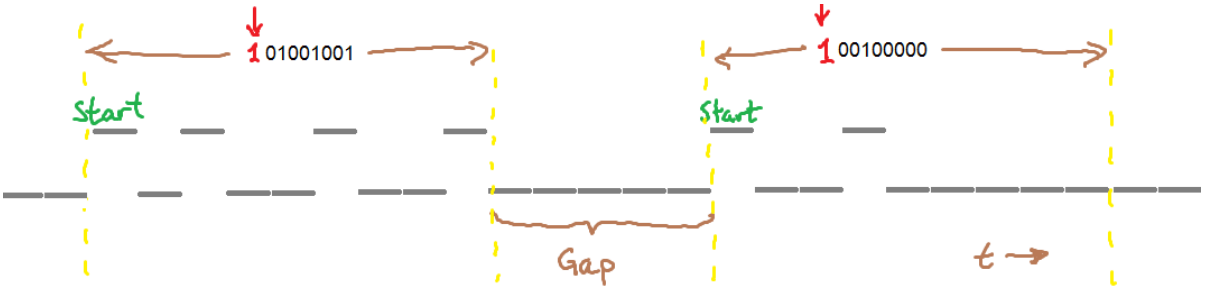


Figure 4.5: Introduction of a gap between characters. In the following setup, the bit transmission rate was set at 0.02 sec and the length of the gap was equal to 5 times the transmission rate, or 0.1 sec. Since the message contained 406 characters, the total length of transmission is approximately 105 sec.

The processing procedure using this layout is outlined below:

1. Scan for a '1'.
3. "Read" off the following byte using an adjusted samples-per-bit value.
3. Convert the binary string to an appropriate character.
4. Append the character to the message and repeat.

This assumes that the system is reset to a '0' state after the transmission of each byte. By using this scheme, many of the desyncronization concerns are resolved. Since the program simply scans for a starting point and then measures out 8 bits from that location, this ensures that slight deviations in timing will not accumulate. Of course, this relies on the observation that timing errors within each byte remain relatively stable.

The result is that one can recover the original message with remarkable accuracy. This is affirmed by looking at the recovered poem printed in Figure 4.6.

4.2 Incorporating Error-Correction

This section assumes some familiarity with elementary coding theory. Thus, we have reviewed key selected topics in Appendix A.

```
I wandered lonely as a cloud
That floats on high o'er vales and hills,
A host of golden daffodils;
Beside the lake, beneath the trees,
Fluttering and dancing in the breeze.

Continuous as the stars that shine
and twinkle on the Milky Way,
They stretched in never-ending line
along the margin of a bay:
Ten thousand saw I at a glance,
tossing their heads in sprightly dance.
```

Figure 4.6: The estimated message using a gapped bit stream transmission. There are two errors highlighted above. This corresponds to an error rate of 2/406 characters, or about 0.5%.

4.2.1 Selecting the Code

Since each character can be encoded in 7 bits, the dimension is $k = 7$ and the length of the code n must therefore be greater than 7. After testing several possibilities, we settle on $n = 15$. Since $\dim(\mathcal{C}) = k = n - \deg g(x)$ where $g(x)$ is the generator polynomial, the degree of our generator must be 8.

Given that the factorization of $x^{15} - 1$ into irreducible polynomials over $\mathbb{F}_2$ is

$$(x+1)(x^2+x+1)(x^4+x+1)(x^4+x^3+1)(x^4+x^3+x^2+x+1), \tag{4.1}$$

we can choose our generator to be

$$g(x) = (x^4+x+1)(x^4+x^3+x^2+x+1) = x^8+x^7+x^6+x^4+1. \tag{4.2}$$

This corresponds to the following generator matrix,

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

,

which may be written in standard form as

$$G' = \left[\begin{array}{ccccccc|cccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{array}\right]$$

,

4.2.2 Encoding Procedure

4.2.3 Meggitt Decoding

4.2.4 Comparisons

Chapter 5

Conclusions

Synchronized chaos is a novel way of adding security to private communications. Furthermore, by incorporating an error-correcting code, almost all errors fixed. With that said, it is important to note that this method is not completely secure. For instance, if someone were to build an identical circuit, "tapped" into the wire and matched all parameters sufficiently, then the intruder would be able to extract the contents of the message. In our implementation, this would be relatively easy considering there is only one parameter to tune. However, assuming the remaining capacitors and resistors were also varied, then the level of complexity is increased to that of a locker combination at minimum.

Additional work to this particular study may include building an additional circuit to analyze the difficulty of matching parameters, and possible techniques for finding these parameters faster. Different encoding schemes may be compared to see which provides a lower error rate. Finally, one can attempt to synchronize the system using other variables.

Methods to increase security include randomizing the '1' state's digipot value, or by using an encryption scheme before sending. However, this does come at the cost of increased memory and lengthier transmission times. Although using circuits is not particularly practical due to the fact that long wires are needed for synchronization and signal subtraction, the concept of using synchronized chaos is seeing greater promise in the field of optics. New communication schemes are being implemented chaotic lasers and fiber optics, which promise to resolve many of the concerns previously discussed [12].

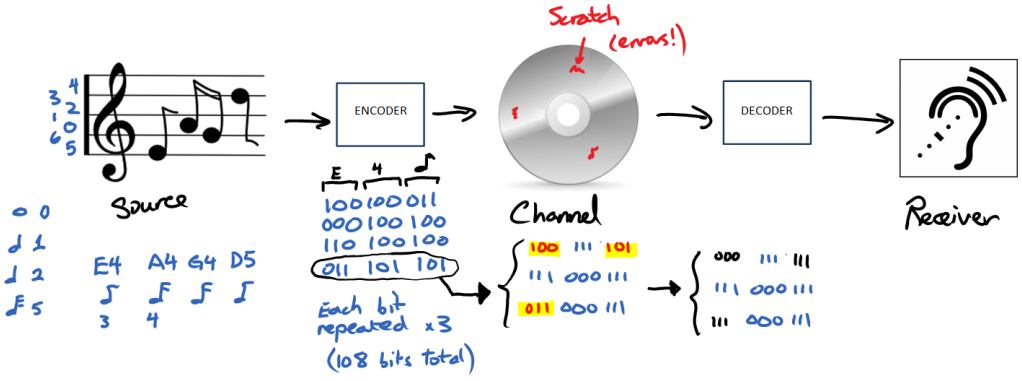
Appendix A

Background: Coding Theory

A.1 Introduction

Key features of a communication channel are illustrated below. The source emanates information, which is then encoded in a message and sent over a channel. Since the channel may be "noisy", we must decode the message by correcting errors, removing redundancies, and forming an *estimate* of the original message.

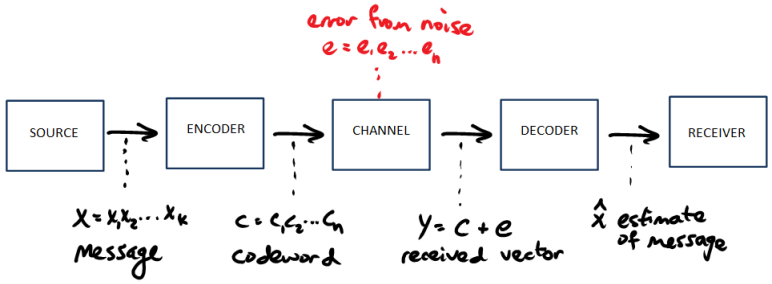
Figure A.1: Communication Channel



More formally, **codeword symbols** will come from a field $\mathbb{F}_q$ with q elements. **Messages** and **codewords** will be vectors in vector spaces $\mathbb{F}_q^k$ and $\mathbb{F}_q^n$ respectively. The relationship between these elements are shown in Figure 1.2:

Note that the error $\mathbf{e}$ can also be written as $\mathbf{y}-\mathbf{c}$. We would certainly hope that $\hat{x} = x$, and **Shannon's Theorem** guarantees that there exists an encoding so that

Figure A.2: Mathematical Model



this happens at an arbitrarily high percentage, though not 100%. Consequently, one of the goals of studying coding theory is to generate codes that satisfy this theorem. Once the code is chosen, encoding is typically straightforward. However, decoding is often more complex.

A.2 Three Fields

We will study linear codes the most since they are easier to describe, encode, and decode. Our alphabet is a field $\mathbb{F}_q$, also written as $GF(q)$, with q elements.

A **field** is an algebraic structure with two operations usually called addition $(+)$ and multiplication $(\cdot)$. Three common fields are the:

- *binary* field: two elements $\{0, 1\}$; ring of intergers modulo 2
- *ternary* field: three elements $\{0, 1, 2\}$; ring of integers modulo 3
- *quaternary* field: four elements $\{0, 1, \omega, \bar{\omega}\}$; this is NOT the ring $\mathbb{Z}_4$.

The quaternary field is defined by the following Cayley tables:

$+$	0	1	ω	$\bar{\omega}$
0	0	1	ω	$\bar{\omega}$
1	1	0	$\bar{\omega}$	ω
ω	ω	$\bar{\omega}$	0	1
$\bar{\omega}$	$\bar{\omega}$	ω	1	0

$\cdot$	0	1	ω	$\bar{\omega}$
0	0	0	0	0
1	0	1	ω	$\bar{\omega}$
ω	0	ω	$\bar{\omega}$	1
$\bar{\omega}$	0	$\bar{\omega}$	1	ω

Note that $x + x = 0$ for all $x \in \mathbb{F}_4$. Also $\bar{\omega} = \omega^2 = 1 + \omega$ and $\omega^3 = \bar{\omega}^3 = 1$.

A.3 Linear Codes

$\mathbb{F}_q^n$ is the vector space of n-tuples over the finite field $\mathbb{F}_q$.

Example: $\mathbb{F}_2^3$ includes (0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), and (1, 1, 1)

An (n, M) **code** $\mathcal{C}$ over $\mathbb{F}_q$ is a subset of $\mathbb{F}_q^n$ of size M.

Example: A (3, 4) code over $\mathbb{F}_2$ may be {(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1)}.

In general, we write the vectors $(a_1, a_2, ..., a_n)$ in $\mathbb{F}_q^n$ as $a_1a_2...a_n$ and call the vectors in $\mathcal{C}$ **codewords**. If $\mathcal{C}$ is a k-dimensional subspace of $\mathbb{F}_q^n$, then $\mathcal{C}$ will be called an [n, k] **linear code** over $\mathbb{F}_q$. The linear code has q^k codewords.

Example: A [3, 1] linear code over $\mathbb{F}_2$ might be {(0, 0, 0), (1, 1, 1)}
A [3, 2] linear code over $\mathbb{F}_2$ might be {(0, 0, 0), (0, 1, 1), (0, 1, 0), (0, 0, 1)}

Two common ways to present a linear code are with a generator matrix or a parity check matrix. A **generator matrix** for an $[n, k]$ code $\mathcal{C}$ is a $k \times n$ matrix G whose rows form a basis for $\mathcal{C}$. It is also defined by the following:

$$\mathcal{C} = \{\mathbf{x}G \,|\, \mathbf{x} \in \mathbb{F}_q^k\} \tag{A.1}$$

Any set of k linearly independent columns is called an **information set** for $\mathcal{C}$. The remaining $r = n - k$ columns form the **redundancy set** and r is called the **redundancy** $\mathcal{C}$. These are illustrated in Figure 1.3.

If the *first* k columns are linearly independent, then there exists a unique generator matrix of the form $[I_k|A]$ where I_k is the $k \times k$ identity matrix. Such a matrix is said to be in **standard form**. An example of a generator matrix for a linear [3, 2] code in standard form is,

$$G = \left[\begin{array}{cc|c} 1 & 0 & 1 \\ 0 & 1 & 1 \end{array}\right]$$

The first two columns form an information set and the last coordinate forms the redundancy set. We can generate codewords from messages in the following manner:

$$\begin{bmatrix} 0 & 0 \end{bmatrix} \left[\begin{array}{cc|c} 1 & 0 & 1 \\ 0 & 1 & 1 \end{array}\right] = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

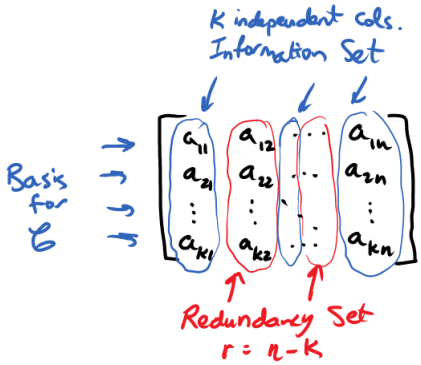


Figure A.3: Depiction of an information set and redundancy set for an arbirary generator matrix.

$$\begin{bmatrix} 0 & 1 \end{bmatrix} \left[\begin{array}{cc|c} 1 & 0 & 1 \\ 0 & 1 & 1 \end{array}\right] = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \left[\begin{array}{cc|c} 1 & 0 & 1 \\ 0 & 1 & 1 \end{array}\right] = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \left[\begin{array}{cc|c} 1 & 0 & 1 \\ 0 & 1 & 1 \end{array}\right] = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}$$

Recall that every subspace is the kernel of a linear transformation between vector spaces. In particular, there exists a **parity check matrix** H for the $[n, k]$ linear code $\mathcal{C}$ defined by:

$$\mathcal{C} = \{\mathbf{x} \in \mathbb{F}_q^n \mid H\mathbf{x}^T = \mathbf{0}\} \tag{A.2}$$

Theorem 1.2.1 (pg. 4):
If $G = [I_k|A]$ is a generator matrix for the $[n, k]$ code $\mathcal{C}$, then $H = [-A^T|I_{n-k}]$ is a parity check matrix for $\mathcal{C}$.

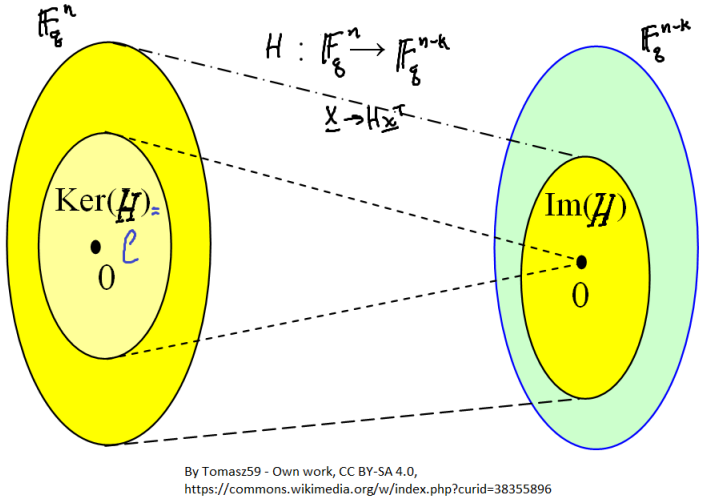
Proof. $HG^T = \left[-A^T_{(n-k)\times k} | I_{n-k} \right] \left[\dfrac{I_k}{A^T_{(n-k)\times k}} \right] = [-A^T + A^T] = 0$. Therefore, $\mathcal{C}$ is *contained* in the kernel of the linear map $\mathbf{x} \to H\mathbf{x}^T$. Lastly, this map has a kernel of dimension k (shown in Exercise 1), which is indeed equal to the dimension of $\mathcal{C}$. Hence, $\mathcal{C}$ is the kernel of the map.

□

Exercise 1 (pg. 4):

Proof. The map $\mathbf{x} \to H\mathbf{x}^T$ is a linear transformation from $\mathbb{F}_q^n$ to $\mathbb{F}_q^{n-k}$ with kernel $\mathcal{C}$ (as illustrated in Figure 1.4). By the rank-nullity theorem, $n = dim(\mathbb{F}_q^n) = dim(\mathcal{C}) + dim(H\mathbf{x}^T)$. Since $dim(\mathcal{C}) = k$, $dim(H\mathbf{x}^T) = n-k$. Therefore, $rank(H) = n-k$ and so the rows of H must be linearly independent since H is an $(n-k) \times n$ matrix. □

Figure A.4:



In our previous example, the parity check matrix would be

$$H = \left[\begin{array}{cc|c} 1 & 1 & 1 \end{array}\right]$$

Notice that if the characteristic of the field is 2 (i.e. $1 + 1 = 0$) as in binary codes, $-A = A$. Given a codeword $\mathbf{x} = [1, 0, 1]$, the syndrome is

$$H\mathbf{x}^T = \left[\begin{array}{cc|c} 1 & 1 & 1 \end{array}\right] \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \end{bmatrix}$$

which is null if and only if the codeword is valid. For example, $\mathbf{x}_{err} = [0, 0, 1]$ results in

$$H\mathbf{x}_{err}^T = \left[\begin{array}{cc|c} 1 & 1 & 1 \end{array}\right] \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \end{bmatrix}$$

A simple encoding scheme is the [n, 1] **binary repetition code**. For $n = 7$, our codewords are $\mathbf{0} = 0000000$ and $\mathbf{1} = 1111111$. If up to three errors are made during transmission, we can decode by using a "majority vote". In general, this code can correct up to $e = \lfloor (n-1)/2 \rfloor$ errors.

Exercise 2 (pg. 5):
A generator matrix for the binary repetition code is:

$$G = \left[\begin{array}{c|ccc} 1 & 1 & \dots & 1 \end{array}\right]$$

Therefore, there are n information sets {column 1}, {column 2}, . . . , {column n} for this code.

Another example is the [7, 4] **Hamming code**, denoted $\mathcal{H}_3$. The matrix $G = [I_4|A]$ where

$$G = \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right]$$

is the generator matrix in standard form. The parity check matrix is,

$$H = [A^T|I_3] = \left[\begin{array}{cccc|ccc} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

Exercise 3 (pg. 5):
The following are information sets of G since these columns are linearly independent:
$\{G_{*1}, G_{*2}, G_{*3}, G_{*4}\}$, $\{G_{*1}, G_{*2}, G_{*3}, G_{*6}\}$, $\{G_{*1}, G_{*2}, G_{*3}, G_{*7}\}$, $\{G_{*1}, G_{*2}, G_{*3}, G_{*5}\}$

However, $\{G_{*2}, G_{*3}, G_{*4}, G_{*5}\}$ is not an information set since:

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Hence, the vectors are linearly dependent and form a redundancy set.

A.4 Weights and Distances

Theorem 1.4.1 (pg. 8): The distance function $d(\mathbf{x}, \mathbf{y})$ satisfies the following four properties:

(i) (non-negativity) $d(\mathbf{x}, \mathbf{y}) \geq 0$ for all $\mathbf{x}, \mathbf{y} \in \mathbb{F}_q^n$.

(ii) $d(\mathbf{x}, \mathbf{y}) = 0$ if and only if $\mathbf{x} = \mathbf{y}$.

(iii) (symmetry) $d(\mathbf{x}, \mathbf{y}) = 0$ if and only if **(y, x)** for all $\mathbf{x}, \mathbf{y} \in \mathbb{F}_q^n$.

(iv) (triangle inequality) $d(\mathbf{x}, \mathbf{z}) \leq d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z})$ for all $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{F}_q^n$.

The minimum distance of a code is the smallest distance between distinct codewords. The (Hamming) weight of a vector $\mathbf{x} \in \mathbb{F}_q^n$ denoted $wt(\mathbf{x})$ refers to the number of nonzero coordinates in $\mathbf{x}$.

Theorem 1.4.2 (Pg. 8):

If $\mathbf{x}, \mathbf{y} \in \mathbb{F}_q^n$, then $d(\mathbf{x}, \mathbf{y}) = wt(\mathbf{x} - \mathbf{y})$. If $\mathcal{C}$ is a linear code, the minimum distance d is the same as the minimum weight of the nonzero codewords of $\mathcal{C}$.

Theorem 1.4.13 (Pg. 12):

Let $\mathcal{C}$ be a linear code with parity check matrix H. If $\mathbf{C} \in \mathcal{C}$, the columns of H corresponding to the nonzero coordinates of $\mathbf{c}$ are linearly dependent. Conversely, if a linear dependence relation with nonzero coefficicients exists among w columns of H, then there is a codeword in $\mathcal{C}$ of weight w whose nonzero coordinates correspond to these columns.

Corollary 1.4.14 (Pg. 13):

A linear code has minimum weight d if and only if its parity check matrix has a set of d linearly dependent columns but no set of $d-1$ linearly dependent columns.

Theorem 1.4.15 (Pg. 13):

If $\mathcal{C}$ is an $[n, k, d]$ code, then every $n - d + 1$ coordinate position contains an information set. Furthermore, d is the largest number with this property.

Appendix B

Source Files: Attractor Analysis

B.1 Fourth-Order Runge-Kutta

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import math

def f(x, y, a):
    return a * (y - x - piecewise(x))

def piecewise(x):
    m0 = -8/7
    m1 = -5/7
    if x < -1:
        return m1 * (x + 1) - m0
    elif x >= -1 and x <= 1:
        return m0 * x
    return m1 * (x - 1) + m0

def g(x, y, z):
    return x - y + z

def h(y, b):
    return -b * y

def runge_kutta(x_init, y_init, z_init, num_iter, a, b):
    x = x_init
    y = y_init
    z = z_init
```

```
    dt = 0.1
    iterations = num_iter
    t = np.linspace(0, dt * iterations, iterations)

    X = np.empty((iterations,))
    Y = np.empty((iterations,))
    Z = np.empty((iterations,))

    X[0] = x
    Y[0] = y
    Z[0] = z

    for i in range(1, iterations):
        k_x1 = f(x, y, a) * dt
        k_y1 = g(x, y, z) * dt
        k_z1 = h(y, b) * dt

        k_x2 = f(x + k_x1/2, y + k_y1/2, a) * dt
        k_y2 = g(x + k_x1/2, y + k_y1/2, z + k_z1/2) * dt
        k_z2 = h(y + k_y1/2, b) * dt

        k_x3 = f(x + k_x2/2, y + k_y2/2, a) * dt
        k_y3 = g(x + k_x2/2, y + k_y2/2, z + k_z2/2) * dt
        k_z3 = h(y + k_y2/2, b) * dt

        k_x4 = f(x + k_x3, y + k_y3, a) * dt
        k_y4 = g(x + k_x3, y + k_y3, z + k_z3) * dt
        k_z4 = h(y + k_y3, b) * dt

        x_0 = x
        y_0 = y
        z_0 = z

        x = x_0 + (1/6) * (k_x1 + 2*k_x2 + 2*k_x3 + k_x4)
        y = y_0 + (1/6) * (k_y1 + 2*k_y2 + 2*k_y3 + k_y4)
        z = z_0 + (1/6) * (k_z1 + 2*k_z2 + 2*k_z3 + k_z4)

        X[i] = x
        Y[i] = y
        Z[i] = z

    return {'x': X, 'y': Y, 'z': Z, 't': t}
```

B.2 Return Maps

```
from collections import deque
vals = runge_kutta(.1, .1, .6, 30000, 10.0, 14)
cutoff = 25000

peaks = []
for i in range(1, len(vals['t']) - cutoff - 1):
    before = vals['y'][cutoff + i-1]
    current = vals['y'][cutoff + i]
    after = vals['y'][cutoff + i+1]

    if current > before and current > after:
        peaks.append(current)

return_map = plt.figure(figsize=(15,4))
shifted_peaks_deque = deque(peaks)

#p-th iterated maps
for i in range(1, 5):
    shifted_peaks_deque.rotate(-1)
    shifted_peaks = list(shifted_peaks_deque)
    axis = return_map.add_subplot(1, 4, i)
    axis.plot(peaks[:-i], shifted_peaks[:-i], 'bo')
```

B.3 Poincaré Sections

```
def poincare_section(b):
    y_section = []
    z_section = []
    vals = runge_kutta(.1, .1, .6, 20000, 10.0, b)
    for i in range(10000, len(vals['t']) - 1):
        current = vals['x'][i]
        after = vals['x'][i+1]

        if current < 1 and after > 1:
            y_section.append(vals['y'][i])
            z_section.append(vals['z'][i])

    return {'y': y_section, 'z': z_section}
```

B.4 Lyapunov Exponent

```
vals = runge_kutta(.1, .1, .6, 5000000, 10.0, 14)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
cutoff = 500000 #remove transient points
pt = 1000000
ax.plot(vals['x'][-1000:], vals['y'][-1000:], vals['z'][-1000:])
ax.plot([vals['x'][pt]], [vals['y'][pt]], [vals['z'][pt]], 'ro')

x_a = vals['x'][pt]
y_a = vals['y'][pt]
z_a = vals['z'][pt]

d_0 = 1E-2
start = 0
offset = 10000
#find nearby point
for i in range(pt + offset, len(vals['t'])):
    x_b = vals['x'][i]
    y_b = vals['y'][i]
    z_b = vals['z'][i]

    separation = math.sqrt( (x_a - x_b) ** 2 + (y_a - y_b) ** 2 + (z_a - z_b) ** 2)
    if separation < d_0:
        ax.plot([vals['x'][i]], [vals['y'][i]], [vals['z'][i]], 'ro')
        start = i
        break

#run through time steps
distances = []
for i in range(len(vals['t']) - start):
    x_a = vals['x'][pt + i]
    y_a = vals['y'][pt + i]
    z_a = vals['z'][pt + i]
    x_b = vals['x'][start + i]
    y_b = vals['y'][start + i]
    z_b = vals['z'][start + i]
    d_1 = math.sqrt( (x_a - x_b) ** 2 + (y_a - y_b) ** 2 + (z_a - z_b) ** 2)
    distances.append(math.log(abs(d_1/d_0)))

#plot final points
ax.plot([x_a], [y_a], [z_a], 'go')
```

```
ax.plot([x_b], [y_b], [z_b], 'go')

#plot ln(delta) vs t
steps = [i for i in range(len(distances))]
fig = plt.figure()
plt.plot(steps[:1000], distances[:1000])
```

B.5 Runge-Kutta: Syncrhonization

```
#fourth-order integration
def runge_kutta_sync(x_init, y_init, y_r_init, z_init, z_r_init,
    num_iter, a, b):
    x = x_init
    y = y_init
    z = z_init
    y_r = y_r_init
    z_r = z_r_init

    #time step
    dt = 0.1
    iterations = num_iter
    t = np.linspace(0, dt * iterations, iterations)

    #plotting parameters
    X = np.empty((iterations,))
    Y = np.empty((iterations,))
    Z = np.empty((iterations,))
    Y_R = np.empty((iterations,))
    Z_R = np.empty((iterations,))

    X[0] = x
    Y[0] = y
    Z[0] = z
    Y_R[0] = y_r
    Z_R[0] = z_r

    for i in range(1, iterations):
        k_x1 = f(x, y, a) * dt
        k_y1 = g(x, y, z) * dt
        k_z1 = h(y, b) * dt
        k_yr1 = g(x, y_r, z_r) * dt
        k_zr1 = h(y_r, b) * dt
```

```
33
        k_x2 = f(x + k_x1/2, y + k_y1/2, a) * dt
35      k_y2 = g(x + k_x1/2, y + k_y1/2, z + k_z1/2) * dt
        k_z2 = h(y + k_y1/2, b) * dt
37      k_yr2 = g(x + k_x1/2, y_r + k_yr1/2, z_r + k_zr1/2) * dt
        k_zr2 = h(y_r + k_yr1/2, b) * dt
39
        k_x3 = f(x + k_x2/2, y + k_y2/2, a) * dt
41      k_y3 = g(x + k_x2/2, y + k_y2/2, z + k_z2/2) * dt
        k_z3 = h(y + k_y2/2, b) * dt
43      k_yr3 = g(x + k_x2/2, y_r + k_yr2/2, z_r + k_zr2/2) * dt
        k_zr3 = h(y_r + k_yr2/2, b) * dt
45
        k_x4 = f(x + k_x3, y + k_y3, a) * dt
47      k_y4 = g(x + k_x3, y + k_y3, z + k_z3) * dt
        k_z4 = h(y + k_y3, b) * dt
49      k_yr4 = g(x + k_x3, y_r + k_yr3, z_r + k_zr3) * dt
        k_zr4 = h(y_r + k_yr3, b) * dt
51
        x_0 = x
53      y_0 = y
        z_0 = z
55      yr_0 = y_r
        zr_0 = z_r
57
        x = x_0 + (1/6) * (k_x1 + 2*k_x2 + 2*k_x3 + k_x4)
59      y = y_0 + (1/6) * (k_y1 + 2*k_y2 + 2*k_y3 + k_y4)
        z = z_0 + (1/6) * (k_z1 + 2*k_z2 + 2*k_z3 + k_z4)
61      y_r = yr_0 + (1/6) * (k_yr1 + 2*k_yr2 + 2*k_yr3 + k_yr4)
        z_r = zr_0 + (1/6) * (k_zr1 + 2*k_zr2 + 2*k_zr3 + k_zr4)
63
        X[i] = x
65      Y[i] = y
        Z[i] = z
67      Y_R[i] = y_r
        Z_R[i] = z_r
69
    return {'x': X, 'y': Y, 'z': Z, 'yr': Y_R, 'zr': Z_R, 't': t}
```

Appendix C

Source Files: Signal Processing

C.1 Transmission and Encoding

```
import spidev
import time
import numpy as np

spi = spidev.SpiDev()
spi.open(0, 1) #open port 1 on bus 0
spi.max_speed_hz = 976000

message = """
I wandered lonely as a cloud
That floats on high o'er vales and hills,
A host of golden daffodils;
Beside the lake, beneath the trees,
Fluttering and dancing in the breeze.

Continuous as the stars that shine
and twinkle on the Milky Way,
They stretched in never-ending line
along the margin of a bay:
Ten thousand saw I at a glance,
tossing their heads in sprightly dance.
"""
message = message.strip()
message = list(message)
message_ascii = [ord(i) for i in message]
message_binary = [format(i, '07b') for i in message_ascii]
```

```
28 message_vectors = [ [int(i) for i in message_binary[j]] \
                       for j in range(len(message_binary)) ]
30
32 k = 7

34 gen_matrix = np.array([[1,0,0,0,0,0,0,1,0,0,0,1,0,1,1],
                          [0,1,0,0,0,0,0,1,1,0,0,1,1,1,0],
36                        [0,0,1,0,0,0,0,0,1,1,0,0,1,1,1],
                          [0,0,0,1,0,0,0,1,0,1,1,1,0,0,0],
38                        [0,0,0,0,1,0,0,0,1,0,1,1,1,0,0],
                          [0,0,0,0,0,1,0,0,0,1,0,1,1,1,0],
40                        [0,0,0,0,0,0,1,0,0,0,1,0,1,1,1]])

42 codeword_vectors = [ np.dot(vector, gen_matrix) \
                        for vector in message_vectors ]
44

46 codewords_mod_2 = np.mod(codeword_vectors, 2)
   codeword_str_vectors = [ [str(i) for i in codewords_mod_2[j].tolist()
       ] \
48                           for j in range(len(codewords_mod_2)) ]

50 aug_codewords = []
   for i in codeword_str_vectors:
52     aug_codewords.append('1' + ''.join(i[0:8]))
       aug_codewords.append('1' + ''.join(i[8:15]))
54
   padding_1 = '111111111'
56 padding_2 = '11111111'
   aug_codewords.insert(0, padding_2)
58 aug_codewords.insert(0, padding_1)
   aug_codewords.append(padding_1)
60 aug_codewords.append(padding_2)

62 #print(aug_codewords)
   #FROM MICROCHIP DATASHEET:
64 # 16-bit instruction
   # 0000  00  nn nnnn nnnn
66 # addr write 10-bit data
   # NOTE: data memory only 9-bits
68 #       msb is ignored
   def write_pot(input):
70     msb = input >> 8
       lsb = input & 0xFF
72     spi.xfer([msb,lsb])
```

```
##SWEEP OVER DIGIPOT VALUES (calibration)
##for i in range(0x00, 0x1FF, 5):
##    write_pot(i)
##    time.sleep(.05)
##    print(i)

HIGH = 100
LOW = 300
RATE = 0.02
GAP = RATE * 5
length = str(len(aug_codewords))

i = 0

write_pot(LOW)
for character in aug_codewords:
    for bit in character:
        if bit == '1':
            write_pot(HIGH)
            time.sleep(RATE)
        else:
            write_pot(LOW)
            time.sleep(RATE)
    i += 1
    write_pot(LOW)
    time.sleep(GAP)
    if i % 10 == 0:
        print("Packet:" + str(i) + "/" + length)

print("Finished Recording")
```

C.2 Receiver Analysis: Without Error-Correction

C.2.1 Waveform Plots

```
from scipy.io.wavfile import read
import matplotlib.pyplot as plt

bit_transmission_rate = 0.02
input_data = read("new_coded_message.wav")
audio = input_data[1]
```

```
sample_rate = input_data[0]
audio_length = len(audio)/sample_rate
samples_per_bit = int(bit_transmission_rate * sample_rate)

print("Sampling Rate:", sample_rate, "samples/sec")
print("Samples:", len(audio))
print("Audio Length:", audio_length, "sec")
print("Bit Transmission Rate:", bit_transmission_rate, "sec")
print("Samples per Bit:", samples_per_bit)

start = 242800
threshold = 13000

plt.axhline(y=threshold, color='blue', lw=3)
plt.axhline(y=-threshold, color='blue', lw=3)
plt.plot(audio[start+samples_per_bit*0:start+samples_per_bit*50:1])
plt.ylabel("Amplitude")
plt.xlabel("Time (samples)")

plt.title("Audio Waveform Sample")

digitized_audio = [0 if abs(i[0]) < threshold else 1 for i in audio]
print(len(digitized_audio))

num = 0
manual_shift= 0
plt.plot(digitized_audio[manual_shift+start+samples_per_bit*9*num:
    manual_shift+start+samples_per_bit*9*(num+1)])

for i in range(10):
    plt.axvline(x=samples_per_bit * i, color='yellow', lw=3)

plt.ylabel("Amplitude")
plt.xlabel("Time (samples)")

plt.title("Binary Digitized Waveform Sample")
```

C.2.2 Message Extraction

```
import string

byte = ['0', '0', '0', '0', '0', '0', '0', '0']
message_bytes = []
```

```
message_ascii = []
message = []
bits = []

samples_per_bit = 900

start = True
buffer = 100
min_count = 30
idx = 0
j= 0
while idx < len(digitized_audio) - 1:

    while digitized_audio[idx] != 1:
        idx += 1

    for i in range(idx + samples_per_bit, idx + samples_per_bit * 9,
    samples_per_bit):
        bits = digitized_audio[i+buffer: i+samples_per_bit-buffer]
        if bits.count(1) > min_count:
            bit = 1
        else:
            bit = 0

        byte[j] = str(bit)
        j += 1

        if j % 8 == 0:
            j = 0
            idx += samples_per_bit * 10
            message_bytes.append(''.join(byte))

    if ''.join(byte) == '11111111' and not start:
        break
    else:
        start = False

message_ascii = [int(num, 2) for num in message_bytes]
message = [str(chr(character)) for character in message_ascii]

corrected_message = [char for char in message if char in string.
    printable]
print(''.join(corrected_message))
```

C.3 Receiver Analysis: With Error-Correction

C.3.1 Meggitt Decoding

```
import numpy.polynomial.polynomial as P
n = 15 #Length of cyclic code
generator = [1,0,0,0,1,0,1,1,1] #g(x) = 1 + x^4 + x^6 + x^7 + x^8

def calc_syndrome(error):
    shift = P.polymul(error, [0,0,0,0,0,0,0,0,1])
    remainder = P.polydiv(shift, generator)[1]
    rem_mod_2 = [i % 2 for i in remainder]
    return rem_mod_2

error_patterns = []
for i in range(15):
    pattern = [0] * 15
    pattern[14] = 1
    pattern[i] = 1
    error_patterns.append(pattern)
error_lookup = {tuple(calc_syndrome(pattern)): pattern for pattern in
    error_patterns}

def meggitt_decode(received):
    codeword = received
    shift = [0] * 15

    for i in range(0, 15):
        shift[i] = 1
        syndrome = tuple(calc_syndrome(P.polymul(shift, received)))
        if syndrome in error_lookup.keys():
            if i != 0:
                shift[i] = 0
                shift[15-i] = 1
            error = P.polymul(shift, error_lookup[syndrome])
            error_mod = [i % 2 for i in P.polydiv(error,
   [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1])[1]]
            codeword = P.polysub(received, error_mod)
            codeword = [int(i % 2) for i in codeword]
            return codeword[0:7]
        shift[i] = 0

    return codeword[0:7]
```

C.3.2 Message Extraction

```
import string

received = ['0'] * 15
message_bin = []
message_ascii = []
message = []
bits = []

samples_per_bit = 894

start = True
buffer = 100
min_count = 30
idx = 0
j = 0
segment_len = 0

while idx < len(digitized_audio) - 1:
    segment_len += 1
    segment_len = segment_len % 2

    while digitized_audio[idx] != 1 and idx < len(digitized_audio) -
    1:
        idx += 1

    for i in range(idx + samples_per_bit, idx + samples_per_bit * (8+
    segment_len), samples_per_bit):
        bits = digitized_audio[i+buffer: i+samples_per_bit-buffer]
        if bits.count(1) > min_count:
            bit = 1
        else:
            bit = 0

        received[j] = bit
        j += 1

        if j % 15 == 0:
            j = 0
            char_bits = meggitt_decode(received)
            char_bits_str = [str(i) for i in char_bits]
            message_bin.append(''.join(char_bits_str))

            if ''.join(str(c) for c in char_bits) == '1111111' and
    not start:
```

```
                break
            else:
                start = False

    idx += samples_per_bit * 10


message_ascii = [int(num, 2) for num in message_bin]
message = [str(chr(character)) for character in message_ascii]

corrected_message = [char for char in message if char in string.
    printable]
print(''.join(corrected_message))
```

List of Figures

References

[1] Gleick, J., 1987. *Chaos: Making a New Science*. Penguin Books, New York, NY.

[2] Lorenz, E.N., 1963. "Deterministic Nonperiodic Flow." *J. Atmos. Sci.*, 20, pp.130-141.

[3] Pecora, L.M., Carroll T.L., 1990. "Synchronization in Chaotic Systems." *Physical Review Letters*, 64(8), pp.821.

[4] Cuomo, K.M., Oppenheim, A.V. and Strogatz, S.H., 1993. "Synchronization of Lorenz-based chaotic circuits with applications to communications." *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 40(10), pp.626-633.

[5] Strogatz, S.H., 2014. *Nonlinear Dynamics and Chaos: with applications to physics, biology, chemistry, and engineering*. Westview press.

[6] Kennedy, M.P., 1993. "Three steps to chaos. II. A Chua's circuit primer." *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 40(10), pp.657-674.

[7] Kennedy, M.P., 1992. "Robust OP Amp realization of chua's circuit." `Frequenz.`, 46(3), pp.66-80.

[8] Malisoff, M. and Mazenc, F., 2009. *Constructions of strict Lyapunov functions*. Springer Science & Business Media.

[9] Ogorzalek, M.J., 1993. "Taming chaos. I. Synchronization." *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 40(10), pp.693-699.

[10] "Microchip: 7/8-Bit Single/Dual SPI Digital POT with Volatile Memory." http://ww1.microchip.com/downloads/en/DeviceDoc/22060a.pdf

[11] Richardson, M., 2012. "Simple Guide to the RPi GPIO Header and Pins." http://www.raspberrypi-spy.co.uk/2012/06/simple-guide-to-the-rpi-gpio-header-and-pins/

[12] Uchida, A., 2012. *Optical communication with chaotic lasers: applications of nonlinear dynamics and synchronization.* John Wiley & Sons.

[13] Takaitra, M., "Controlling the MCP4151 Digital Potentiometer with the Raspberry Pi." http://www.takaitra.com/posts/503

[14] Glover, J.C., Lazzarini, V. and Timoney, J., 2011. "Python for audio signal processing."

[15] Huffman, W.C. and Pless, V., 2010. *Fundamentals of error-correcting codes.* Cambridge university press.