2024

Generation of true random numbers with a first order chaotic circuit

Austin Davis

GENERATION OF TRUE RANDOM NUMBERS WITH A FIRST ORDER CHAOTIC CIRCUIT

Austin Davis

A THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering
in
Electrical and Computer Engineering
to
The Graduate School
of
The University of Alabama in Huntsville

August 2024

Approved by:

Dr. Aubrey Beal, Research Advisor/Committee Chair
Dr. Ned Corron, Committee Member
Dr. Laurie Joiner, Committee Member
Dr. Aleksandar Milenkovic, Department Chair
Dr. Shankar Mahalingam, College Dean
Dr. Jon Hakkila, Graduate Dean

Abstract

GENERATION OF TRUE RANDOM NUMBERS WITH A FIRST ORDER CHAOTIC CIRCUIT

Austin Davis

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering

Electrical and Computer Engineering

The University of Alabama in Huntsville

August 2024

True randomness backed by first principles theory is surprisingly rare in physically implemented circuit for cybersecurity applications. This thesis provides an illustrative case of a chaotic one-dimensional map circuit with a design driven by a need for rudimentary theory concerned with the limits of its entropy production. Analysis, simulation, and hardware measurements are evaluated against statistical randomness tests issued by the National Institute of Standards and Technology. Interestingly, the results provided here highlight strong relationships between first principles theory of design and measured results when varying parameters. We observe matching characteristics between analytic, simulated and measured hardware results for NIST performance indicators of entropy and randomness. Altogether, this work enables theoretical guidance for entropy assurance in a class of chaotic oscillators used as hardware security primitives.

Acknowledgements

I would like to graciously thank my professors at UAH, without whom I would not be the engineer I am today. Dr. Laurie Joiner for imparting on me so much knowledge and keeping me on top of deadlines when I needed to be. Dr. Beal for introducing me to chaos and allowing me to continue working with him throughout my graduate program.

For their expertise in the field of nonlinear dynamics, I would like to thank Dr. Ned Corron and Dr. Seth Cohen for advising me. Without their assistance my work would not nearly be up to the standard it is today. I stand on the shoulders of giants.

For their camaraderie and support, I would like to thank my peers and lab-mates; Aidan Barton, Tamseel Syed, Mattan Tseng, and Micah Tseng. My dearest friends who have kept me sane through all the work; Will, Sam, Lachlan, Ella Grace, Nick, Elizabeth, Colin, Rachel, and Katie. A final thanks to my parents and family for their unyielding love.

Table of Contents

List of Figures

List of Tables

Introduction

As our increasingly digital world relies on trusting the security of our devices, how can we be assured our data and communications are actually secure? Encryption is the heart of data privacy and secure communications. It relies on having plentiful access to random bits in order to obscure data from malicious actors [19]. But what happens when the bits used to encrypt our data fail to be sufficiently random? Methods of producing true random numbers can be susceptible to vulnerabilities [3] or have design flaws that limit their effectiveness [24]. More reliable and cost effective solutions are needed to produce quality random bits. Can the properties of solvable chaos and nonlinear dynamics be utilized to implement in hardware a circuit capable of generating highly entropic random bit strings?

Chaos theory defines systems with sensitive dependence on initial conditions where trajectories of sequences will differ greatly from one another over time [2]. Chaotic systems are capable of infinitely producing strings of non-repeating numbers. The best way to harness this is with solvable chaos. Dynamical differential equations with exact solutions can have chaotic behavior even though the properties of solvability and chaos seem antithetical [8][6]. With an exactly solvable differential equation, the system's behavior can be modeled and realized in hardware.

Any physical system needs to be evaluated empirically to measure adherence to the theoretical system. The primary qualifier for a system's randomness is entropy. Entropy is a measure of new information revealed to an observer after an event [26]. High entropy is a quality of data with uncertainty. That is, data do not contain predictable patterns such that future data cannot be guessed from observed past data with confidence. Maximum entropy of a system can be calculated analytically, but for the physical realization of a chaotic system, the entropy can only be estimated from the observation of large sets of data. The performance of the circuit will be evaluated in terms of its entropy and statistical randomness. The research will evaluate its viability as an entropy source for the generation of random bit-streams for use as a true random number generator.

Background

High quality random numbers are needed for secure electronics. Fields include cryptography, password and key generation, data encryption and compression, as well as nonce generation and simulated randomness in games of chance [21][13]. Our digitally connected world relies on these cryptographically secure random numbers more every year as the number of connected devices continues growing. In the IoT sector alone it is estimated that over 20 billion devices will be added by 2025 compared to the surveyed number of devices taken between 2015 and 2019 [1]. Every new device represents an attack vector for malicious actors. This increasingly long chain is still only as strong as its weakest link.

There are currently many implementations of random number generators with different robustness and limitations. The primary two methods to create a cryptographically secure random number generator (CSRNG) are with a pseudo random number generator (PRNG) and a True random number generator (TRNG). Both methods have the capability to produce cryptographically secure random numbers. True random number generators produce randomness by observing a physical entropy source. Thermal noise, atmospheric noise and electromagnetic radiation and even a wall of lava lamps [13] are used as entropy sources. A pseudo random number generator instead creates random numbers algorithmically, given some input initial condition or seed.

True random number generators exhibit blocking behavior [27][17]. They generate new random numbers as fast as the entropy source can be sampled to fulfill the need of the desired amount of random numbers. This creates a limitation where requests for random numbers occur more frequently than a system can produce unique numbers. The raw outputs from physical sources of randomness may not be sufficient for use in secure electronics. The raw output may contain biases or be non-uniformly distributed. Some conditioning can be performed on the output to make it more ideal in uniformity or to remove biases [20][13].

Pseudo random number generators are not as limited in speed. Numbers are generated algorithmically as fast as the hardware can compute them [18][4]. The PRNG must be reseeded periodically [20] to continue producing random numbers. A pseudo random number generator is deterministic and relies on a robust algorithm that is not prone to creating repeated sequences and an entropy source that can produce a wide array of seed values [4]. Each seed to the PRNG has a finite amount of entropy that is used up as the algorithm produces new bits. Once the PRNG produces a repeated pattern, after a long sequence of statistically random bits, then it has exhausted all the entropy the seed value was able to provide [20]. Flawed PRNGs may rely on entropy sources that do not seed the algorithm with sufficient entropy. Seed values with low entropy will cause the PRNG to frequently produce shorter unique sequences. An entropy source that only produces a limited number of values will cause the PRNG to generate the same sequences often. Another point of failure for a PRNG is a poor algorithm. If the PRNG has a bias, then it may tend towards certain sub-sequences even with a high entropy seed. These underlying biases can be exploited to guess future values. The most robust systems use a hybrid implementation where a TRNG acts as an entropy source which seeds a quality PRNG. This approach allows for the high speed generation of random numbers found in PRNGs along with the high entropy of a TRNG [20].

Implementations of random number generators in use today still exhibit the same basic flaws. Systems that lack a robust, fast entropy source with an unbiased data can be circumvented by malicious actors. Intel's third generation of core processors, code-named Ivy Bridge, included a true random number generator on chip [11][27]. Their implementation utilizes a metastable RS-NOR latch as its entropy source [11]. The state of the random bit fluctuates due to thermal noise conditioned by a capacitive feedback which nudges value 1 bits towards 0 [11]. All bits sampled from the entropy source are conditioned with post processing circuitry to improve the quality of randomness [16]. In a collaborative research effort led by professors at the University of Massachusetts Amherst, a method of undermining the TRNG system with a hardware level trojan was discovered [3]. The attack reduces the entropy from 128 bits to just 32 bits. The generated random numbers could become so predictable that it would be trivial for an assailant to guess the numbers with a high degree of accuracy. It was concluded that this method of tampering was undetectable by the system's built-in self tests.

In a report for DEF CON 29, researchers reported on a security flaw affecting millions of IoT devices [24]. The IoT devices in question utilize a TRNG System on a Chip (SOC) to create random numbers. This TRNG system was prone to blocking errors when function calls to the TRNG subsystem exceeded the TRNG's ability to produce random numbers. They observed three failure states that the devices could enter depending of what error handling capabilities the IoT device had. The random number generator could produce numbers with only partial entropy, output sequences of zeros, or even return uninitialized memory [24]. The two current solutions for handling the error are to abort the process all together or to await its completion when enough bits can be generated. Neither of these are seen as acceptable solutions, so developers leave the errors unhandled. This leaves millions of common IoT devices unsecured. So what tools do we have at our disposal to create a high quality random number generator with a measurable entropy?

2.1 Chaos Theory

A new paradigm in random number generation is the use of chaos as an entropy source for true and pseudo random number generators. Chaos is defined by two characteristics: sensitive dependence on initial conditions and deterministic behavior [2][18]. A deterministic system that is dependent on initial conditions seems like it would not be a good entropy source. That definition more closely describes a PRNG than a TRNG. However, a deterministic chaotic system does have an entropy rate associated with it [14][19]. PRNGs cannot create entropy

[9]. They can only use the entropy provided by the seed value. Chaotic systems amplify the least significant bits of an initial condition to radically change its trajectory over time [20][9].

Before continuing, it is important to discuss several key aspects of chaotic systems that will be used to qualify its characteristics and quantify its performance. One of the most well known chaotic systems is the logistic equation, defined in equation 2.1. We will observe its properties to understand how to characterize other chaotic systems:

$$x_{n+1} = f(x_n) = \beta x_n(1 - x_n). \tag{2.1}$$

2.1.1 Stability, Fixed Points, and Orbits

A fixed point is defined in a deterministic chaotic discrete-time system as any value, such that the subsequent value is equal to the current value. This means that the system has reached a state for which there is only one possible value it can achieve [2]. These points can be solved for in any discrete time system by defining the output variable of the function to be the same as the input and solving for that variable. Mathematically defined as $x_{n+k} = f^k(x_n) = x_n$, where k is any positive integer denoting the function is applied to itself k times.

Each fixed point can either be stable or unstable. Stable fixed points in a system will cause values in the neighborhood of the fixed point to converge towards the fixed point. Unstable fixed points are the opposite. A point in the neighborhood of an unstable fixed point will diverge from it over time [2]. The stability of a fixed point can be found by taking the magnitude of the derivative at the fixed point f_p. If the magnitude of the derivative is less than 1, then the fixed point is stable. If the magnitude of the derivative is greater than 1, then the fixed point is unstable [2][29]. Both fixed points of the logistic map occurring at 0 and $\frac{\beta-1}{\beta}$ are unstable. This is good for the overall system as it will not become stuck in a fixed point.

Similar to fixed points, orbits are any set of points, such that after n samples, the points will have the same value [2]. The system is in a repeating pattern of values over some interval of length n. Orbits of length n are of period n. Orbits are calculated in a similar manner as the fixed points. For a period n orbit, the function will be nested n times and the input and output will be defined as the same variable. In the case of a piecewise function, all permutations of the equations must be tested to find all orbits.

Orbits can also be chaotic. A chaotic orbit perpetually exhibits unstable behavior which is neither fixed nor periodic for any finite amount of time [2]. A system is in a chaotic orbit if it meets the conditions of periodicity and a positive Lyapunov exponent, or the deviation rate, of the system is greater than zero [2].

2.1.2 Maps and Partitions

The dynamics of a chaotic system can be represented graphically by a return map. A return map plots the current value in a discrete system against the next value. It is a useful tool for observing the behavior of the discrete time series. The logistic map is plotted in Figure 2.1.

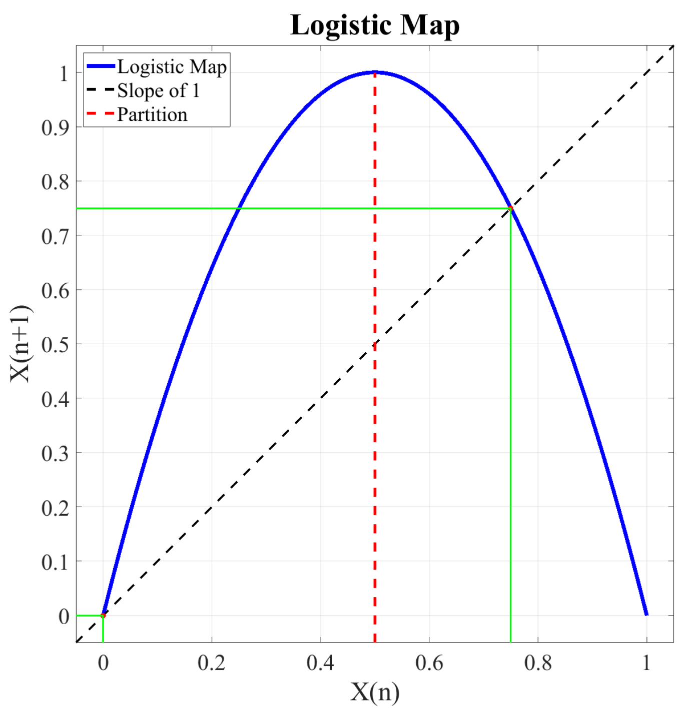

Figure 2.1: The one-dimensional return map of the ideal, full height logistic map. It features two unstable fixed points at 0 and $\frac{\beta-1}{\beta}$. It can be exactly partitioned at $\frac{1}{2}$.

Plotting a slope 1 through the map further reveals its characteristics. Locations where the line intersects the map reveal the system's fixed points [2]. For points on the map above the line of slope 1, a line is traced from the map horizontally to the right until reaching line of slope 1. Then, a line is traced vertically towards the map. For points below the line of slope 1, a line is traced from a point on the map to the line of slope 1 horizontally to the left. Then a line is traced from the line of slope 1 vertically towards the map. The new location on the map is the next value the system will produce.

Any map can be partitioned such that it is divided into more than one region containing points. Splitting the map into multiple regions allows the points in those regions to be referred to symbolically. Data points can be 'A' and 'B' instead of their individual numeric values. Symbolic dynamics is useful for observing trends in the data as the states change over long periods of time.

2.1.3 Bifurcation and the Lyapunov Exponent

As previously mentioned, the dynamics of the system are affected by the constants in the dynamical equations. By altering the value of β in the logistic map equation, the trajectories of the map and possible values in the system also change. Plotting return maps of systems with decreasing slope parameters shows a reduction in the height of the map. There are now fewer values the system is able to be. An orbit diagram, commonly referred to as a bifurcation diagram, plots the possible values the system is able to produce against the slope parameter used in the system's equations.

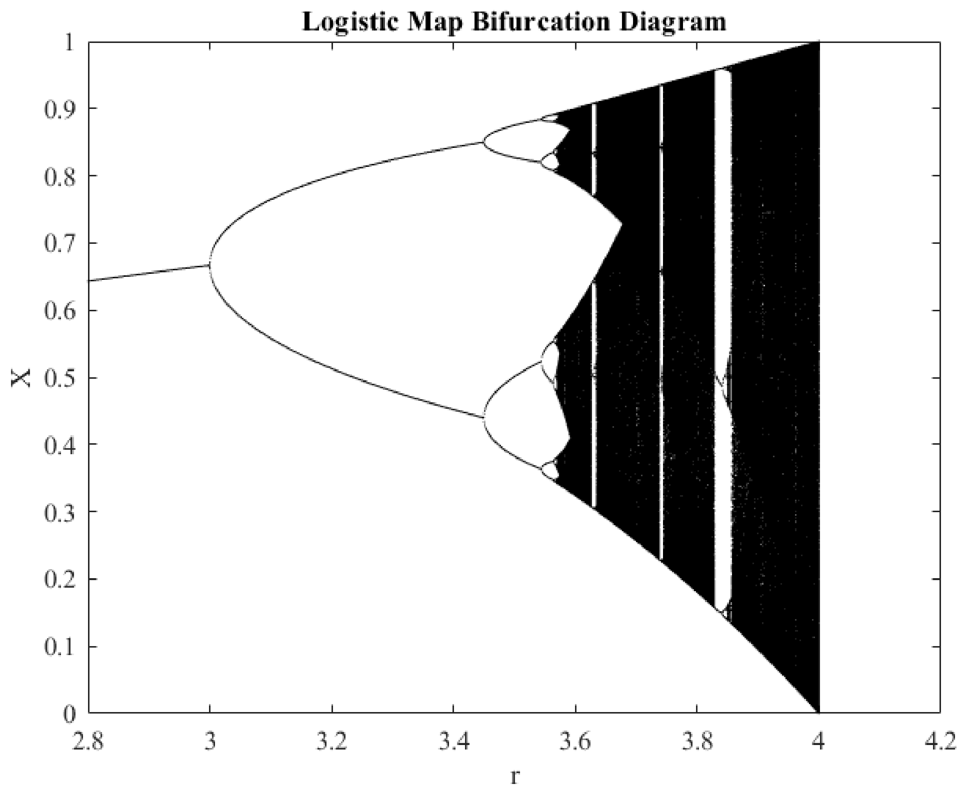

Figure 2.2: Bifurcation diagram of the logistic map equation. As the slope parameter 'r' increases, the greater number of values can be contained in an orbit. For the largest values, the system exhibits chaotic behavior where it can take on any value in the region.

As the slope parameter approaches its maximum value, the system will produce a more robust chaos. The diagram of the logistic map shows regions where there can only be one value for the system, regions where a few values can be produced, and regions where the system becomes chaotic. In chaotic regions, the system is able to produce values anywhere between the lower and upper bounds. In both the logistic and tent map, as the slope parameter approaches its maximum possible value, the range of possible values expands to the entire interval of (0,1). The system becomes more robust as the slope parameter and range of values increase.

The Lyapunov exponent is a measure of the average rate of per-step divergence rate of points in a sequence on an orbit [2]. This can be defined mathematically as the natural logarithm of the summation of the derivative of the function taken at points x_n, where n is integers from one to infinity, divided by the number of points n:

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} |f'(x_i)|. \tag{2.2}$$

For two sequences starting at arbitrarily close and unequal points, the difference between the points after each iterate will change based on the Lyapunov exponent. A positive exponent indicates the points will diverge at a rate of e^{λ} [2][23][8]. Diverging sequences is a requisite for a chaotic sequence. If a system has a negative Lyapunov exponent, then it will converge to a fixed point.

The system has no stable fixed points and for higher values of μ there is a robust set of values the system can take on. For all the qualities that make this chaotic system useful, why not implement it as a digital PRNG instead of a physical random bit generator? The answer lies in precision. Any digital system has a finite number of bits that can be used to represent any data point. This information limit is inherent to how a PRNG can only use the entropy provided to it from a seed value. As the least significant bits are amplified, no new entropy is created. A physical system is sensitive to external noise [20]. A physical implementation of a chaotic system will constantly be re-seeded in its initial condition. Thermal and electromagnetic radiation nudge the voltage values causing variance in the least significant digits of voltage values. Though an ideal physical system implementing the tent map is still deterministic, the influence of outside noise makes it an ideal entropy source. The small variations in voltages have drastic long-term effects on the trajectory of the system.

2.1.4 IID and Markov Processes

For an ideal random variable, it should meet the sufficient conditions for independence and identical distribution (IID). That is, each value outcome from an observation of the random variable should have no correlation to any previous or future outcomes. Every observation is entirely independent. The distribution of values observed must also have the same distribution for all observations. No factors will alter the probability density. A binary random variable will have a uniform distribution. It is expected that for an infinitely large sample there will approximately be the same number of 1's as 0's. Chaotic systems are deterministic, so it may seem impossible for one to be IID since it would fail the condition of independence. For a chaotic system to achieve IID, one must look at the symbol sequence instead.

A first order Markov process only depends on its current state to determine its next state as defined by the state transition probabilities[30][26]. Chaotic systems can be defined as Markov processes when the state transitions are known. Since the outcomes of chaotic systems are deterministic, each state contains all the information from all prior states. For a chaotic map with a Markov partition, whereby the map is one-to-one, the backwards itinerary can be traced along an infinite sequence of known states. Each step backwards in time will reduce the space each symbol would have been established in. It would require knowledge of an infinite number of symbols to know the state of the system and be able to predict future values. This requirement of infinite knowledge of the symbol sequence is what allows chaotic systems to be classified as IID Markov processes. Chaotic systems of higher order than one can be reduced to order one by observing longer words of the order of the Markov process

2.2 Tent Map

The tent map has been shown to be an effective system in various engineering applications [23]. The tent map chaotic system can be described by a piecewise linear discrete time equation which maps all of its points in the set onto itself. The system has two variables to control the behavior. This research will be focusing on the ideal version of the system where the slope parameter $\mu = 2$ and the threshold $\gamma = 0.5$. The system maps points on the interval [0,1]. The system at these conditions is referred to as a full height tent map as the slope cannot be greater than 2. The full height tent map features a Markov partition at threshold which is the peak of the map at $x_n = 0.5$:

$$f(x_n) = x_{n+1} = \begin{cases} \mu x_n & x \leq \gamma \\ \mu(1 - x_n) & x > \gamma \end{cases} \quad x_n \in [0, 1]. \tag{2.3}$$

The bifurcation diagram shows how the robustness of the chaos grows as μ increases towards 2. The range of values the system is able to take on also increases. The system features two fixed points at 0 and $\frac{2}{3}$ [12]. Both fixed points are unstable because the absolute value of the derivatives of the function for both pieces are greater than 1. When the system travels near a fixed point, future iterates will grow away from the fixed point. A derivation of these calculations can be found in Appendix A, A.5.

The tent map is named for the shape of its one-dimensional return map [12][15]. It is formed by two sloping lines making a triangular shape. The slope on each side is defined by the value of μ in the equations. The left half is a positive slope and the right half is a negative slope of the same magnitude. The map for the ideal system reveals several properties by observation, that have been determined analytically before. There is a Markov partition in the full height map at 0.5. The intersections of the map and a line with a slope of 1 show the system's fixed points at 0 and $\frac{2}{3}$, as derived in A.5 and A.6. The slope of the lines is also what determines the growth rate or the Lyapunov exponent. It is defined as the natural logarithm of the slope magnitude. $\ln \mu = \ln 2 = 0.693$ The tent map, at a slope of $\mu = 2$, is a first order Markov process. The probabilities for all state transitions are also $\frac{1}{2}$. While all the state transitions are equal symbols generated from the map are independent and identically distributed.

The ideal tent map's partition at 0.5 is also a Markov point. Markov partitions require a real-valued system where the set of all points fully maps onto itself, $[0, 1] \to [0, 1]$, this includes the full height tent map. A Markov partition

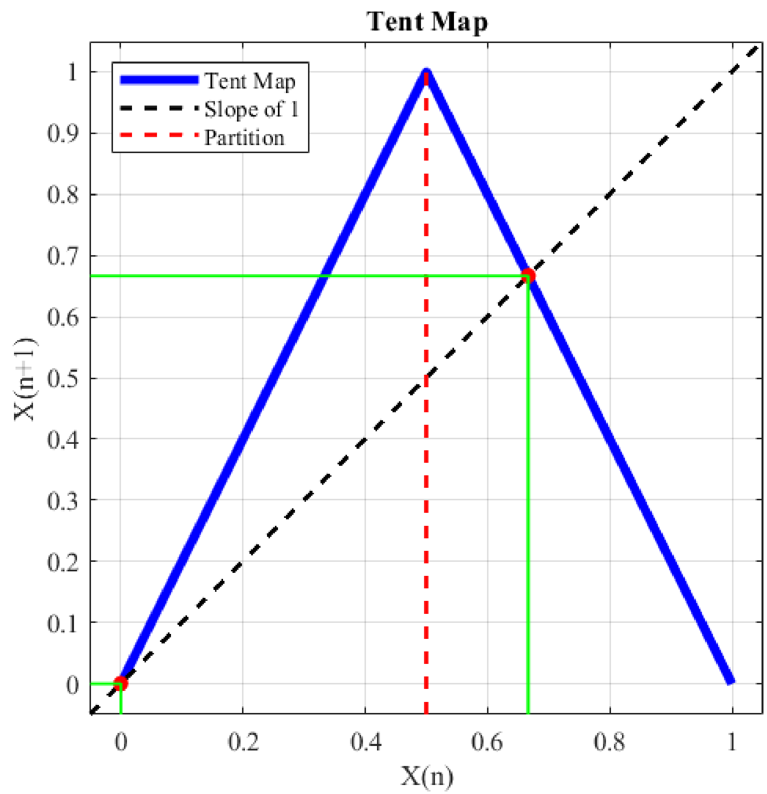

Figure 2.3: The one dimensional return map of the tent map system. In blue is the mapping of points from the piecewise linear equation. The black dashed line has a slope of 1. Where it intersects with the Map shows the system's fixed points as red dots. The red dashed line is the partition which divides the map into two regions.

exists where the map can be divided into regions where the system is monotonically increasing or decreasing and maps onto a union of other partitions [2][5]. Versions of the tent map below ideal height do not feature complete mapping onto for all regions of the map.

A strength of the tent map is its large regions of chaos. This makes it a more robust system than other systems, like the logistic map. There are not many regions where the system will fall into limited periodic orbits. The system always converges to 0 when μ is set to a value below 1. At values greater than 1, the system enters into chaotic orbits and becomes more robust as the slope increases past $\sqrt{2}$.

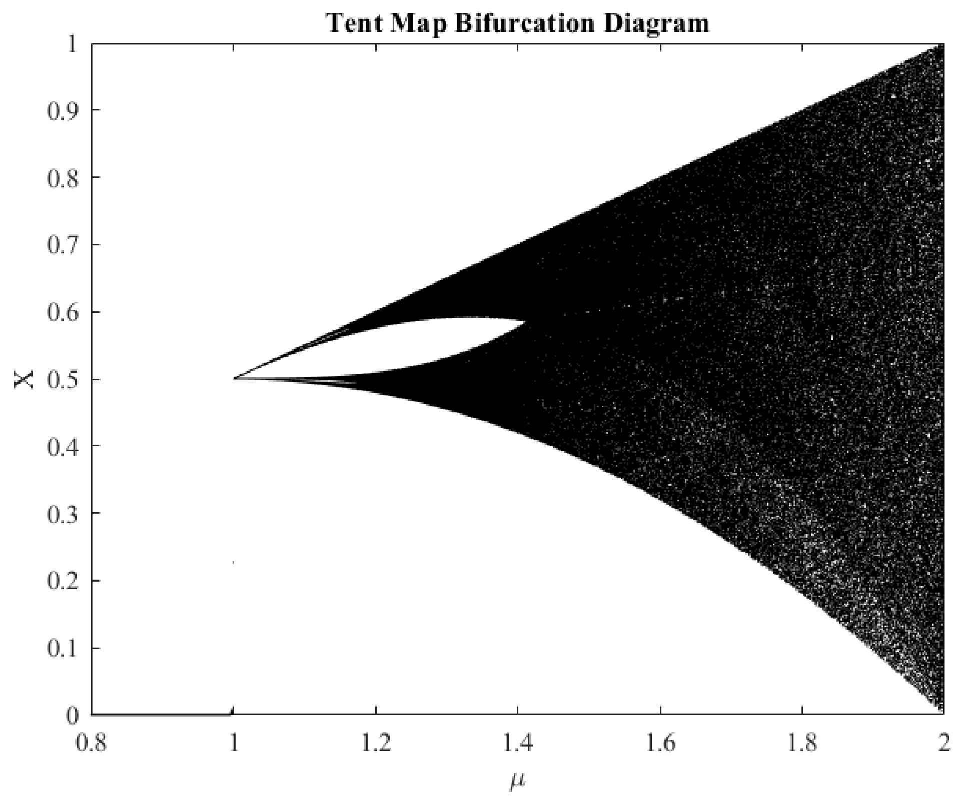

Figure 2.4: Bifurcation diagram of the tent map equation. As the slope parameter μ increases, the region of chaos also grows to the maximum region of (0,1). Below values of 1, the system always collapses to a value of 0.

It is very useful to work with the tent map system as a discrete equation of iterated points. The equation can be easily tuned by the only parameter μ and it only features linear equations. However, there is a more complex way to analyze the system with a differential equation and continuous time series. A conjugate of the system, represented by a first-order differential equation [6], is defined in equation 2.4:

$$\dot{u}(t) = u(t) + s. \tag{2.4}$$

Solving this first order ODE provides the solution for the continuous time representation of the tent map. A complete derivation is found in Appendix B, B.1. The resulting equation is a summation of basis pulses[8][7] that form the entire time series, 2.5. The pulses are forced by the function $s(t)$ which can take on two states $-1, +1$. The forcing function appears the the solution as s_n which takes on values $-1, +1$. It controls the sign on the summation of pulses. A new pulse must be added to the summation within the fixed time period, T:

$$u(t) = \sum_{n=-\infty}^{\infty} s_n P(t - nT). \tag{2.5}$$

The basis pulse is piecewise in time, equation 2.6. Before the start of the forcing function pulse the wave grows exponentially from $t = 0$ to $\frac{1}{2}$, from a starting time of $t = -\infty$ to time $t = 0^-$. After the start of the forcing pulse at time 0, the wave exponentially decays back to 0 within the pulse time T. For all time in the future, the function is 0:

$$P(t) = \begin{cases} (1 - e^{-T})e^t, & t < 0 \\ 1 - e^{t-T}, & 0 \leq t < T \\ 0, & T \leq t \end{cases}. \tag{2.6}$$

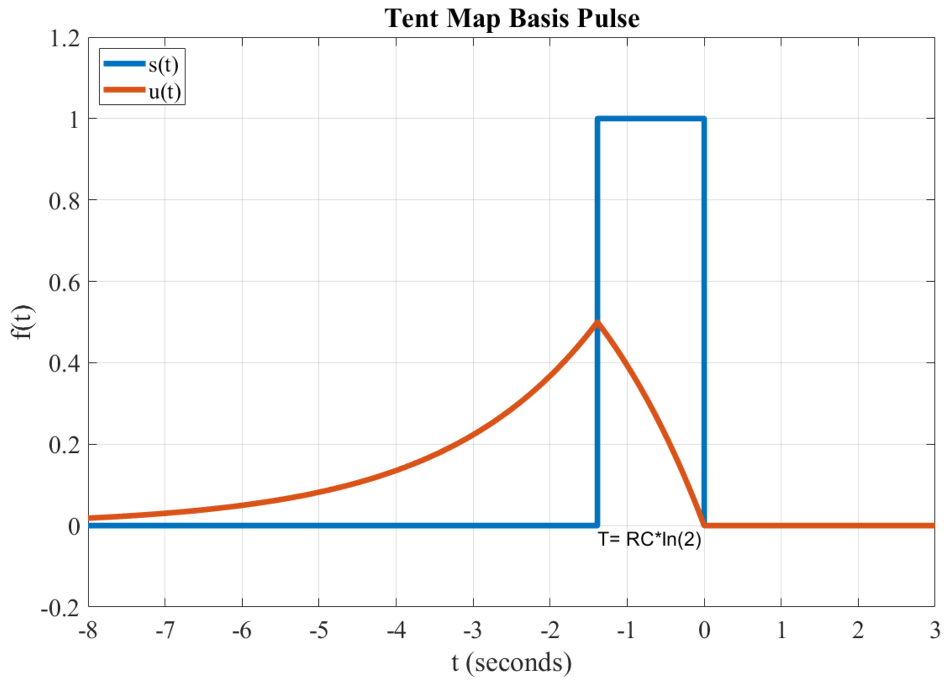

Figure 2.5: The basis pulse for the tent map continuous time waveform. The forcing function s(t) is a square wave with a period of T defined by the RC values used. The wave formed rises to the edge of the square wave at its midpoint before falling back down to 0 across the duration of the pulse. The function is an exponential growth curve and an inverted exponential growth curve.

2.3 Entropy and Information

A physical RNG system should be evaluated on its ability to produce random numbers but also on its qualities as an entropy source [20]. Biased outputs are able to be corrected with post-processing, but the entropy of a system plays a greater role in the viability of the system to produce random numbers [9]. In its broadest terms, entropy is a measurement of the surprisingness of a given event and how much information is gained from the observation of a single event [26].

If an observer could see all the data a system had produced for all time prior to now, how well could that observer predict the outcome of a new observation? Predictable, repetitious data will have low entropy. Unpredictable, independent data will measure as having high entropy.

Now that we have declared a system to generate entropy, we must define ways to measure it to ensure our system is producing a quality output. Shannon defined a method of measuring entropy based on the probabilities of a random variable's output [26]. $H = -K \sum_i^n p_i \times \log p_i$ Where p_i is the probability of occurrence of each state. The constant K is used as a base change to measure either in bits or nats. Utilizing the properties of logarithms, a logarithm of base 'x' can be changed to a base of 'y' by dividing by the logarithm of 'y' with base 'x'. If the log base used is 2, then the measurement is in bits. Using the natural logarithm gives you a measurement of entropy in nats [26].

The simplest case to consider is a binary random variable. There are two possible outcomes for any given event. If an observer were to see one million events all be a 1, then they should not be surprised when the next observed event is also a 1. Given a probability of occurrence of 1, nothing new is learned about the system from the observation. The outcome was already known. In the ideal case, each event has an exact probability of 0.5. From previous observations, an observer would not be able to correctly guess the outcome of the next event with more than 50% accuracy. Each new observation provides an entirely new bit of information to an observer. The relationship between probability and Shannon entropy for a binary random variable is shown in 2.6.

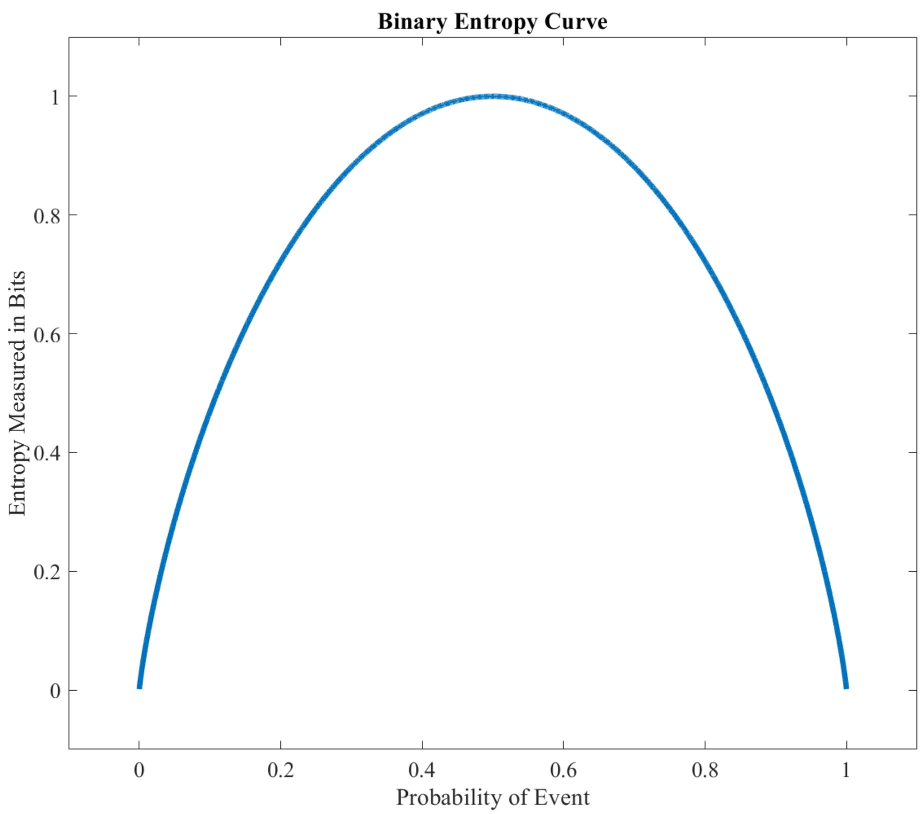

Figure 2.6: Plot of the Shannon entropy of a binary random variable given the probability of one of the two outcomes. The maximum entropy of 1 is achieved when the outcomes have equal probability at 1/2. This is true intuitively as the most uncertainty is when all possibilities have the same likelihood of occurring.

This measure H is the entropy per symbol of any given observation [26]. For a Markov process with a defined rate at which symbols are generated, the entropy rate is defined as the entropy per symbol times the frequency at which new symbols are produced [26]. This definition holds only for systems that are independent in time and stationary. [10] This measure is denoted as H'. $H' = \sum_{i=1}^{n} H_i$ An entropy source must be fast to produce new symbols at a rate greater than needed by the system using the data to eliminate the blocking behavior of some TRNGs discussed prior.

The Lyapunov exponent of the ideal tent map system is the natural logarithm of the map's slopes, which is $\ln 2 = 0.6931$. An interesting property is how the slope of the map relates to the entropy of the system. The ideal system will exactly map onto itself. The system is tuned exactly to a Markov point when the slopes of the tent map are 2 and the probability of each state is one-half. The Shannon entropy, in bits, is 1 and in nats is 0.6931.

As the slope of the tent map is lowered from the ideal, the system will show grammar restrictions as longer runs and certain combinations of bits no longer naturally occur from the generated symbol sequence. The probability of each word occurring will no longer be equal and will become more unequal as the slope lowers to 1. A derivation of the equivalency between Shannon entropy and the growth rate of the system for a tent map with a slope of $\mu = 2$ can be found in Appendix C C.1. With this relationship, there is an assurance that the maximum entropy of any tent map with slope greater than 1 can be known from the logarithm of the slope. There is a provable entropy source of known quality generating bits [9].

Circuit Design

The circuit design used, Figure 3.1, is based on the work of Corron *et al.* [7]. The circuit was designed to implement the first order chaotic tent map as described by the prior discussed equations. Their design features the same configuration of a negative RC filter paired with a digital reset latch by comparators and buffers. They also employ a pair of diodes in parallel to ground to force symmetry on the square pulses that drive the negative RC. The diodes clamp the voltage output of the comparator to the forward bias voltage of the diodes, or approximately ±0.7v. Using a precise voltage divider can achieve a similar result without the use of diodes. However, including the diodes allows for more flexibility in design.

Their original design uses MAXIM MAX912 comparators. The new design uses the Texas Instruments LM339N to achieve the same output. Their digital latch uses a NOT input on one of the two AND gates to invert the clock signal input. The implementation of an exclusive-or (XOR) gate with the second input driven by a positive voltage will produce the same result.

The design can be divided into two primary sections. There is an analog negative RC filter and a digital reset latch, as well as the components that connect them. The negative RC filter is able to create the waveform to implement the first

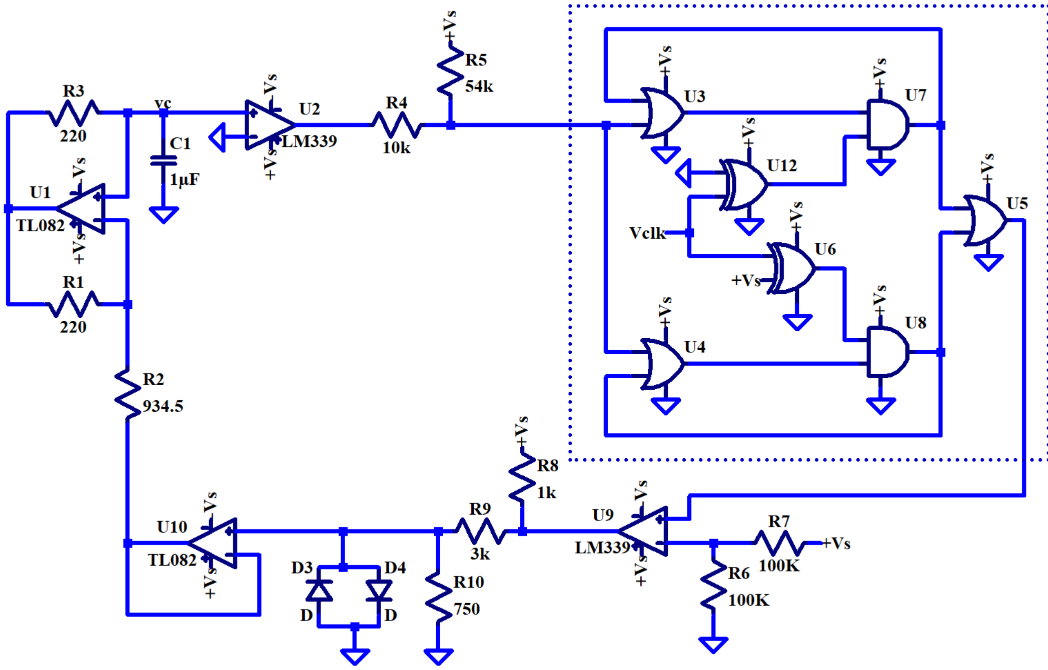

Figure 3.1: The complete circuit diagram implemented in LTspice.

order system when fed with appropriate square waves. The reset latch controls when and for how long the negative RC is fed with positive and negative voltages.

3.1 Negative Impedance Converter and RC

A Negative Impedance Converter (NIC) is used to reverse the direction of current flow. This changes Ohm's law relationship from $V = I \times R$ to $V = -I \times R$. The NIC has a finite region where the current direction is inverted, which is defined by the characteristics of the op-amp. A diagram of this device is shown in Figure 3.3. The impedance measured looking into the positive terminal of the op-amp is described as the negative ratio of the feedback resistors multiplied by the grounded resistor. Equation 3.1 is the formula for calculating the negative voltage relationship of a negative impedance converter and its derivation can be found in Appendix D.1:

$$\frac{V_{in}}{I_{in}} = -\frac{R_1}{R_2} R_3. \tag{3.1}$$

The negative impedance can be calculated by taking precise measurements of the resistors. $R_1 = 219.1\Omega$, $R_2 = 220.3\Omega$, and $R_3 = 934.5\Omega$. The theoretical resistance should be $R_{-} = -928.46\Omega$. This can be confirmed by taking a measurement of the NIC in isolation with a voltage sweep. The ideal RC charge time for the system to achieve full height is computed to be $644.56\mu s$. This corresponds to an input clock frequency of 775.72Hz. Unfortunately, the physical system is unstable close to a slope of two, so the frequency must be adjusted to lower the charge time. The system was stable at 850Hz. The theoretical slope of the map can be calculated from the known values of resistance, capacitance and period as shown in equation 3.2. The estimated symbol entropy in bits will be the base two logarithm of the slope. $H = \log_2(1.8825) = 0.9216$ bits. These calculations will be compared to measured values from the physical circuit:

$$\mu = e^{\frac{-T}{RC}} = e^{\frac{-588.25\times10^{-6}}{-926.2\times1.004\times10^{-6}}} = 1.8825. \tag{3.2}$$

By sweeping a range of voltages and measuring the current relationship, an impedance curve can be plotted. Figure 3.2 shows the IV curve of the NIC used in the circuit on a voltage sweep from -2.5 to +2.5 volts. As the voltage rises, there exists a region for which Ohm's law is inverted. The relationship between

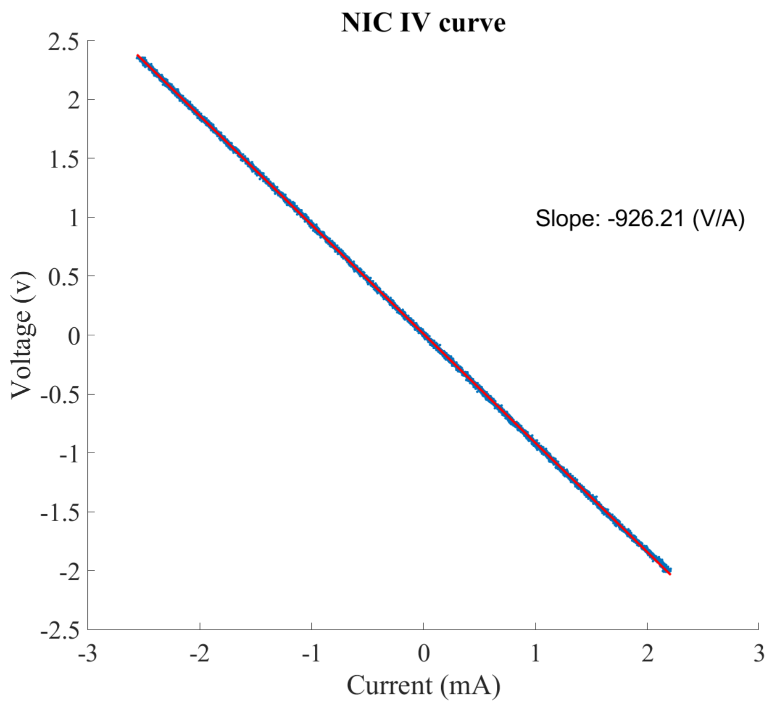

Figure 3.2: The characteristic curve of the Negative Impedance Converter. The circuit is swept from -2.5 to +2.5 volts as the current and voltage at the input terminal are measured. The current drops as the voltage increases. By approximating the slope of the IV curve, the impedance is measured as -926Ω.

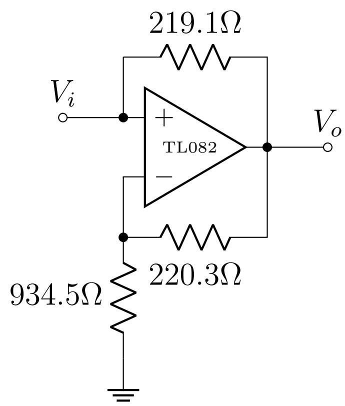

Figure 3.3: Negative Impedance Converter circuit diagram. Voltage is input at the non-inverting terminal. Resistors R_1 and R_2 in the feedback are equivalent. R_3 defines the positive resistance of the NIC. Voltage output at the output of the Op-amp.

current and voltage is reversed. As the voltage rises, the current falls. The slope of the IV curve in the inverted region is equal to the negated value of resistor R3. A polyfit curve in MATLAB estimates the measured slope to be -926.21Ω, which agrees with the calculated value.

By pairing the NIC with a single capacitor, it creates a negative RC filter. What results from this is a time reversed RC filter. A positive voltage input will cause the voltage on the capacitor node to drop exponentially. A negative voltage input on the negative resistor will cause the voltage to grow at an exponential rate. By feeding the negative RC with square wave pulses, it will alternate between charging and discharging, creating a negative saw-tooth wave. A simulation of this behavior is shown in Figure 3.4.

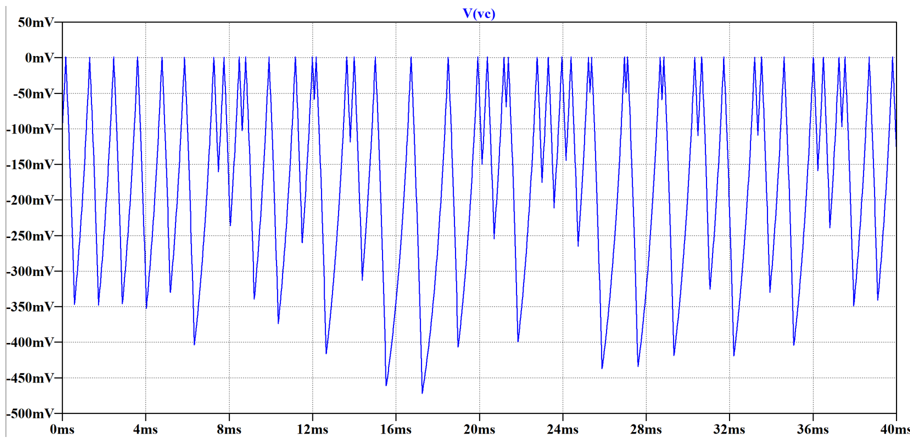

Figure 3.4: The voltage measured on the negative RC node when fed with square pulses symmetrical about 0v. The simulation shows a saw-tooth wave pattern made from the time-reversed RC charging and discharging.

3.2 Comparators and Other Analog Components

The sub-circuit in Figure 3.5 follows the negative RC filter. It is a comparator referenced to ground followed by a voltage divider. The comparator is powered by ±5v. The voltage divider acts as a pull-up for the comparator output. Instead of the low output being -5v, it is moved up to 0v. The high output is maintained at +5v. When the comparator detects the negative RC has reached 0v, it will output an instantaneous transition which will travel to the digital latch beyond the voltage divider. As the voltage on the negative RC node drops below 0v, the comparator will output the low state. The transition from low to high to low again should be almost instantaneous, creating a series of pulses. An example of the output from this sub-circuit is seen in Figure 3.6.

The sub-circuit in Figure 3.7 takes the output from the digital latch and prepares it to feed into the negative RC filter. Its output is the forcing function

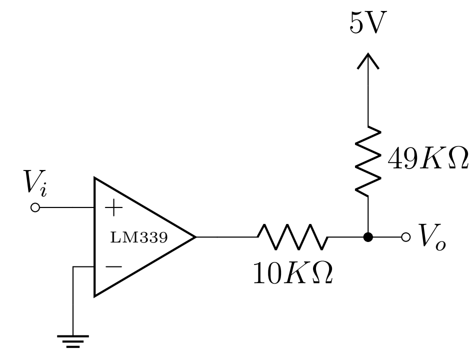

Figure 3.5: Comparator referenced to ground triggers to create an instantaneous transition from the low state (-5v) to the high state (+5v) when the voltage of the negative RC reaches 0v. Once the negative RC falls below 0v, the comparator returns to the low state. The voltage divider raises the minimum voltage output to 0v.

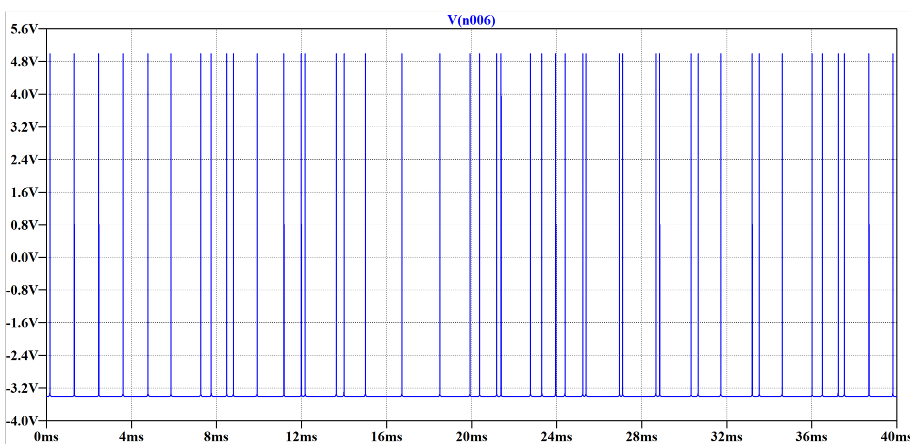

Figure 3.6: The output from the comparator circuit taken from an LTspice simulation. As the voltage from the -RC reaches 0v, the comparator outputs instantaneous transitions.

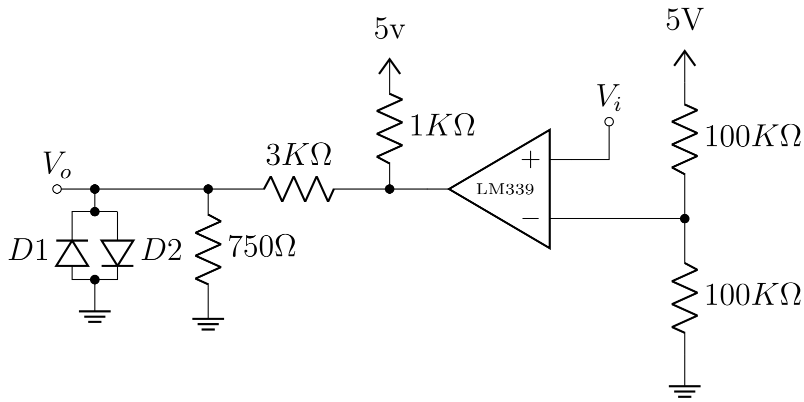

Figure 3.7: Comparator referenced to 2.5v to create symmetrical square pulses. The voltage divider provides the 2.5v reference to the inverting input to the op-amp.

of the tent map system. The comparator uses a voltage divider from +5v to ground as a 2.5v reference. The use of large resistors minimizes the current draw from the power supply. The op-amp comparator is powered by ±5 volts. As the digital signal comes into the non-inverting terminal, the comparator will output the same square wave stretched from 0-5v to ±5v. The forcing function needs to be a square wave symmetrical about 0. The following resistors are a voltage divider to lower the amplitude from 5v to around 1v. Lastly, the diodes further restrict the amplitude to the diode's forward voltage. Ideally, the diodes are fully matched to allow perfect symmetry in the forcing function, as seen in Figure 3.8. This does not actually occur with real components. The physical circuit implementation will have a small asymmetry in the forcing function.

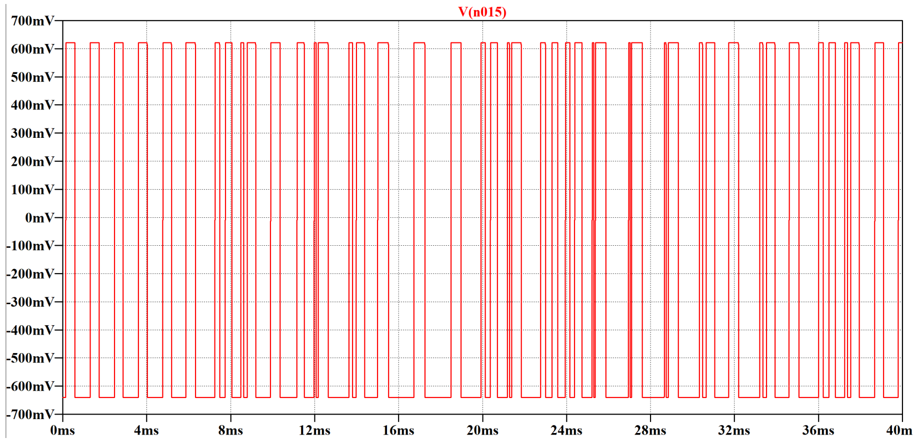

Figure 3.8: This is the forcing function of the -RC taken from an LTspice simulation. The waveform is perfectly symmetrical about 0, fixed in amplitude to the forward voltage of the diodes.

3.3 Digital Reset Latch

The digital reset latch is comprised of three OR gates, two AND gates and two exclusive-or (XOR) gates. All seven logic structures are two input gates. The design, shown in Figure 3.9, is symmetric about the XOR gates. The lower XOR takes the 5 volt, square wave clock input and a positive 5 volt input. The output of that XOR will be an inverted clock, such that the signal will be 5 volts when the input clock is low and 0 volts when the input clock is high. The upper XOR takes the clock as one input and ground as the other. This will buffer the clock so the clock and inverted clock remain in phase with one another. Above and below the XOR gates are an OR gate followed by an AND gate. The upper AND gate receives the clock as an input and the lower AND gate receives the inverted clock as an input.

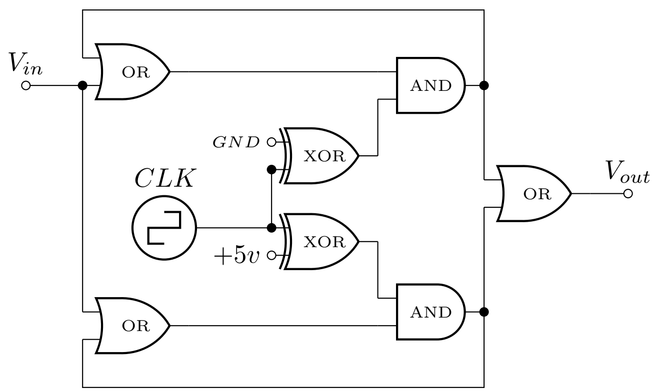

Figure 3.9: Detailed schematic of the digital reset latch.

The input to the reset latch comes from the output of the comparator at the negative RC filter. The input is in the form of an instantaneous pulse. The pulse is fed into inputs on two of the OR gates. Both OR gates will output a high signal for the duration of the instant pulse. The output then drives one input of the two AND gates. One of the AND gates will have a high input from either the clock or the inverted clock. Whichever AND gate has the high clock signal will create a positive feedback to its OR gate until the clock pulse changes to a low signal. Both AND gates also feed the last OR gate in the latch. When either AND gate is outputting a high voltage level it will be output from the latch and sent to a comparator. Figure 3.10 shows a visual representation of the mechanics of the latch.

This design controls when the capacitor on the negative RC filter is permitted to charge and discharge. Immediately, when the initial pulse is generated from the first comparator, the voltage polarity on the input to the NIC is reversed. The NIC is fed with a positive voltage, which allows the capacitor voltage to drop,

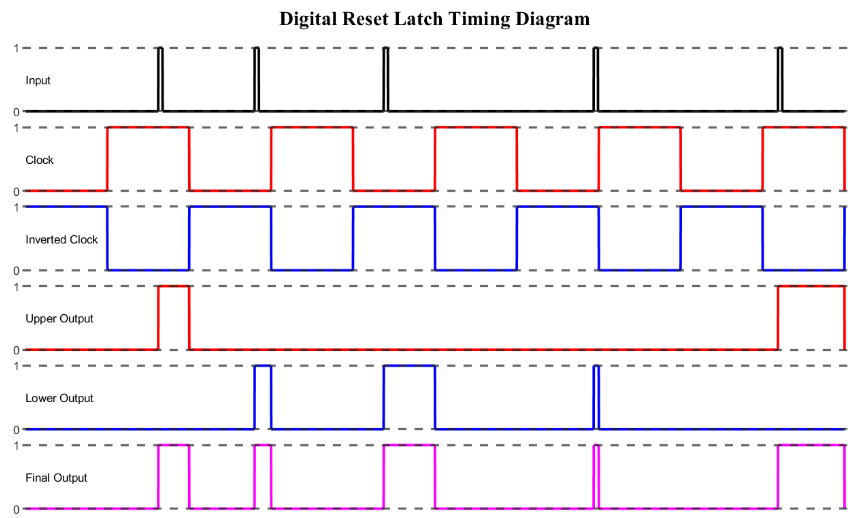

Figure 3.10: A timing diagram that shows the function of the reset latch. The top line (black) shows the input pulses from the comparator that measures for the guard condition on the -RC. Lines 2 and 3 (red and blue) are the clock and inverted clock signals which are inputs to the upper and lower AND gates respectively. Lines 3 and 4 (red and blue) represent the outputs of the two AND gates. These signals become high if their clock input is high when the pulse arrives and remains high until the clock's falling edge. The final output of the circuit (magenta) is the OR of the two previous signals.

until the clock cycle changes. After the clock cycle, the latch no longer outputs a high signal and the input to the NIC is a negative voltage. The capacitor charges until it reaches zero volts, triggering the comparator.

3.4 Power

Two external voltage sources are needed to operate the circuit. The comparators and amplifiers are powered by positive and negative 5 volt inputs. The digital logic components also use positive 5 volts for power. Both power inputs are filtered with three coupling capacitors placed between the voltage rail and ground. The values of the coupling capacitors are 10μF,1μF, and 100nF. These serve to maintain a consistent voltage and smooth power spikes. The total current draw of the circuit is 54 mA. The total power draw is 5V times .054A = 270mW.

Methodology

4.1 Data Capture and Processing

The breadboard circuit is powered by an external power supply and fed a square wave clock signal from a signal generator. An Analog Discovery 2 serves as an oscilloscope for data capture. By using the recording function in the Waveforms software to control the Analog Discovery 2, time series data from the negative RC node and the input clock signal were captured at a rate of 250kHz. Manual data capture was performed in 15 minute intervals six times and one ten minute interval. This provided 100 total minutes and over 11GB worth of data. The oscilloscope was set to the default configuration of averaged samples to help smooth out excess noise in the circuit.

The captured time series data are imported into MATLAB for processing. The code Data_Extraction.m in Appendix E, takes the time series data captured by the oscilloscope and writes two files with all of the sampled voltages from each clock edge and the binary symbol sequence. Voltage samples from the time series are taken one sample back from where the center of the clock transition is to account for a timing delay in the measurement between the clock signal and the negative RC node. The clock wave is centered at 2.5V. Each point taken at the

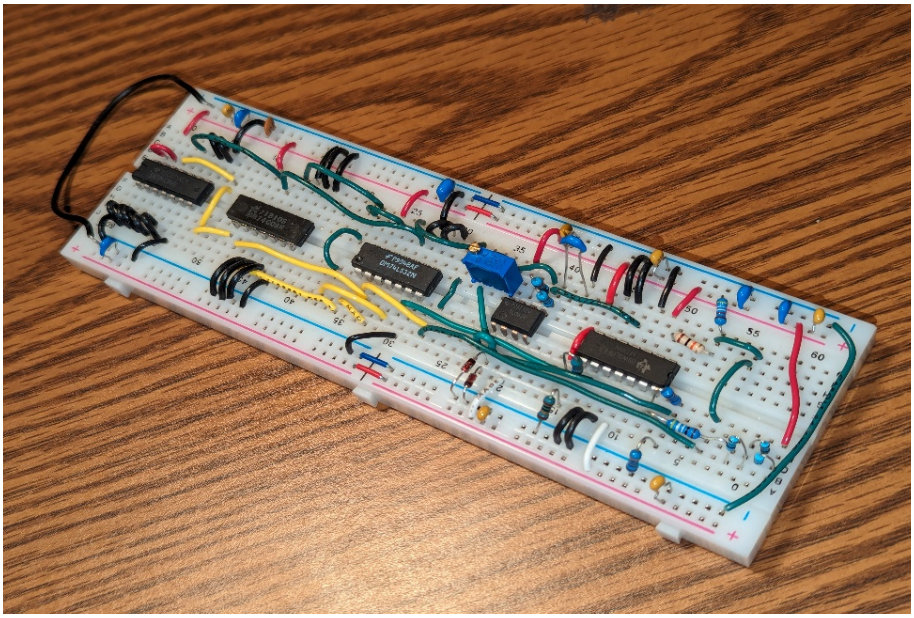

Figure 4.1: This is an image of the physical realization of the tent map as a circuit constructed on a breadboard.

rising and falling edge of the clock is written to a file. The output file of 10.2 million samples is reduced to less than 80MB.

An ideal mathematically modeled map will have a Markov point which partitions the map at one quarter of the driving voltage amplitude (Vs/4). The circuit models a shifted map. This map is the original map scaled and shifted so that the largest point is 0 and the lowest is around 0.7. The ideal Markov point for partitioning the map is -Vs/4. Although the map approximately models this system, a true Markov point cannot be achieved due to the nature of the physical system where the slope of the map cannot be tuned to precisely 2. Instead, a generating partition is chosen where the two slopes of the map meet. This is where the highest points on the Y-axis of the map are. MATLAB isolates the

100 largest values in the array of Y-axis points. Then the corresponding values in the array of X-axis points are found. These X-axis points will be centered about the partition. Averaging these values taken from the X-axis will give an accurate approximation of the partition point.

The discrete time sequence of values sampled from the circuit can be turned into a symbol sequence. MATLAB performs a simple logical operation where any values less than the partition point will be classified as a 0 and values greater than the partition are coded as a 1. Any symbols can be chosen to represent the data, but binary symbols are chosen for their usefulness in analyzing the symbol entropy and randomness of the system. The binary symbol sequence is written to a binary file for later analysis. The stored file of 10.2 million bits is less than 10MB. The unprocessed bit samples are needed for estimations of the symbol entropy, but, due to an underlying bias in the data, a whitening algorithm needs to be performed.

The Von-Neumann bit correction algorithm [31] is a method of whitening a set of binary data such that it has qualities closer to that of an ideal binary random variable. The algorithm works by looking at non-overlapping pairs of bits. Any bit pairs of identical bits, *i.e.*, '11' or '00', are completely removed from the dataset. Bit pairs of '01' are replaced with a single '0'. A bit pair of '10' is replaced with a '1'. An example of this method is shown in Figure 4.2. While this algorithm is very efficient, it also results in a data loss of approximately 75%.

Another method of improving uniformity is to utilize two independent bit streams. The two sets of data will be reduced to a single bit stream by performing

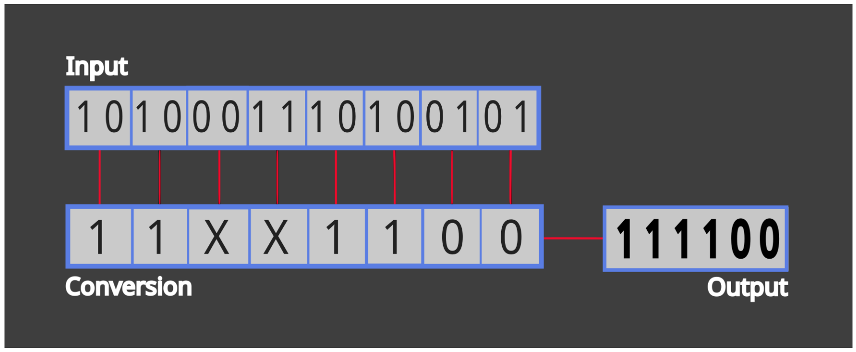

Figure 4.2: This diagram shows the method by which a sequence of bits is made more uniform by the Von-Neumann bit correction algorithm. A pair of the same bits is replaced by nothing. A pair of opposite bits is replaced by the leading bit of the pair.

a bitwise exclusive-OR operation. This method is 50% efficient and requires a second source of binary data. All tests to be performed on binary data will be run on the original and the whitened datasets.

4.2 Basic Statistics

The three easiest measures of the random bits are mean, variance, and standard deviation. An ideal uniform binary random variable will have a mean of exactly $\frac{1}{2}$, a variance of exactly $\frac{1}{12}$ and a standard deviation of $\sqrt{\frac{1}{12}}$. Close adherence to these measures should be seen in collected data.

Auto-correlation is the measure of how repetitious a dataset is. By shifting the dataset across itself and measuring the number of like values, it can reveal places where the dataset is similar to itself. An ideal random binary variable will have no significant self-correlation. The data will not have sequences that repeat over time. An auto-correlation plot will have a spike showing complete correlation when the data is not shifted, as every sample will be the same. For all non-zero shift values, the correlation should be near zero.

Uniformity is another graphical observation which will be made. The binary data will be grouped by an overlapping 12-bit window and converted from a binary number to a decimal. A histogram of the decimal values will be created using 100 bins. The random variable should be completely uniform for large datasets. Regions where there are significantly more or fewer counts in a bin will reveal underlying biases in the system.

4.3 NIST Statistical Tests

The National Institute of Standards and Technology is an organization under the US Department of Commerce with the purpose of creating standard methods for scientific research. Two of their standards will be utilized in order to evaluate the results. The first document, A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications (800-22), details 15 tests to be performed on a set of binary data [25]. Working MATLAB implementations of 14 of those tests can be found in Appendix E. Each test matches the results produced by NIST's own testing suite.

Each test measures a specific aspect of a random variable. They will reveal if the data has good uniformity, produces statistically likely runs, and has independence from other samples. Each test produces a statistic by which the ideal measurement is compared to and returns a P-value. The tests are performed on a 99% confidence interval using a significance level α of 0.01. When a test returns a P-value greater than 0.01, it is 99% confident the data provided came from a random sequence. When a test returns a P-value less than 0.01, it is 99% confident the data provided came from a non-random sequence. It is expected that 1 in 100 sequences tested to return a false result.

There are 6 tests with variable parameters to test under. Of those 6, I have changed 4 of them to better suit the size of data I have. The Block Frequency size has been increased to 2048. Approximate Entropy block length has been changed from the default of 10 to 12. The block length of the Serial Test has been increased from 16 to 18. The Linear Complexity test's substring length has been raised from the default 500 to 1000. All of these parameter changes are within the bounds defined by NIST 800-22 [25].

The data collected only feature 10 million bits sampled from the circuit. While this meets the minimum criteria for the tests, NIST recommends using the largest dataset possible and testing multiple subsections of the entire dataset. Performing tests this way is more robust and adds confidence to results showing where the NIST tests may give false results occasionally. For simplicity, I will be running the data through as a single dataset of 10 million bits. The tests will be performed on the original symbol sequence extracted from the circuit and the smaller whitened datasets to measure performance difference between them.

In Appendix E, there is MATALB code which implements 14 of the 15 statistical tests for randomness as defined in NIST 800-22. All but the Linear Complexity was successfully able to be recreated and tested against NIST's own available C coded program suite. NIST's suite was used for testing of the sampled binary data due its speed. However, the MATLAB code can be tailored to fit other datasets.

4.4 NIST Entropy Estimators

NIST provides another document helpful to the evaluation of entropy sources. NIST 800-90B, Recommendation for the Entropy Sources Used for Random Bit Generation, details several tests to quantify the performance for comparison to other entropy sources, and methods of estimating the symbol entropy of a set of binary data [30]. I have implemented 5 of the entropy estimators provided in MATLAB. They are, The Most Common Value Estimate, The Markov Estimate, The Tuple Estimate, The Longest Repeated substring Estimate, and the Most Common Window Estimate.

NIST recommends running as many estimators as possible on a sample of at least 1 million bits [30]. The lowest observed symbol entropy estimation should be taken as the minimum entropy per-symbol of the system. The estimations are expected to return values below the base 2 logarithm of the tent map slope, which is the maximum entropy per-symbol the system can produce. This will indicate a region between the minimum and maximum symbol entropy where the system's true entropy lies.

To supplement these estimations, a block entropy measurement will be taken. This entropy calculation uses Shannon's symbol entropy formula, equation C.1. Instead of single bit word lengths, I use 16-bit non-overlapping words. Since Shannon's entropy formula relies on the frequency of sequences, using longer blocks will reveal when certain bit patterns occur less often. Shannon's formula works ideally when you can sample from an infinitely large bit sequence with an arbitrarily large block size. A trade off to using a larger block size is that it needs more data to accurately measure entropy.

In Appendix E, 5 entropy estimators from NIST 800-90B have been implemented in MATLAB code. These programs have perfectly implemented their algorithms and have been tested against their example data. Each program takes in an array of binary data of arbitrary length and computes the entropy estimation. These MATLAB programs are used to estimate the entropy of 10.2 million bits sampled from the chaotic oscillator.

Results

The analysis of the results from the data collection will be broken into several sections to measure performance across different qualitative and quantitative metrics. Behavior of the system will be observed graphically and compared to prior estimated values. The performance of the binary symbol sequence derived from the sampled output will also be evaluated in terms of entropy measurements and with statistical tests for randomness. By estimating the system's entropy, it can be proven from first principles that the chaotic oscillator is entropic at a predictable, measurable rate. NIST statistical tests will verify the output as likely to come from a random sequence.

5.1 Time Series

Primary operation of the physical circuit is validated by comparing the captured time series data to that of a spice simulation. Under the same operating parameters, the waveforms exhibit generally the same shape as shown in 5.1. The physical circuit reaches lower voltages than simulation. There are some aspects of the physical circuit that spice simulation is not accurately representing. These conditions could be noise interference, temperature changing operating

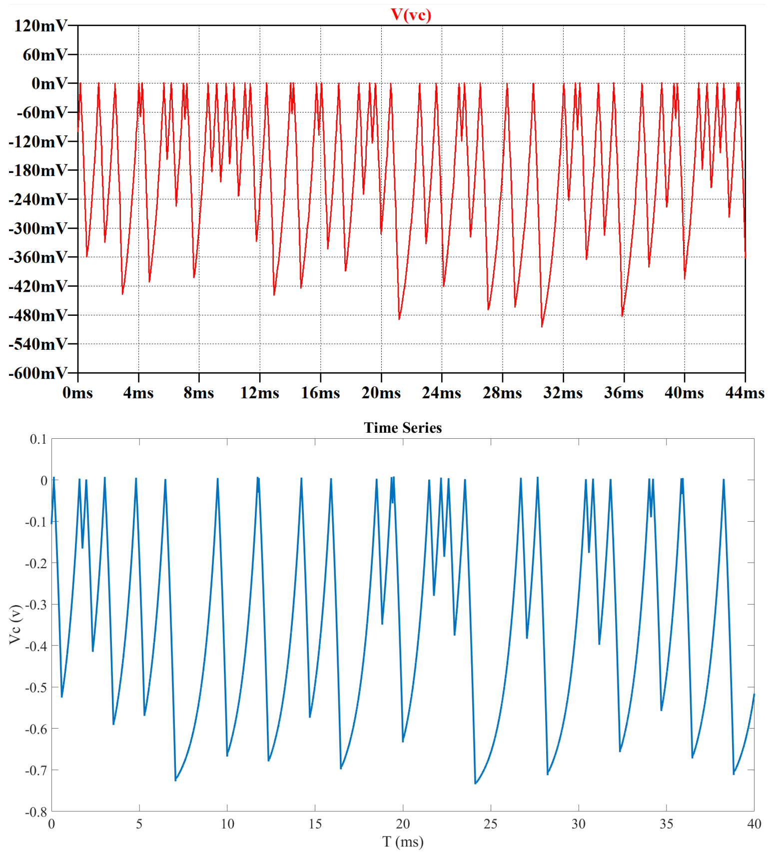

Figure 5.1: Upper: Time series of the negative RC from LTspice simulation. **Lower:** Time series data capture from the negative RC of the physical circuit.

characteristics of components, or manufacturing variations in components. Their performance is identical as possible and the physical circuit is fully functional.

5.2 Negative RC Voltage Measurements

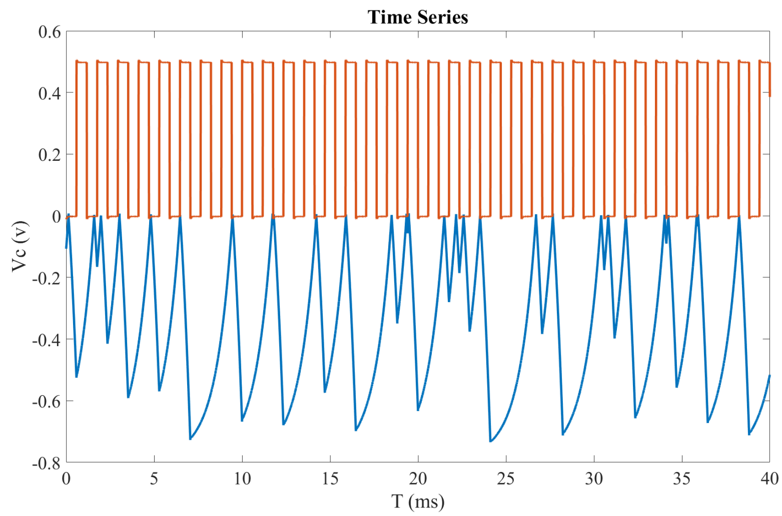

Figure 5.2: The real-time captured time-series of the oscillator's output sampled at the negative RC node.

Figure 5.2 plots a small sample of the data captured from the oscillator circuit. At each rising and falling edge of the clock signal in orange, the chaotic wave in blue will be sampled. This data comes from the first 900 seconds of the data comprising the 10.2 million samples used to generate a random bitstream. Each sample is an iterate of the tent map system and will be used to construct a one-dimensional return map. These samples will also be converted to the binary symbol sequence for further analysis.

The generated one-dimensional return map plots the trajectory of the map. The resultant map of the collected data is shown in 5.3. This map strongly resembles the idealized map. Both branches are straight and close to full height.

The left branch features a small restriction in value. It should reach all the way down to -0.75v, but falls short. On the right, there is a small nonlinear portion near 0. This feature results from an imperfect comparator reference causing the latch to trigger at slightly different values, as well as sample timing issues discussed prior. Slight timing mismatches in the sampling of the waveform near zero, where the growth and decay is most significant, causes more dramatic errors than mistimed samples elsewhere. More precise timing and data capture methods may help reduce this effect.

The map partition is calculated from an average of the x-axis location of the 100 highest points on the map. This places the partition as close as possible to the true generating partition. The true generating partition would be 0.375 exactly halfway between the lowest value and 0. The measured partition is calculated to be 0.3748.

The polyfit function in MATLAB performs a slope approximation for the two branches after partitioning. The left side slope is computed as 1.9299 and right slope is computed to be 2.0002, as shown in Figure 5.4. The growth rate cannot possibly be more than 2, so this number is likely a slight overestimation. For future calculations, the smaller of the two slopes is assumed to be the true growth rate.

A key property of chaotic systems discussed earlier is divergence from sensitivity to initial conditions. By searching through the symbol sequence to locate the longest sequence of bits that occurs twice, the corresponding sections of the times series reveal two waveforms that seem to converge at a single point

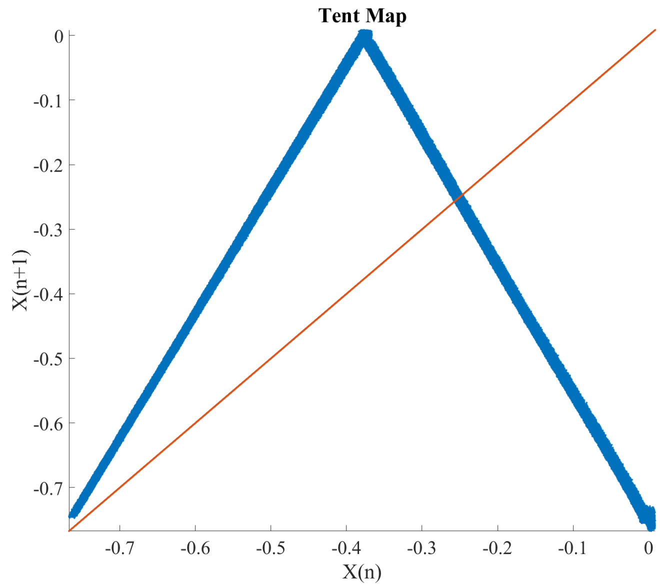

Figure 5.3: Tent map generated from the sampled points captured from the physical circuit tuned to the maximum height, or the lowest frequency.

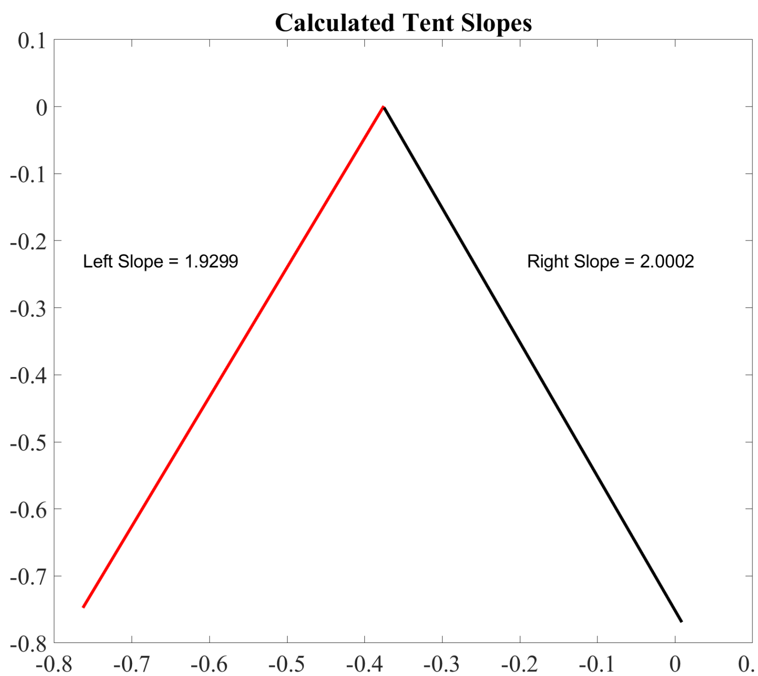

Figure 5.4: Slopes of the map's branches calculated from a polyfit curve in MATLAB.

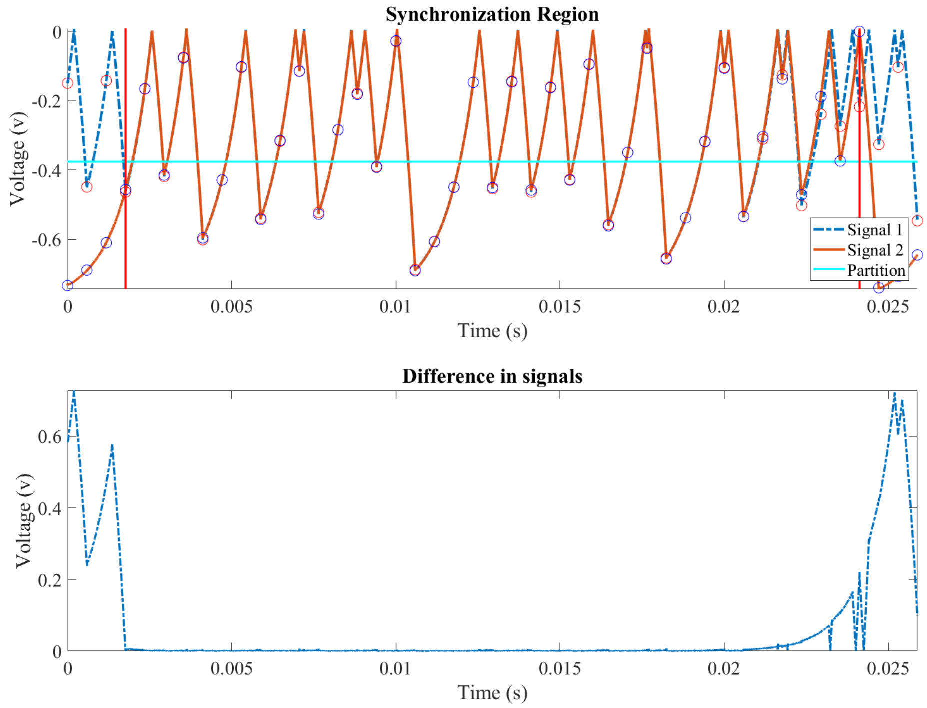

Figure 5.5: Upper: A region of synchronization plotted for two sections of the time series with the longest matching symbol sequence. Each sampled point is shown as a hollow circle. The threshold is represented as a cyan line. The beginning and end of the region where the symbols match are denoted by the red vertical bars.
Lower: The difference of the two waves showing the exponential divergence of two points within the neighborhood of one another.

and then diverge over time. Figure 5.5 shows these two waves overlapped and the difference between them.

Figure 5.6 is a histogram of all the samples that comprise the map. It shows high uniformity of samples except for the most negative values. There is a roll off in the lowest bins as the circuit struggles to produce samples near the bottom edge of the forcing function. This result is expected and consistent with other measurements. Since the slope is not perfectly tuned to 2, there is a bias

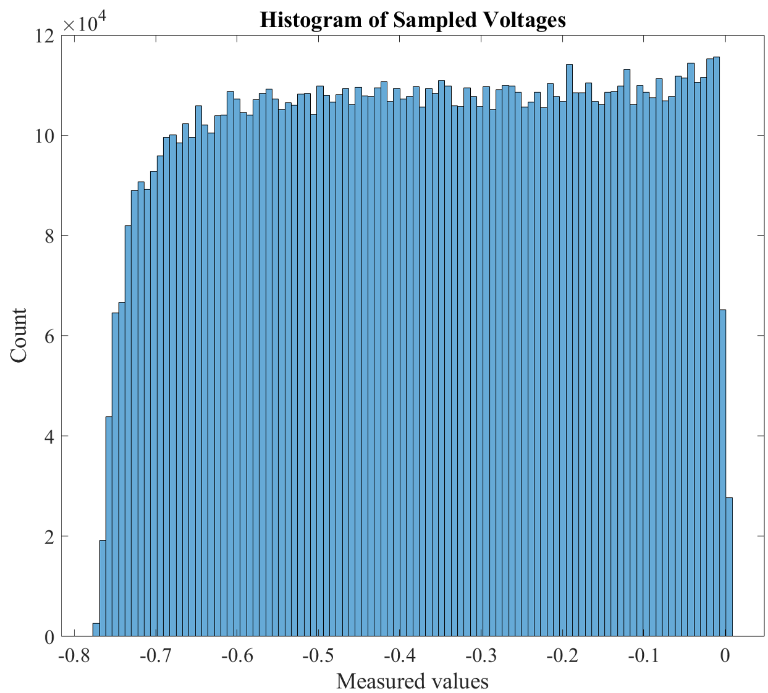

Figure 5.6: Histogram of voltage samples taken from the circuit on every clock edge.

away from the lower fixed point. That is what is observed in the histogram of voltage samples. This indicates that there may also be areas of poor uniformity in the symbol sequence.

To establish a link between the characteristics of an ideal map and the hardware, a rudimentary bifurcation diagram from sampled data is constructed. This is done by de-tuning the oscillator to lower, pre-calculated slopes and capturing 1 million clock edges worth of data. The oscillator was made to operate at slopes decreasing from the maximum possible slope of 2 by steps of 0.05 down to 1.05. Results should show a decreasing range of values the system can produce as the slope is reduced. Further, for slopes below 1.41, there should be a gap in the continuity of values seen in both the bifurcation diagram and the one-dimensional return map for a given slope.

This test was performed with distinct real-world data taken from the physical implementation of the chaotic oscillator and compared to a synthetic simulation in MATLAB. The MATLAB simulation was created from 5 million iterates of an initial condition between 0 and 1 at each slope. The resultant plots of the bifurcation diagrams in Figure 5.7 show expected behavior. The synthetic data from MATLAB strictly follows the ideal map shown in Figure 2.4. The measured data, while similar, has a few differences worth noting. Most strikingly, the possible values are only restricted from one side. The range of values shrinks from the negative side towards zero. This is a symptom from the mechanics of the circuit. In order to lower the slope of the map by restricting the growth rate, the frequency is adjusted higher. Higher frequencies on the clock input reduce the time the negative RC is permitted to charge and discharge. The ground reference on the comparator after the negative RC does not change. The circuit will always reach 0v before falling over the period one half clock cycle. The measured data do still feature a discontinuity below a growth rate of $\sqrt{2}$. The location of the discontinuity moves upwards towards zero with the rest of the data, unlike the theoretical data.

Table 5.1: Table containing slope decay measurements. The values are taken for fixed values of R and C at 2.01kΩ and 1uF. Raising the input clock frequency lowers the RC charge time and the slope of the map. Error between the measured and theoretical slope grows for higher frequencies.

Theoretical Slope	Frequency	Measured Slope
1.95	366.85	1.9429
1.90	381.70	1.8925
1.85	398.24	1.8428
1.80	416.81	1.7851
1.75	437.79	1.7341
1.70	461.70	1.6823
1.65	489.23	1.6328
1.60	521.26	1.5739
1.55	559.02	1.5137
1.50	604.23	1.4569
1.45	659.36	1.3991
1.40	728.12	1.3474
1.35	816.36	1.3039
1.30	933.79	1.2548
1.25	1097.91	1.2004
1.20	1343.74	1.0922
1.15	1752.93	1.1135
1.10	2570.48	1.0335

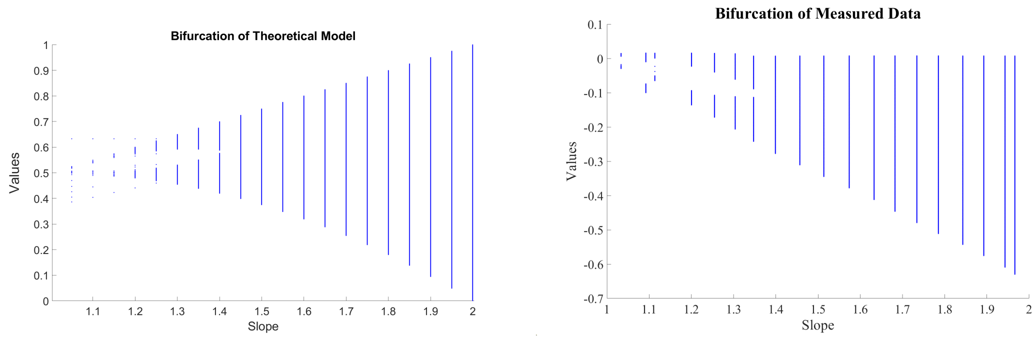

Figure 5.7: Left: Bifurcation of the tent map at fixed slopes given from 5 million samples per slope generated in MATLAB.
Right: Bifurcation of the tent map at fixed slopes given from 1 million samples per slope taken from physical hardware.

Figure 5.8 shows these effects of lowered slope on the one-dimensional return map. Observed behavior matches with results from the bifurcation. Lowering μ causes the tent slopes to lower as well. Additionally, the tent map's left branch shortens away from the fixed point located at the minimum value of the ideal map. Once the slope drops below $\sqrt{2}$, a discontinuity appears in the map on the right branch. Points are pushed away from the upper fixed point.

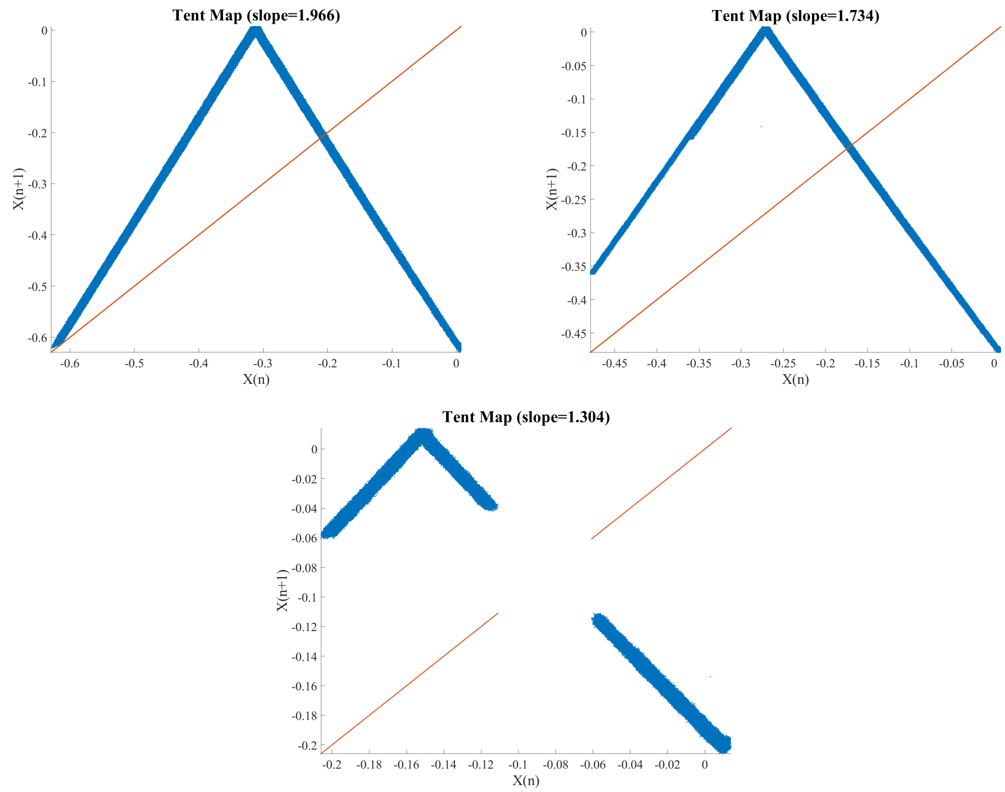

Figure 5.8: This figure shows how lowering the slope of the tent map affects its return mapping. Initially, the lower slope alters both branch slopes and the left branch's length. When $\mu < \sqrt{2}$ the map begins deteriorating on the right branch about the fixed point.
Top-Left: Tent map measured at a slope of 1.966.
Top-Right: Tent map measured at a slope of 1.7314.
Bottom-Middle: Tent map measured at a slope of 1.3034.

The behavior of systems with less than ideal slopes match the predicted behavior from simulations. As the slope is lowered, the tent map experiences restrictions on the values it can take on. The hardware implementation sees grammar restrictions starting with the left slope at the lower fixed point. This will impact the symbolic dynamics which govern the bits generated. The information content of the bits from lower slope maps are tested in the next section.

5.3 Entropy Estimations

The primary characteristic to care about as a measure of uncorrelated bits is the entropy of the system that is producing them. From theory, the entropy of an ideal, full height tent map is directly related to the Lyapunov exponent given by the natural logarithm of the slope. The relationship of symbol entropy (in nats) to the Lyapunov exponent should hold for maps with non-ideal slopes. If so, we can make estimations of the maximum symbol entropy of the system operating at a given growth rate. To test this behavior the same data was used from the bifurcation diagram. 1 million samples from a physical circuit and 5 million samples generated in MATLAB. At each measured slope, the entropy was estimated using 3 algorithms from NIST [20]. Plotting the symbol entropy rate estimates against the measured slope shows trends between the growth rate and the entropy per-symbol in Figure 5.9. There are several pieces of relevant information to glean from these plots alone.

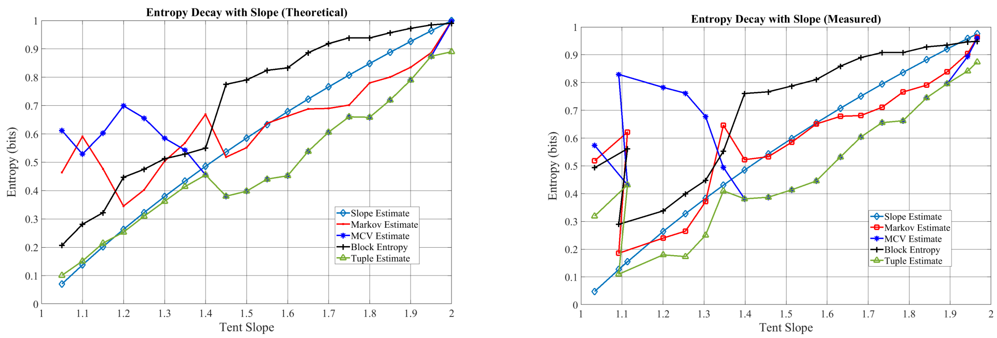

Figure 5.9: The relationship between the slope of the tent map and symbol entropy estimations of the symbol sequence.
Left: The theoretical model, made from 5 million samples for each slope, was generated in MATLAB.
Right: The real-world data taken from the chaotic oscillator circuit with 1 million samples per slope. The data shows agreement between the model and real system with a linear relationship for slopes at and above $\sqrt{2}$.

Firstly, using the base 2 logarithm of the slope as an estimation of the maximum symbol entropy rate, all three estimators from the NIST documentation fall below that line. What seems like an outlier is the Block entropy, which is calculated from Shannon's formula with a block size of 16 [26]. This measure is not congruous to the other entropy rate estimations. The tent map at lower slopes is no longer a first order Markov process. It takes on a much higher order with longer correlations in the samples. To overcome this, the Shannon block entropy should be calculated with a block size equal to or greater than the Markov order. As the block size increases, the block entropy will converge to the entropy rate of the system. Ideally, Shannon's formula is used on an infinitely long sequence of data with an arbitrarily large block size. Using a larger block size to account for longer words has the drawback of needing magnitudes more data than what is collected. The limited data will cause the entropy estimator to converge to near zero as there are not enough unique blocks and the test will indicate that there are grammar restrictions for words that can occur, but haven not been observed in the tested dataset. On the other hand, using too small of a block size is also an issue. When a smaller block size than the Markov order is is used, grammar restrictions that only occur for larger block sizes will not be noticed and the block entropy estimation will be larger than the true entropy rate. For the tests here a block size of 16 was chosen as a balance between a more representative measure without needing far more data to be collected. A graphical justification is shown in Figure 5.10. A block size of 16 is not greater or equal to the Markov order, so this measurement should be kept in the context of other block entropy measurements of block size 16 and not taken as a measure of the entropy rate.

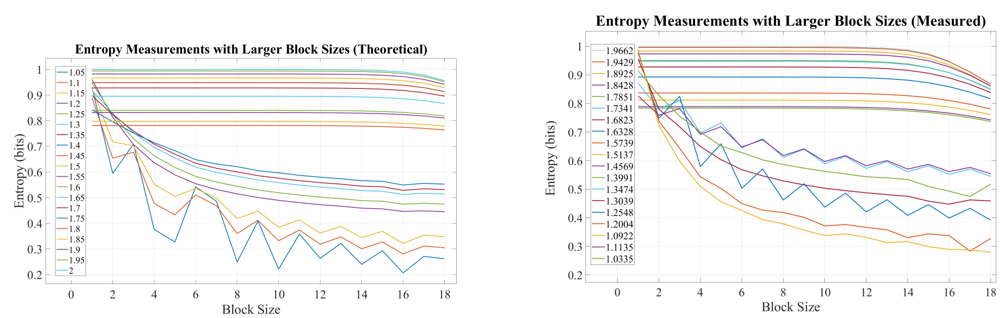

Figure 5.10: Block entropy calculated with Shannon's formula evaluates entropy by measuring the frequency of sequences of a certain fixed block length. Smaller block lengths tend to overestimate the true entropy rate. Larger block lengths do not observe all possible sequences the system may produce. As a balance, a block length of 16 was chosen to be more accurate than smaller blocks, but not lose information compared to larger block sizes.

The other feature that is obviously apparent is that the estimations break from the linearly decreasing trend starting with the sample at a slope of 1.4.

This phenomenon is explained by the Bifurcation diagram 2.4. The tent map's bifurcation diagram shows a region from 1 until $\sqrt{2}$ where there is a large gap in the values that can occur. For slopes in this region the tent map constructed also has a discontinuous region, shown in Figure 5.8 and the symbolic dynamics breakdown in unpredictable ways. The entropy estimators can no longer make correct predictions about the entropy of the system. Entropy estimations from this region should not be used to characterize the system. Taken as a whole, for regions where the slope of the map is greater than $\sqrt{2}$, there is a direct relationship between the symbol entropy of the system and slope of the map. We can assume that the true entropy rate of the system for any slope in this region lies between the base 2 logarithm of the slope and the minimum estimation.

Table 5.2 shows the symbol entropy estimations for the data 10 million bits taken from the oscillator. As predicted, the Block entropy is an overestimate of the true entropy per-symbol which should be bounded by the slope estimate of 0.9495. The lowest estimator was the Tuple estimate at 0.8755. We should assume the actual symbol entropy lies between those values. The Block entropy is an overestimation. Without a much larger sample of data, it will not approach the true entropy of the system. Increasing the block size above 16 returns inaccurate measurements since the full grammar of the system cannot be captured.

These results prove the system is a predictable and measurable high quality source of entropy. The system is close to reaching the maximum possible entropy for a binary system. We can also know the entropy beforehand from the relationship between the tunable growth rate and the entropy. This greatly improves on

Table 5.2: Table containing entropy estimations. The slope estimate is calculated as the base 2 logarithm of the calculated or lowest measured slope. Shannon entropy is calculated from bit sequences with different block lengths. The longer the block length, the closer the entropy gets to the true value. All other estimators are taken from the NIST 800-90B document. The lowest value is taken as the minimum symbol entropy of the system.

Entropy Estimator	Entropy Value (bits)
Ideal	1
Slope Estimation (Calculated)	0.912650
Slope Estimation (Measured)	0.949498
Shannon Entropy (Block Size 16)	0.9947
Markov Estimate	0.917802
Most Common Value Estimate	0.937776
Most Common Window Estimate	0.941281
Tuple Estimate	0.875501

other systems that rely on tenuous methods of extracting bits that do not have entropy rates that can be known with much certainty [9]. With improvements to the design, it may be able to reach higher entropy rates through more precise tuning.

5.4 Statistical Binary Measurements

The binary data will also be evaluated based on its statistical properties and adherence to a uniform binary variable. These tests are performed three times. Once on the original bit sequence, again on bit sequences after correction from the Von-Neumann whitening algorithm and once more after an XOR operation.

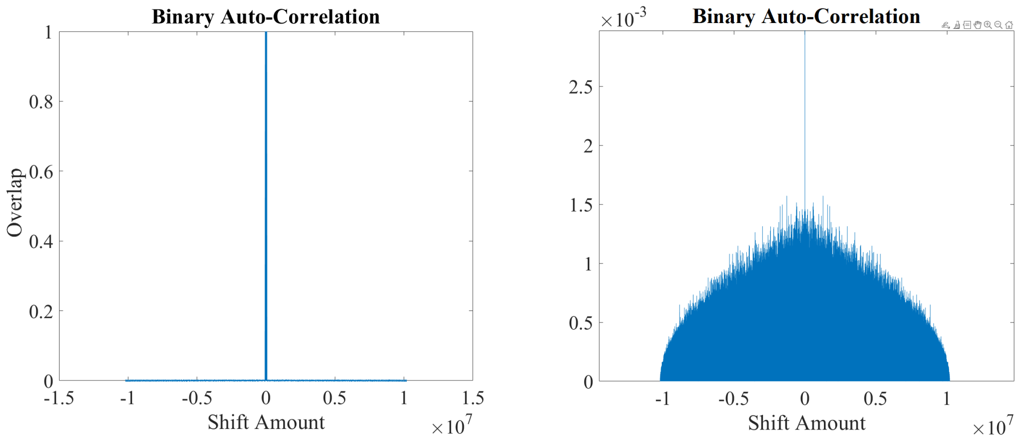

Figure 5.11: Left: Auto-Correlation plot of the binary dataset from the symbol sequence extracted from the oscillator.
Right: A zoomed in picture of the Auto-Correlation plot to better show relevant data.

The auto-correlation plot of the original voltage samples indicates there are no extreme correlative peaks, as seen in Figure 5.11. Observing the complete image shows all correlations shifted in either direction are close to zero. The largest shifted correlation only showed a similarity of just over 0.15% or 15300 bits similar within 10.2 million bits. More detailed measures will better quantify the independence of the data.

The first comparative measure of the binary datasets will be their uniformity and basic statistical properties. As outlined in the methodology, an overlapping 12-bit window, with the leftmost bit being the MSB, is used to convert the binary sequence into decimal values. The decimal values are in the region of 0 to 1. Figure 5.12 graphically shows how the uniformity changes after correction. As predicted, the samples taken directly from the symbol sequence have poor uniformity in sequences that require long runs of 0s. Most obviously, near 0 and at 0.5, 0.75, and 0.875. These bins need values with a leading one, or ones, followed by zeros for the remainder of the 12-bit sequence. From the tent map, we observed the left branch slope to be both lower and shorter than the right. To maintain stability in the system, the oscillator must be tuned away from the minimum bound. This minimum value is also a fixed point in the system. The tent map produces long runs of symbols only when it reaches a state in the neighborhood of a fixed point. Iterates will grow away from the fixed point over time, but will produce a run of the same symbol. Since the system cannot get as close to the lower fixed point, it will not produce long runs of 0's.

Performing either bit correction method greatly improves the uniformity. Between the two methods, XOR bit correction seems to have better uniformity while retaining twice as much data. Looking at the measurements in Table 5.3 confirms this. Overall, XOR bit correction returns a mean and variance closer to an ideal uniform random variable. Von-Neumann correction only gives slightly worse results within 1% difference. This is not significant enough to determine whether one method is objectively better than the other.

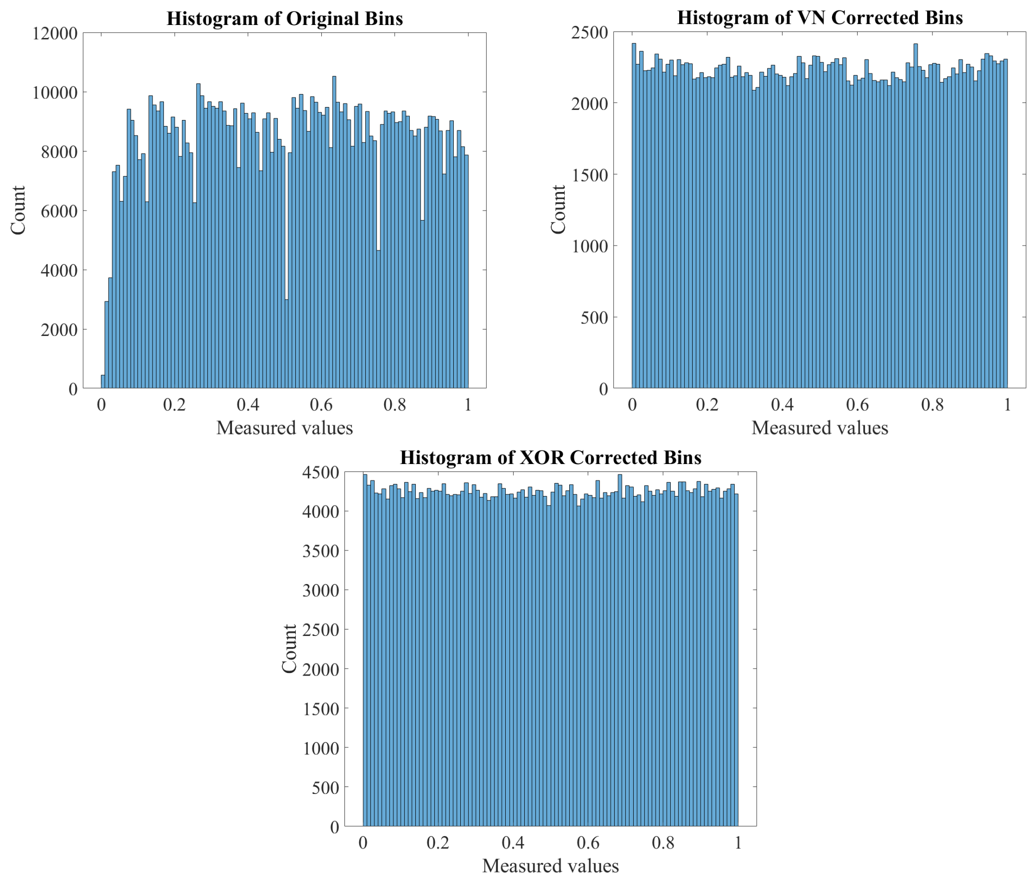

Figure 5.12: These three graphs show the distribution of values for decimal converted values of 12-bit binary sequences.
Top-Left: 10.2 million samples from the circuit's symbol sequence.
Top-Right: 2.68 million samples remaining after Von-Neumann correction.
Bottom: 5.1 million samples remaining after XOR bit correction.

While it is good to acknowledge the tests that succeeded, it is more important to consider the tests that failed and why. The original bit sequence, before any correction, fails nearly all statistical tests. It is clear to see that there are issues in distribution uniformity as shown in Figure 5.12. An ideal random variable will have a completely uniform distribution of values. Each state should have an

Table 5.3: The measure of mean, variance and standard deviation for three bit sequences compared to an ideal uniform random variable. Original data consists of 10.2 million bits. Von-Neumann corrected data has 2.68 million bits. XOR corrected data is 5.1 million bits. These measures were taken from decimal converted values of 12-bit overlapping blocks.

	Mean	Variance	Standard Deviation
Ideal	0.500000	0.083333	0.288675
Original	0.512230	0.077599	0.278566
Von-Neumann	0.499745	0.084173	0.290126
XOR	0.499901	0.083715	0.289336

Table 5.4: Table containing statistical test results of the original 10 million bits.

Test Name	P-Value (Original)	Result (Original)
Frequency (Monobit)	0.00000	**FAIL**
Frequency Within a Block	0.00000	**FAIL**
Runs Test	Incomplete	**FAIL**
Longest Run of Ones in a Block	0.015976	PASS
Binary Matrix Rank	0.900518	PASS
Discrete Fourier Transform	0.00000	**FAIL**
Non-Overlapping Template Matching	Incomplete	**FAIL**
Overlapping Template Matching	0.00000	**FAIL**
Maurer's 'Universal Statistic'	0.00000	**FAIL**
Linear Complexity	0.687602	PASS
Serial	0.00000, 0.000018	**FAIL**
Approximate Entropy	0.00000	**FAIL**
Cumulative Sums	0.00000, 0.00000	**FAIL**
Random Excursions	Incomplete	**FAIL**
Random Excursions Variant	Incomplete	**FAIL**

equal probability of occurrence. After correction, the data is much closer to an ideal uniform distribution. As a result, all but three tests fail to indicate the data was from a non-random sequence.

The underlying biases and grammar restrictions cause the two Frequency tests to fail on grounds of uneven distributions of 1s and 0s within large and small blocks of the dataset. The result from the Discrete Fourier Transform test shows that the data has some underlying spectral content which indicates sequences can repeat when given enough time. The Random Excursions tests cannot be completed at all. They require a random walk that crosses the origin at least 500 times within a sample. Testing shows there are only 50 zero crossings in the entire 10 million bit dataset. The bias towards 1s over 0s prevents data from cycling evenly. The Runs test also refuses to execute as the distribution of 1s and 0s is not close enough to a uniform distribution.

The only front where the tests indicate positive results towards randomness are Linear Complexity, Binary Matrix Rank and Longest Run of Ones in a Block. The Longest Run of Ones in a Block test likely passes as the system tends towards longer runs of ones. If the test was repeated on the longest run of zeros, it would surely fail. The two other tests measure statistical independence between bits and sequences. As the system is highly entropic it is not surprising to see results that indicate strong independence.

Table 5.5 shows the results from the tests performed on the Von-Neumann corrected data of 2.6 million bits. 13 of the 15 tests indicate strongly that the measured data came from a random sequence. More interestingly, are the three

Table 5.5: Table containing Statistical Test Results from the binary sequence after Von-Neumann correction.

Test Name	P-Value (Corrected)	Result (Corrected)
Frequency (Monobit)	0.343843	PASS
Frequency Within a Block	0.931430	PASS
Runs Test	0.000000	**FAIL**
Longest Run of Ones in a Block	0.548034	PASS
Binary Matrix Rank	0.163638	PASS
Discrete Fourier Transform	0.101155	PASS
Non-Overlapping Template Matching	0.179838	PASS
Overlapping Template Matching	0.000763	**FAIL**
Maurer's 'Universal Statistic'	0.300869	PASS
Linear Complexity	0.752528	PASS
Serial	0.425430, 0.883963	PASS
Approximate Entropy	0.000071	**FAIL**
Cumulative Sums	0.421252, 0.511154	PASS
Random Excursions	0.47 0.72 0.87 0.75 0.91 0.59 0.37 0.31	PASS
Random Excursions Variant	0.29 0.20 0.31 0.30 0.17 0.18 0.20 0.14 0.16 0.60 0.41 0.17 0.11 0.18 0.34 0.36 0.21 0.14	PASS

tests that still reject the null hypothesis and indicate the sequence is not from a random sequence.

The Runs test fails when the distribution of runs of a given set is not statistically appropriate for a random binary variable. This is specifically an area where humans are not good at representing random data. A human tasked with saying a random sequence of bits will often switch between the two states often, only letting runs go on for 3 or 4 bits. Saying the same bit over and over does not feel random, but for an extremely large sample of data, consisting of millions or billions of bits, it should be expected that long runs of occur. If there are too many runs found in a sequence, then the data oscillates too quickly with smaller run lengths. If too few runs are observed, the data tends to stay in long runs more than would be probable to observe. For the data tested here, too many runs were found, indicating that there are more shorter runs than expected. Perhaps the Von-Neumann algorithm is not robust enough to condition the data well. It is possible that the use of a different algorithm or the XOR method may yield better results on this test.

The Approximate Entropy test also fails. The way its test statistic is computed, it is incredibly difficult to pass. The test calculates what it calls the Approximate Entropy of the data in nats. A difference is taken between the maximum entropy, $ln(2)$, and the computed Approximate Entropy. This makes sense, low entropy would be an indicator of poor randomness. However, the test statistic is not merely this difference. The test statistic is the difference of the two values multiplied by the number of bits in the sequence tested. Given that the test requires over one million bits to be significant, the Approximate Entropy cannot be more than 0.000005 off from the ideal value without failing the test. This problem worsens for testing larger datasets. The test encourages the use of cherry-picked, small data sets to pass, even when a larger sample of data from the same source might not. For the tested data, the calculated Approximate Entropy was 0.692317. That is very close to the ideal, but the difference is blown up by the number of bits tested. The data needed to pass this test needs to be maximally entropic with high uniformity in the distribution of data.

The Overlapping Template Matching test returns a P-value greater than zero, but not by much. This test searches for the frequency of patterns with an overlapping window [25]. A weak result from this test indicates that the distribution is still not close enough to a uniform random binary variable to be considered statistically random.

The results from the XOR post-processing are seen in Table 5.6. Similar to the Von-Neumann corrected data, this sample passes Most of the NIST tests with 5.1 million bits. Notably, the Approximate Entropy Test is passed, which the other datasets particularly struggled with. The Runs test still fails outright, but produces a larger p-value indicating the data is more fitting of the statistical ideal. This tracks with the improved mean variance, and distribution over the original sample and the Von-Neumann corrected data.

Both Excursion tests failed on the grounds that there were an insufficient number of crossings to perform the analysis. The data here likely still contains long runs that need longer to cycle back down to an even number of 1s and 0s.

Table 5.6: Table containing Statistical Test Results from two binary sequences after an XOR operation.

Test Name	P-Value (Corrected)	Result (Corrected)
Frequency (Monobit)	0.456927	PASS
Frequency Within a Block	0.214911	PASS
Runs Test	0.000038	**FAIL**
Longest Run of Ones in a Block	0.892183	PASS
Binary Matrix Rank	0.589465	PASS
Discrete Fourier Transform	0.666666	PASS
Non-Overlapping Template Matching	1.00000	PASS
Overlapping Template Matching	0.734506	PASS
Maurer's 'Universal Statistic'	0.717100	PASS
Linear Complexity	0.499607	PASS
Serial	0.397615, 0.907350	PASS
Approximate Entropy	0.476261	PASS
Cumulative Sums	0.117542, 0.502430	PASS
Random Excursions	Incomplete	**FAIL**
Random Excursions Variant	Incomplete	**FAIL**

Since the two binary datasets were taken from the same oscillator it is possible that the resulting data carries some of the same biases, like long runs of ones. A longer collection of data is needed to evaluate this property with significance.

I consider this data the best of the three. It returns one weak result in the Runs test and two incomplete tests. With enough data, the Excursion tests may pass, leaving only the Runs test to indicate the data may not be suitably random. That is a great improvement over the three failures from the Von-Neumann Correction and the 12 failures from the original data. Considering the improvements both methods of data correction had, it would be interesting to see how implementing both correction methods together would impact results.

Conclusion

The work here has shown that there is provable link between the growth rate of the tent map chaotic system and the entropy of binary data the system is able to produce. That link holds for both a theoretical system, simulated design, and physical implementation. Primarily, for the use of random number generation, we can create a system with a known symbol entropy to validate statistical test results. While other methods of true random bit generation from entropy measures rely on unproven or entropy sources of unknown quality [9][13], this chaotic oscillator is designed to produce maximally entropic bits. Modern statistical tests [25] are excellent at evaluating whether a sequence conforms to properties of ideal uniformity and indepence. However, these tests can fail to capture the importance of a high quality entropy source [20]. The next generation of true random number generators should be built on the principles of high entropy and evaluated not just on statistical properties, but also for its informational content.

The tent map circuit implementation requires very few components, consisting mostly of a negative RC filter and few digital logic components. The low power, low frequency breadboard design is already passing nearly all NIST statistical tests for random bit generation with rudimentary whitening algorithms. Now that the fundamentals of the design have been proven, improved implementations can be worked on. The easiest next step is moving the design to a small PCB. This implementation could improve overall stability by reducing noise sensitivity. Further design changes would include using a built-in low frequency square wave pulse generator, such as the 555-timer circuit, instead of an external function generator. The design would also be bolstered by the addition of an on device sampling circuit to extract symbols in real-time. With these design changes the chaotic oscillator could function fully independently and provide a continuous stream of random bits.

There are also more ways to test physical realizations of the tent map. Would using two synchronized oscillators produce significantly different results? How much improvement can be seen from using two completely oscillators for bit generation tied together with an XOR? Can improved designs with larger data collections pass all NIST tests and be tested against other rigorous statistical tests [18]? More testing should also be performed to verify the results here.

References

[1] Internet of Things (IoT) and non-IoT active device connections worldwide from 2010 to 2025, September 2022. Publication Title: Statista.

[2] Kathleen T. Alligood, Tim D. Sauer, and James A. Yorke. *Chaos: an introduction to dynamical systems*. Textbooks in mathematical sciences. Springer, New York, NY Heidelberg, repr. edition, 2010.

[3] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy dopant-level hardware Trojans: extended version. *Journal of Cryptographic Engineering*, 4(1):19–31, April 2014.

[4] Kamalika Bhattacharjee and Sukanta Das. A search for good pseudo-random number generators: Survey and empirical studies. *Computer Science Review*, 45:100471, August 2022.

[5] Abraham Boyarsky and Pawel Gora. *Laws of chaos: invariant measures and dynamical systems in one dimension*. Springer Science & Business Media, 2012.

[6] Ned J. Corron. Continuous-Time Limit of a Shift Map Yielding an Exactly Solvable Chaotic Differential Equation. *IFAC Proceedings Volumes*, 42(7):140–143, 2009.

[7] Ned J. Corron, Seth D. Cohen, Aubrey N. Beal, and Jonathan N. Blakely. Exact analytic solution for a chaotic hybrid dynamical system and its electronic realization. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 30(7):073112, July 2020.

[8] Ned J. Corron, Roy M. Cooper, and Jonathan N. Blakely. Analytically solvable chaotic oscillator based on a first-order filter. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 26(2):023104, February 2016.

[9] Ned J Corron, Roy M Cooper, and Jonathan N Blakely. Entropy rates of low-significance bits sampled from chaotic physical systems. *Physica D: Nonlinear Phenomena*, 332:34–40, 2016. Publisher: Elsevier.

[10] Thomas M Cover. *Elements of information theory*. John Wiley & Sons, 1999.

[11] George Cox, Charles Dike, and DJ Johnston. Intel's digital random number generator (DRNG). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–13. IEEE, 2011.

[12] Michael Crampin and Benedict Heal. On the Chaotic Behaviour of the Tent Map. *Teaching Mathematics and its Applications*, 13(2):83–89, 1994.

[13] Simson L. Garfinkel and Philip Leclerc. Randomness Concerns when Deploying Differential Privacy. In *Proceedings of the 19th Workshop on Privacy in the Electronic Society*, pages 73–86. ACM, November 2020. Place: Virtual Event USA.

[14] Pierre Gaspard and Xiao-Jing Wang. Noise, chaos, and (ε, τ)-entropy per unit time. *Physics reports*, 235(6):291–343, 1993. Publisher: Elsevier.

[15] Paweł Góra and Abraham Boyarsky. ON THE SIGNIFICANCE OF THE TENT MAP. *International Journal of Bifurcation and Chaos*, 13(5):1299–1301, May 2003.

[16] Michael Hamburg. Understanding Intel's Ivy Bridge Random Number Generator, December 2012. Publication Title: Electronic Design.

[17] Mike Hamburg, Paul Kocher, and Mark E Marson. Analysis of Intel's Ivy Bridge digital random number generator. *Online: http://www. cryptography. com/public/pdf/Intel_TRN G_Report_20120312. pdf*, 2012.

[18] R. Chase Harrison, Benjamin K. Rhea, Ariel N. Ramsey, Robert N. Dean, and J. Edmon Perkins. A True Random Number Generator based on a Chaotic Jerk System. In *2019 SoutheastCon*, pages 1–5, 2019.

[19] Remington Chase Harrison. *True Random Number Generation from a High Frequency Chaotic Jerk Circuit*. PhD Thesis, Auburn University, 2019.

[20] Joseph D Hart, Yuta Terashima, Atsushi Uchida, Gerald B Baumgartner, Thomas E Murphy, and Rajarshi Roy. Recommendations and illustrations for the evaluation of photonic random number generators. *arXiv preprint arXiv:1612.04415*, 2016.

[21] Frederick James and Lorenzo Moneta. Review of High-Quality Random Number Generators. *Computing and Software for Big Science*, 4(1):2, December 2020.

[22] Prasanna Raghaw Mishra, Bhartendu Nandan, and Navneet Gaba. An Efficient and Compact Reformulation of NIST Collision Estimate Test. *Cryptology ePrint Archive*, 2019.

[23] Nithin Nagaraj. The Unreasonable Effectiveness of the Chaotic Tent Map in Engineering Applications. *Chaos Theory and Applications*, 4(4):197–204, December 2022.

[24] Dan Petro and Allan Cecil. You're Doing IoT Security RNG: The Crack in the Foundation of IoT, August 2021.

[25] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, N. Heckert, James Dray, San Vo, and Lawrence Bassham. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Number NIST Special Publication (SP) 800-22 Rev. 1. National Institute of Standards and Technology, April 2010.

[26] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[27] Thomas Shrimpton and R Seth Terashima. A provable-security analysis of Intel's secure key RNG. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 77–100. Springer, 2015.

[28] Sedra Smith. Microelectronic Circuits 7th ed., 2014.

[29] Micah Tseng. Extended, exactly solvable chaotic oscillator. 2023.

[30] Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry A McKay, Mary L Baish, and Mike Boyle. *Recommendation for the entropy sources used for random bit generation*. Number NIST SP 800-90b. National Institute of Standards and Technology, Gaithersburg, MD, January 2018.

[31] John Von Neumann and others. Various techniques used in connection with random digits.

Derivation of Fixed Points

A.1 Logistic Map

Fixed points of a dynamical system with a discrete time representation can easily be found by substituting the variable for the next iterate with the current iterate. Then, by solving for the sole variable, the value or values for which the system reaches a steady state will be known:

$$\begin{aligned} x_{n+1} &= r x_n (1 - x_n) \\ x_f &= r x_f (1 - x_f). \end{aligned} \tag{A.1}$$

Rearranging gives the form of a quadratic equation, where $a = -r$, $b = r - 1$, and $c = 0$. Equation A.2 implements the quadratic formula and solves for both possible solutions:

$$0 = -rx_f^2 + (r-1)x_f$$

$$x_f = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$x_f = \frac{-(r-1) \pm \sqrt{((r-1)^2 - 0}}{-2r} \tag{A.2}$$

$$x_f = \frac{-r + 1 \pm (r-1)}{-2r}.$$

Two solutions arise from equation A.2. The first case is taken when addition is used in the numerator. The first fixed point is located at 0. Equation A.3 simplifies the case where addition is chosen in the numerator:

$$x_{f1} = \frac{-r + 1 + r - 1}{-2r}$$

$$x_{f1} = \frac{0}{-2r} \tag{A.3}$$

$$x_{f1} = 0.$$

The second solution is the case when subtraction is used in the numerator. The fixed point is $1 - \frac{1}{r}$, where r is the growth rate of the system. Equation A.4 simplifies the case where subtraction is chosen in the numerator:

$$x_{f2} = \frac{-r + 1 - r + 1}{-2r}$$

$$x_{f2} = \frac{-2r + 2}{-2r} \tag{A.4}$$

$$x_{f2} = 1 - \frac{1}{r}.$$

A.2 Tent Map

The fixed points of the tent map are found using the same method as with the logistic map. First, the discrete time equation is altered so the output state matches the input state. The important difference for the Tent Map is that it is a piecewise equation, so both equations need to be tested for fixed points. The fixed point is located at 0. Equation A.5 solves the first case where values are below the threshold:

$$x_{n+1} = \mu x_n$$

$$x_f = \mu x_f \tag{A.5}$$

$$x_f = 0,$$

The second case for values above the threshold requires a little more effort. Equation A.6 shows the second fixed point is located at the value $\frac{\mu}{(1-\mu)}$:

$$
\begin{aligned}
x_{n+1} &= \mu(1 - x_n) \\
x_f &= \mu(1 - x_f) \\
x_f &= \mu - \mu x_f \\
(1 - \mu)x_f &= \mu \\
x_f &= \frac{\mu}{(1 - \mu)}.
\end{aligned} \tag{A.6}
$$

Stability of the two fixed points are a trivial calculation for the tent map as all equations are linear. The first fixed point, 0 lies below the threshold value, so equation A.5 will be used to evaluate the stability. Equation A.7 takes the magnitude of the derivative of the function at the fixed point and solves for the stability:

$$
\begin{aligned}
|\frac{d}{dx}f(x_f)| &= |\frac{d}{dx}[\mu x_f]| \\
|\frac{d}{dx}f(x_n)| &= \mu.
\end{aligned} \tag{A.7}
$$

For all values of μ, where $\mu > 1$, the fixed point located at 0 will be an unstable source. The second fixed point, $x_f = \frac{\mu}{(1-\mu)}$ is above the threshold $\frac{1}{2}$ for all values of μ, where $\mu > 1$. As such, the equation from A.6 will be used to evaluate the fixed point's stability. Equation A.8 performs the same computation as before, this time using the equation for values above the threshold:

$$
\begin{aligned}
&|\frac{d}{dx}f(x_f)| = |\frac{d}{dx}[\mu - \mu x_f]| \\
&\frac{d}{dx}f(x_f) = |-\mu| \\
&|\frac{d}{dx}f(x_f)| = \mu.
\end{aligned}
\tag{A.8}
$$

This fixed point is also an unstable source for all values of μ, where $\mu > 1$. Both fixed points are unstable, preventing the system from reaching a steady state.

Derivation of Tent Map Basis Function

The analytic solution for the basis function of the tent map has been previously solved [8] [7]. It establishes a set of continuous time functions with a discrete switching condition to guard against the system becoming unstable. The initial differential equation is in B.1 features a continuous state $u(t)$ and a discrete state $s(t)$ [8]:

$$\begin{aligned} \frac{du}{dt} &= u - s \\ s(t) &\in -1, +1 \\ u(t) &\in R. \end{aligned} \tag{B.1}$$

The discrete state is defined by the sign of the continuous state. When $u(t) > 0$, $s(t) = 1$ and when $u(t) \leq 0$, $s(t) = -1$. The switching condition bounds the system so long as the switching period is smaller than the time the system needs to grow past the bounds [8]. Equation B.2 defines the boundingg conditions:

$$s(nT) = \begin{cases} +1, & u(nT) > 0 \\ -1, & u(nT) \leq 0 \end{cases}. \tag{B.2}$$

The equation in B.3 is chosen to satisfy the conditions of B.1 and B.2 as the forcing function of the differential equation. The summation will switch between the positive and negative guard conditions as the solution u(t) grows in either direction and the pulse timing remains within T [8]. $\phi(t)$ is defined for time t, where $0 \leq t < T$ in equation B.4:

$$s(t) = \sum_{n=-\infty}^{\infty} s_n \phi(t - nT). \tag{B.3}$$

$$\phi(t) = \begin{cases} 1, & 0 \leq t < T \\ 0, & otherwise \end{cases} \tag{B.4}$$

We assume equation B.1 is a first order linear ODE with forcing function s(t) defined in equation B.3. e^{-t} is the chosen integrating factor to solve the differential equation [8]. This results in equation B.5:

$$u(t) = \int_{t}^{\infty} s(\tau) e^{t-\tau} \, d\tau. \tag{B.5}$$

$s(\tau)$ in equation B.5 can be substituted with the full equation from B.3 resulting in equation B.6 [8]:

$$u(t) = \sum_{n=-\infty}^{\infty} s_n \int_{t}^{\infty} \phi(t - nT) e^{t-\tau} \, d\tau. \tag{B.6}$$

A simple change of variable will simplify the integration [8]. $\tau - nT$ is substituted for θ in equation B.7:

$$u(t) = \int_{t}^{\infty} \phi(t - nT)e^{t-\tau}\, d\tau = e^{t-nT} \int_{t-nT}^{\infty} \phi(\theta)e^{-\theta}\, d\theta. \tag{B.7}$$

The integration only depends on the interval of $t - nT$ for all time [8]. Equation B.8 shows how the integration of $u(t)$ will be defined as the function $P(t)$:

$$u(t) = \sum_{n=-\infty}^{\infty} s_n P(t - nT). \tag{B.8}$$

$P(t)$ is integrated over all time to derive the basis pulse of the system, B.9 [8]:

$$P(t) = e^{t} \int_{t}^{\infty} \phi(\tau)e^{\theta}\, d\theta. \tag{B.9}$$

From the integration, P(t) is defined for three regions as seen in B.10. For $t < 0$, $P(t)$ is exponentially increasing from to and bounded to $1 - e^{-T}$ at $t = 0$. For $0 \leq t < T$, $P(t)$ exponentially decreases from $1 - e^{-T}$ to 0 over the period of T. For all other time, The function is 0 [8]. Equation B.10 defines the basis function $P(t)$ for all three cases:

$$P(t) = \begin{cases} (1 - e^{-T})e^{t}, & t < 0 \\ 1 - e^{t-T}, & 0 \leq t < T \\ 0, & T \leq t \end{cases}. \tag{B.10}$$

Shannon Entropy to Lyapunov Exponent

Claude Shannon developed a versatile method of measuring entropy [26] for systems with any number of states where each state has an observed probability probability of occurrence, p_i. Equation C.1 shows Shannon's equation for entropy:

$$H = -K \sum_{i}^{n} p_i \times \log_2 p_i. \tag{C.1}$$

For an ideal uniform random binary variable, the probability of both states will be exactly $\frac{1}{2}$. The units of the entropy measurement can be changed by altering the base of the logarithm used with constant K. Log base 2 will give entropy in units of *bits* and the natural logarithm will give entropy in units of *nats*. The rule of logarithms allows the change of base by dividing the current logarithm by a logarithm with the same base of the desired base in equation C.2:

$$\begin{aligned} p_0 = p_1 &= \frac{1}{2} \\ K &= \frac{1}{\log_2(e)}. \end{aligned} \tag{C.2}$$

Expanding the equation for each iteration of the summation gives equation C.3:

$$H = -\frac{1}{\log_2(e)} \times (p_0 \times \log_2(p_0) + p_1 \times \log_2(p_1)). \tag{C.3}$$

Calculating the result with the known variables and constants gives equation C.4:

$$H = -\frac{1}{\log_2(e)} \times (\frac{1}{2} \times \log_2(\frac{1}{2}) + \frac{1}{2} \times \log_2(\frac{1}{2})). \tag{C.4}$$

Computing the final result in equation C.5 shows that the entropy in nats is the natural logarithm of 2:

$$H = -\frac{\log_2(\frac{1}{2})}{log_2(e)} = 0.6931. \tag{C.5}$$

The Lyapunov exponent of a chaotic systems measures the rate of divergence from two close points over time [2]. For a continuous time map, this is defined in equation C.6 as the limit of the natural logarithm of the state at time t divided by an initial state at time t_0 all divided by the time since t_0 as time goes to infinity. This equation is different for a discrete time system. The new equation, shown in C.7, is the limit of the average of the natural logarithm of the absolute value of the derivative of the system for n number of states, as n goes to infinity. Equations C.6 and C.7 show the two methods for calculating the Lyapunov exponent:

$$\lambda = \lim_{t\to\infty} \frac{1}{t} \ln \frac{|x(t)|}{|x(t_0)|}. \tag{C.6}$$

$$\lambda = \lim_{n \to \infty} \frac{1}{n} \sum_{1}^{n} \ln \| \frac{d}{dx} f(x) | \tag{C.7}$$

For the tent map system this is a trivial calculation. It is a discrete time systems, so equation C.7 will be used. The derivative of the system is 2 for all possible states, so the equation will collapse to the natural logarithm of 2, shown in equation C.8:

$$\lambda = \ln | \frac{d}{dx} \begin{cases} 2x_n, & x_n \leq \frac{1}{2} \\ 2(1 - x_n), & x_n > \frac{1}{2} \end{cases} | = \ln |2| = 0.6931. \tag{C.8}$$

There is an established link between the growth rate of the Tent Map and its entropy when it is operating under ideal conditions as a full height map.

Negative Impedance Converter Solution

The negative impedance converter is the backbone of the negative RC circuit which creates the chaotic waveform. It is important to know how this circuit causes current flow in the opposite direction and allows the voltage across the capacitor to mimic a reverse time charge and discharge. The system is made with three resistors. R_1 and R_2 are feedback from the output of the op-amp to the positive and negative terminals respectively. The third resistor, R_3, connects to the inverting terminal and ground. Ideal op-amp assumptions are used for infinite input impedance and zero input offset [28]. $R_{in} = \infty$ and $V_+ = v_-$.

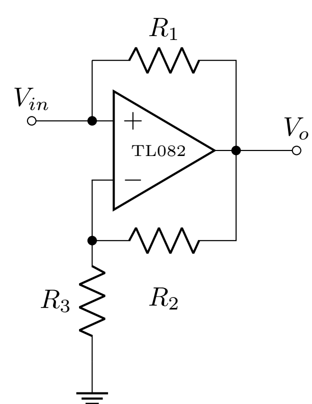

Figure D.1: Negative Impedance Converter Circuit Diagram - Voltage input at the non-inverting terminal. Resistors R_1 and R_2 in the feedback are equivalent. R_3 defines the positive resistance of the NIC. Voltage output at the output of the Op-amp.

We will also define currents I_1, I_2, and I_3 as the currents across resistors R_1, R_2, and R_3. The voltage inputs to the op-amp terminals are noted as V_+ and V_-. The equations in D.1 are the three currents slowing through their respective resistors:

$$
\begin{aligned}
I_1 &= \frac{V_+ - V_o}{R_1} \\
I_2 &= \frac{V_- - V_o}{R_2} \\
I_3 &= \frac{V_- - 0}{R_3}.
\end{aligned}
\tag{D.1}
$$

The first step is to observe from Kirchhoff's Current Law there is a single path for current to flow from the input voltage to ground [28]. The result needed is the voltage at the non-inverting terminal, V_+ in terms of the current through the circuit and the three resistances. Solving for V_o in terms of I_1 allows us to make a substitution in the equation in D.1 for I_2. Equation D.2 defines the output voltage interms of the input voltage:

$$
V_o = V_{in} - I_1 R_1. \tag{D.2}
$$

Equation D.2 is substituted into equation D.1 to redefine I_2 only in terms of current and resistances:

$$
\begin{aligned}
I_2 &= \frac{V_{in} - I_1 R_1 - V_{in}}{R_2} \\
I_2 &= \frac{-I_1 R_1}{R_2}.
\end{aligned}
\tag{D.3}
$$

Another assumption can be made through the observation of the circuit diagram. Currents I_2 and I_3 have only one path to ground since there is assumed to be no current flow to the inverting terminal of the op-amp. As such, it is safe to assume that these currents are equal. Using the other assumption for zero input offset, V_- in equation D.1 for I_3 can be replaced with V_+. Then, rearrange for V_+ in terms of I_2:

$$V_+ = I_2 R_3. \tag{D.4}$$

Substitute the result from equation D.3 into equation D.4 to obtain the non-inverting terminal voltage in terms of I_1:

$$V_+ = \frac{-R_1}{R_2} I_1 R_3. \tag{D.5}$$

For the case where feedback resistors R_1 and R_2 are equal, the equation simplifies to only depend on the current I_3, which is equal to I_1, and resistor R_3. The voltage at the non-inverting terminal is simply the negative of the current times the resistance of R_3:

$$V_+ = -I_3 R_3. \tag{D.6}$$

MATLAB Code

E.1 Data Sampling and Testing

E.1.1 Data Extraction

```
%% Extract Data
% This code will extract the voltage samples and binary values from the
% data taken from the circuit and store them in binary files with int-32
% precision.

% Use a loop to continue prompting for data files until all data has been
% extracted

%% Set up environment
clear
addpath("funnctions\")


%% First data sample

% Have the user select the text file with the circuit output data
[baseName, folder] = uigetfile("*.bin*",'Select a File','C:\Users\tinma\Documents\Master_Thesis\Data\');
bin_file_name = fullfile(folder, baseName);
disp(baseName)
disp("")

```

```
clear baseName folder
TIME = input('Time Length: '); %10*60;
RATE = input('Sample Rate: '); %5e5;
FREQ = input('Oscillator Frequency: '); %~363Hz
threshold = 2.5;
fileID = fopen(bin_file_name);
data = fread(fileID,[2,TIME*RATE],'float32'); %'float32'
fclose(fileID);

% Imported Data
vc = data(1,:);
clk = data(2,:);
clear data

% Sample data from circuit
disp('Sampling Values')
x = [];      % array to hold sampled points

if RATE <= 250000
    N = 1;
elseif RATE <= 312500
    N = 2;
elseif RATE <= 400000
    N = 3;
else
    N = 4;
end

crossings = ((circshift(clk < 2.5,1) - (clk<2.5)) ~= 0);
clear clk

samples = circshift(crossings,-N).*vc;
x = samples(samples~=0);
clear crossings samples vc

% Partition samples and convert to binary
```

```
partition = FindPartition(x);
fprintf('Partition: %f\n\n', partition)
bits = (x>=partition);


% Create files to write voltages and bits to
% date_TIME(s)_FREQ(Hz)
file_name_v = strcat(string(datetime('today','Format','uuuu-MM-dd')),"_",string(round(FREQ)),"Hz","_voltages.bin");
fileID = fopen(file_name_v,'w');
fwrite(fileID,x,'double');
fclose(fileID);

file_name_b = strcat(string(datetime('today','Format','uuuu-MM-dd')),"_",string(round(FREQ)),"Hz","_bits.bin");
fileID = fopen(file_name_b,'w');
fwrite(fileID,bits,'logical');
fclose(fileID);
%% Remaining Data samples

[baseName, folder] = uigetfile("*.bin*",'Select a File','C:\Users\tinma\Documents\Master_Thesis\Data\');
bin_file_name = fullfile(folder, baseName);
disp(baseName)
disp("")

% Loop through all needed data
while (baseName ~= 0)
    clear baseName folder

    TIME = input('Time Length: '); %10*60;
    RATE = input('Sample Rate: '); %5e5;
    FREQ = input('Oscillator Frequency: '); %~363Hz
    threshold = 2.5;
    fileID = fopen(bin_file_name);
```

```
    data = fread(fileID,[2,TIME*RATE],'float32'); %'
       float32','float64','int16'
    fclose(fileID);

    % Imported Data
    vc = data(1,:);
    clk = data(2,:);
    clear data

    % Sample data from circuit
    disp('Sampling Values')
    x = [];     % array to hold sampled points

    if RATE <= 250000
        N = 1;
    elseif RATE <= 312500
        N = 2;
    elseif RATE <= 400000
        N = 3;
    end

    crossings = ((circshift(clk < 2.5,1) - (clk<2.5))
       ~= 0);
    clear clk

    samples = circshift(crossings,-N).*vc;
    x = samples(samples~=0);
    clear crossings samples vc

    % Partition samples and convert to binary
    partition = FindPartition(x);
    fprintf('Partition: %f\n\n', partition)
    bits = (x>=partition);

    % Write to previously created files
    file_name_v = strcat(string(datetime('today','
       Format','uuuu-MM-dd')),"_",string(round(FREQ)),"
       Hz","_voltages.bin");
```

```
    fileID = fopen(file_name_v,'a');
    fwrite(fileID,x,'double');
    fclose(fileID);

    file_name_b = strcat(string(datetime('today','Format','uuuu-MM-dd')),"_",string(round(FREQ)),"Hz","_bits.bin");
    fileID = fopen(file_name_b,'a');
    fwrite(fileID,bits,'logical');
    fclose(fileID);


    % Attempt to open next file
    [baseName, folder] = uigetfile("*.bin*",'Select a File','C:\Users\tinma\Documents\Master_Thesis\Data\');
    bin_file_name = fullfile(folder, baseName);
    disp(baseName)
    disp("")
end

% clear all
```

This code takes in time series data taken from an oscilloscope for two waveforms, the chaotic wave and the clock. Multiple data captures can be input and appended together. A user inputs the length of the data in seconds and the frequency at which it was sampled. Then, the code samples the chaotic wave at each rising and falling edge and writes the voltage values out to a binary file. It also calls a function to calculate the partition of the tent map in order to generate the symbol sequence. The binary symbols are also written to a file.

E.1.2 Binary to Text Conversion

```
%% Binary to Text
% This code will take a bitstream binary file and convert it into a
% text file with 50 columns

%% Set up environment
clear
addpath("funnctions\")


%% IMPORT DATA
[baseName, folder] = uigetfile("*.bin*");
bin_file_name = fullfile(folder, baseName);
fileID = fopen(bin_file_name);
bits = fread(fileID,'logical')'>0;
fclose(fileID);

%% Reshape Array

c_size = 100;
r_size = floor(length(bits)/c_size);

%% Write to File

file_name = strcat(string(c_size*r_size),"_bits.txt");
fileID = fopen(file_name,'w');

for i=1:r_size
    % turn row into string
    line = strcat(strrep(num2str(bits(c_size*(i-1)+1:c_size*i)),' ',''),'\n');
    % frwite string + newline (\n)
    fprintf(fileID,line);
end

fclose(fileID);
```

This code converts a binary data file into an ASCII test file format.

E.1.3 Analog Testing

```
%% Analog Testing

clear
close all
addpath("funnctions\")
addpath("NIST Entropy Tests\")
addpath("NIST Stat Function\")
addpath("Entropy Estimation\")

%% Import analog voltage samples
[baseName, folder] = uigetfile("*.bin*");
bin_file_name = fullfile(folder, baseName);
fileID = fopen(bin_file_name);
x = fread(fileID,'double')';
fclose(fileID);

%% TESTS

L = length(x);

% Volt Hist
VoltHist(x)

% Partition Tent
partition = FindPartition(x(1:800000));
fprintf('Partition: %f\n\n', partition)

% Slopes
[pL,pR,pA] = Calculate_Slopes(x(1:800000),partition);
fprintf(" Left Branch slope: %f\n Right Branch Slope: %
    f\n Overall Slope: %f\n\n",pL,pR,pA)

ENTROPY_TEST_NAME = [];
ENTROPY_TEST_STAT = [];

%% NIST Entropy tests
```

```
T = Excursion(x);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "Max Deviation from Avg"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" Maximum Deviation from average: %f\n\n", T)

T = DirectionalRunNum(x);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "# of Directional Runs"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" Number of Directional Runs: %d\n\n",T)

T = DirectionalRunLen(x);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "Longest Run"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" Longest Run Length: %d\n\n",T)

T = NumUpDown(x);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "Max Increase|Decrease"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" Maximum number of increases or decreases: %d\n\n",T)

T = MedianRunNum(x);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "# of Runs based on Median"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" Number of Runs based on Median: %d\n\n",T)

T = MedianRunLen(x);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "Max Run Based on Median"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" Maximum run length based on Median: %d\n\n",T)

[aT, mT] = Collisions(x);
```

```
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "Avg Length
   before Collision"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT aT];
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "Longest
   Distance between Collision"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT mT];
fprintf(" Average length before a collision: %f\n\n",aT
   )
fprintf(" Longest distance between collisions: %d\n\n",
   mT)


T = Periodicity(x);
y = [1 2 8 16 32];
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME strcat(string(y)
   ,repmat(" Period Structures",1,length(y)))];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" 01 period structures: %d\n 02 period
   structures: %d\n 08 period structures: %d\n" + ...
    " 16 period structures: %d\n 32 period structures:
       %d\n\n",T(1),T(2),T(3),T(4),T(5))

T = Covariance(x);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME strcat(string(y)
   ,repmat(" Period Covariance",1,length(y)))];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" 01 period covariance: %d\n 02 period
   covariance: %d\n 08 period covariance: %d\n" + ...
    " 16 period covariance: %d\n 32 period covariance:
       %d\n\n",T(1),T(2),T(3),T(4),T(5))


ENTROPY_TESTS = table(ENTROPY_TEST_NAME',
   ENTROPY_TEST_STAT');
ENTROPY_TESTS.Properties.VariableNames = ["Entropy Test
    Name","Test Value"];
```

```
xl_filename = strcat(string(datetime('today','Format','uuuu-MM-dd')),"_TESTING_RESULTS_",string(L),"_Samples.xlsx");
writetable(ENTROPY_TESTS,xl_filename,'Sheet',"Voltage Samples",'Range','F1','WriteVariableNames',true)
%% Entropy Estimations
ENTROPY_EST_NAME = [];
ENTROPY_EST_VAL = [];
% Most Common Value Estimate
T = MostCommonValue(x(1:2e6));
ENTROPY_EST_NAME = [ENTROPY_EST_NAME "Most Common Value Estimate"];
ENTROPY_EST_VAL = [ENTROPY_EST_VAL T];
fprintf(" Most Common Value Estiamte: %f\n\n",T)
% Tuple Estimate
T = TupleEstimateVoltage(x(1:2e6));
ENTROPY_EST_NAME = [ENTROPY_EST_NAME "Tuple Estimate"];
ENTROPY_EST_VAL = [ENTROPY_EST_VAL T];
fprintf(" Tuple Estimate: %f\n\n",T)
% Most Common Window Estimate
T = MostCommonWindowEstimate(x(1:2e6));
ENTROPY_EST_NAME = [ENTROPY_EST_NAME "Most Common Window Estimate"];
ENTROPY_EST_VAL = [ENTROPY_EST_VAL T];
fprintf(" Tuple Estimate: %f\n\n",T)
ENTROPY_ESTS = table(ENTROPY_EST_NAME',ENTROPY_EST_VAL');
ENTROPY_ESTS.Properties.VariableNames = ["Entropy Estimate Name","Estimate Value"];
```

```
writetable(ENTROPY_ESTS,xl_filename,'Sheet',sheets(
    corrected+1),'Range','I1','WriteVariableNames',true)
```

This code reads the binary data file of voltage samples and calls many functions to run tests on the data. This includes generating a tent map and calculating the slopes. Results are written to a CSV file.

E.1.4 Digital Testing

```
%% Digital Testing

clc
clear
close all
addpath("funnctions\")
addpath("NIST Entropy Tests\")
addpath("NIST Stat Function\")
addpath("Entropy Estimation\")


%% IMPORT DATA
[baseName, folder] = uigetfile("*.bin*");
bin_file_name = fullfile(folder, baseName);
fileID = fopen(bin_file_name);
bits = fread(fileID,'logical')'>0;
fclose(fileID);

% bits = bits(1:3699200);          % FOR TESTING ONLY
%% TESTS


disp("ORIGINAL BIT TESTS")
disp(repmat('~',1,45))

L = length(bits);
corrected = 0;

while 1

% Auto-Correlation
AutoCorr(bits);

% Binning
bn = MakeBins(bits);

```

```
% Distribution of converted binary values
% Should be close to uniform ditribution
figure('DefaultAxesFontSize',10,'DefaultAxesFontName','
   Times New Roman', 'Color', 'White');
histogram(bn,100);
if corrected > 0
    t = "Histogram of Corrected Bins";
else
    t = "Histogram of Original Bins";
end
title(t)
xlabel('Measured values')
ylabel('Count')

clear bn


STAT_TEST_NAME = [];
STAT_TEST_RESULT = [];
STAT_TEST_PVAL = [];

ENTROPY_TEST_NAME = [];
ENTROPY_TEST_STAT = [];

ENTROPY_EST_NAME = [];
ENTROPY_EST_VAL = [];


%% NIST STAT TESTS
[P,pass] = Monobit(bits);
STAT_TEST_NAME = [STAT_TEST_NAME "Monobit"];
STAT_TEST_RESULT = [STAT_TEST_RESULT pass];
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
if pass > 0
    fprintf("Monobit: PASS\nP-Value: %f\n\n",P)
else
    fprintf("Monobit: FAIL\nP-Value: %f\n\n",P)
end
```

```

[P,pass] = Block(bits);
STAT_TEST_NAME = [STAT_TEST_NAME "Block Frequency"];
if pass>0 STAT_TEST_RESULT = [STAT_TEST_RESULT "PASS"];
    else STAT_TEST_RESULT = [STAT_TEST_RESULT "FAIL"];
   end
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
if pass > 0
    fprintf("Block Frequency: PASS\nP-Value: %f\n\n",P)
else
    fprintf("Block Frequency: FAIL\nP-Value: %f\n\n",P)
end


[P,pass] = Runs(bits);
STAT_TEST_NAME = [STAT_TEST_NAME "Runs Test"];
if pass>0 STAT_TEST_RESULT = [STAT_TEST_RESULT "PASS"];
    else STAT_TEST_RESULT = [STAT_TEST_RESULT "FAIL"];
   end
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
if pass > 0
    fprintf("Runs Test: PASS\nP-Value: %f\n\n",P)
else
    fprintf("Runs Test: FAIL\nP-Value: %f\n\n",P)
end


[P,pass] = LongestRun(bits);
STAT_TEST_NAME = [STAT_TEST_NAME "Longest Run"];
if pass>0 STAT_TEST_RESULT = [STAT_TEST_RESULT "PASS"];
    else STAT_TEST_RESULT = [STAT_TEST_RESULT "FAIL"];
   end
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
if pass > 0
    fprintf("Longest Run: PASS\nP-Value: %f\n\n",P)
else
    fprintf("Longest Run: FAIL\nP-Value: %f\n\n",P)
end
```

```

[P,pass] = MatrixRank(bits);
STAT_TEST_NAME = [STAT_TEST_NAME "Matrix Rank"];
if pass>0 STAT_TEST_RESULT = [STAT_TEST_RESULT "PASS"]; else STAT_TEST_RESULT = [STAT_TEST_RESULT "FAIL"]; end
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
if pass > 0
    fprintf("Matrix Rank: PASS\nP-Value: %f\n\n",P)
else
    fprintf("Matrix Rank: FAIL\nP-Value: %f\n\n",P)
end

[P,pass] = DFT(bits);
STAT_TEST_NAME = [STAT_TEST_NAME "Discrete Fourier Transform"];
if pass>0 STAT_TEST_RESULT = [STAT_TEST_RESULT "PASS"]; else STAT_TEST_RESULT = [STAT_TEST_RESULT "FAIL"]; end
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
if pass > 0
    fprintf("Discrete Fourier Transform: PASS\nP-Value: %f\n\n",P)
else
    fprintf("Discrete Fourier Transform: FAIL\nP-Value: %f\n\n",P)
end


[P,pass] = NonOverlapTemplate(bits);
STAT_TEST_NAME = [STAT_TEST_NAME "Non-Overlapping Template Matching"];
if pass>0 STAT_TEST_RESULT = [STAT_TEST_RESULT "PASS"]; else STAT_TEST_RESULT = [STAT_TEST_RESULT "FAIL"]; end
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
if pass > 0
```

```
    fprintf("Non-Overlapping Template Matching: PASS\nP-Value: %f\n\n",P)
else
    fprintf("Non-Overlapping Template Matching: FAIL\nP-Value: %f\n\n",P)
end


[P,pass] = OverlapTemplate(bits);
STAT_TEST_NAME = [STAT_TEST_NAME "Overlapping Template Matching"];
if pass>0 STAT_TEST_RESULT = [STAT_TEST_RESULT "PASS"]; else STAT_TEST_RESULT = [STAT_TEST_RESULT "FAIL"]; end
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
if pass > 0
    fprintf("Overlapping Template Matching: PASS\nP-Value: %f\n\n",P)
else
    fprintf("Overlapping Template Matching: FAIL\nP-Value: %f\n\n",P)
end


[P,pass] = Maurer(bits);
STAT_TEST_NAME = [STAT_TEST_NAME "Maurer's Test"];
if pass>0 STAT_TEST_RESULT = [STAT_TEST_RESULT "PASS"]; else STAT_TEST_RESULT = [STAT_TEST_RESULT "FAIL"]; end
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
if pass > 0
    fprintf("Maurer's Test: PASS\nP-Value: %f\n\n",P)
else
    fprintf("Maurer's Test: FAIL\nP-Value: %f\n\n",P)
end


[P,pass] = LinearComplexity(bits);
```

```
if pass > 0
    fprintf("Linear Complexity: PASS\nP-Value: %f\n\n",P)
else
    fprintf("Linear Complexity: FAIL\nP-Value: %f\n\n",P)
end


[P1,P2,pass] = Serial(bits);
STAT_TEST_NAME = [STAT_TEST_NAME "Serial Test"];
if pass>0 STAT_TEST_RESULT = [STAT_TEST_RESULT "PASS"]; else STAT_TEST_RESULT = [STAT_TEST_RESULT "FAIL"]; end
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
if pass > 0
    fprintf("Serial Test: PASS\nP-Values: %f, %f\n\n",P1,P2)
else
    fprintf("Serial Test: FAIL\nP-Value: %f, %f\n\n",P1,P2)
end


[P,pass] = ApproxEntropy(bits);
STAT_TEST_NAME = [STAT_TEST_NAME "Approximte Entropy Test"];
if pass>0 STAT_TEST_RESULT = [STAT_TEST_RESULT "PASS"]; else STAT_TEST_RESULT = [STAT_TEST_RESULT "FAIL"]; end
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
if pass > 0
    fprintf("Approximate Entropy Test: PASS\nP-Value: %f\n\n",P)
else
    fprintf("Approximate Entropy Test: FAIL\nP-Value: %f\n\n",P)
end
```

```
[P,pass] = CumSum(bits,0);
STAT_TEST_NAME = [STAT_TEST_NAME "Cumulative Sums Forward"];
if pass>0 STAT_TEST_RESULT = [STAT_TEST_RESULT "PASS"]; else STAT_TEST_RESULT = [STAT_TEST_RESULT "FAIL"]; end
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
if pass > 0
    fprintf("Cumulative Sums Forward: PASS\nP-Value: %f\n\n",P)
else
    fprintf("Cumulative Sums Forward: FAIL\nP-Value: %f\n\n",P)
end


[P,pass] = CumSum(bits,1);
STAT_TEST_NAME = [STAT_TEST_NAME "Cumulative Sums Backwards"];
if pass>0 STAT_TEST_RESULT = [STAT_TEST_RESULT "PASS"]; else STAT_TEST_RESULT = [STAT_TEST_RESULT "FAIL"]; end
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
if pass > 0
    fprintf("Cumulative Sums Backward: PASS\nP-Value: %f\n\n",P)
else
    fprintf("Cumulative Sums Backward: FAIL\nP-Value: %f\n\n",P)
end


[P,pass] = RandomExcursion(bits);
x = [-4 -3 -2 -1 1 2 3 4];
STAT_TEST_NAME = [STAT_TEST_NAME strcat(repmat("Random Excursions ",1,length(x)),string(x))];
```

```
RESULT = string(zeros(1,length(pass)));
for i=1:length(pass), if pass(i)>0, RESULT(i)="PASS"; else, RESULT(i)="FAIL"; end,end
STAT_TEST_RESULT = [STAT_TEST_RESULT RESULT];
STAT_TEST_PVAL = [STAT_TEST_PVAL P'];
for i = 1:length(P)
    if pass(i) > 0
        fprintf("Random Excursion (%d): PASS\nP-Value: %f\n\n",x(i),P(i))
    else
        fprintf("Random Excursion (%d): FAIL\nP-Value: %f\n\n",x(i),P(i))
    end
end

[P,pass] = RandomExcursionVariant(bits);
x = [-9 -8 -7 -6 -5 -4 -3 -2 -1 1 2 3 4 5 6 7 8 9];
STAT_TEST_NAME = [STAT_TEST_NAME strcat(repmat("Random Excursions Variant ",1,length(x)),string(x))];
RESULT = string(zeros(1,length(pass)));
for i=1:length(pass), if pass(i)>0, RESULT(i)="PASS"; else, RESULT(i)="FAIL"; end,end
STAT_TEST_RESULT = [STAT_TEST_RESULT RESULT];
STAT_TEST_PVAL = [STAT_TEST_PVAL P];
for i = 1:length(P)
    if pass(i) > 0
        fprintf("Random Excursion Variant (%d): PASS\nP-Value: %f\n\n",x(i),P(i))
    else
        fprintf("Random Excursion Variant (%d): FAIL\nP-Value: %f\n\n",x(i),P(i))
    end
end

STAT_TESTS = table(STAT_TEST_NAME',STAT_TEST_RESULT',STAT_TEST_PVAL');
STAT_TESTS.Properties.VariableNames = ["Stat Test Name","Test Result","P-Value"];
```

```
xl_filename = strcat(string(datetime('today','Format','uuuu-MM-dd')),"_DIGITAL_TESTING_RESULTS_",string(L),"_BITS.xlsx");
sheets = ["Original Bits" "Corrected Bits"];
writetable(STAT_TESTS,xl_filename,'Sheet',sheets(corrected+1),'Range','B1','WriteVariableNames',true)


%% NIST ENTROPY TESTS

T = Excursion(bits);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "Max Deviation from Avg"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" Maximum Deviation from average: %d\n\n", T)

T = DirectionalRunNum(bits);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "# of Directional Runs"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" Number of Directional Runs: %d\n\n",T)

T = DirectionalRunLen(bits);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "Longest Run"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" Longest Run Length: %d\n\n",T)

T = NumUpDown(bits);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "Max Increase|Decrease"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" Maximum number of increases or decreases: %d\n\n",T)

T = MedianRunNum(bits);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "# of Runs based on Median"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
```

```
fprintf(" Number of Runs based on Median: %d\n\n",T)

T = MedianRunLen(bits);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "Max Run Based on Median"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" Maximum run length based on Median: %d\n\n",T)

[aT, mT] = Collisions(bits);
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "Avg Length before Collision"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT aT];
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME "Longest Distance between Collision"];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT mT];
fprintf(" Average length before a collision: %f\n\n",aT)
fprintf(" Longest distance between collisions: %d\n\n",mT)

T = Periodicity(bits);
x = [1 2 8 16 32];
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME strcat(string(x),repmat(" Period Structures",1,length(x)))];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" 01 period structures: %d\n 02 period structures: %d\n 08 period structures: %d\n" + ...
    " 16 period structures: %d\n 32 period structures: %d\n\n",T(1),T(2),T(3),T(4),T(5))

T = Covariance(bits);
x = [1 2 8 16 32];
ENTROPY_TEST_NAME = [ENTROPY_TEST_NAME strcat(string(x),repmat(" Period Covariance",1,length(x)))];
ENTROPY_TEST_STAT = [ENTROPY_TEST_STAT T];
fprintf(" 01 period covariance: %d\n 02 period covariance: %d\n 08 period covariance: %d\n" + ...
```

```
    " 16 period covariance: %d\n 32 period covariance:
       %d\n\n",T(1),T(2),T(3),T(4),T(5))

ENTROPY_TESTS = table(ENTROPY_TEST_NAME',
   ENTROPY_TEST_STAT');
ENTROPY_TESTS.Properties.VariableNames = ["Entropy Test
    Name","Test Value"];
writetable(ENTROPY_TESTS,xl_filename,'Sheet',sheets(
   corrected+1),'Range','F1','WriteVariableNames',true)

if corrected == 1
    break
end


%% ENTROPY ESTIMATION

% Shannon Entropy
Hs = Block_Entropy(bits);
ENTROPY_EST_NAME = [ENTROPY_EST_NAME "Shannon Entropy
   "];
ENTROPY_EST_VAL = [ENTROPY_EST_VAL Hs];
fprintf(" Stochastic Entropy: %f\n\n",Hs)

% Most Common Value Estimate
T = MostCommonValue(bits(1:2e6));
ENTROPY_EST_NAME = [ENTROPY_EST_NAME "Most Common Value
    Estimate"];
ENTROPY_EST_VAL = [ENTROPY_EST_VAL T];
fprintf(" Most Common Value Estiamte: %f\n\n",T)

% Markov Estimate
T = MarkovEstimate(bits(1:2e6));
ENTROPY_EST_NAME = [ENTROPY_EST_NAME "Markov Estimate
   "];
ENTROPY_EST_VAL = [ENTROPY_EST_VAL T];
fprintf(" Markov Estimate: %f\n\n",T)

```

```
% Tuple Estimate
T = TupleEstimate(bits(1:2e6));
ENTROPY_EST_NAME = [ENTROPY_EST_NAME "Tuple Estimate"];
ENTROPY_EST_VAL = [ENTROPY_EST_VAL T];
fprintf(" Tuple Estimate: %f\n\n",T)

% Most Common Window Estimate
T = MostCommonWindowEstimate(bits(1:2e6));
ENTROPY_EST_NAME = [ENTROPY_EST_NAME "Most Common Window Estimate"];
ENTROPY_EST_VAL = [ENTROPY_EST_VAL T];
fprintf(" Most Common Window Estimate: %f\n\n",T)

ENTROPY_ESTS = table(ENTROPY_EST_NAME',ENTROPY_EST_VAL');
ENTROPY_ESTS.Properties.VariableNames = ["Entropy Estimate Name","Estimate Value"];
writetable(ENTROPY_ESTS,xl_filename,'Sheet',sheets(corrected+1),'Range','I1','WriteVariableNames',true)


bits = VNB_Correction(bits);
corrected = 1;

disp("VON-NEUMANN BIT CORRECTION TESTS")
disp(repmat('~',1,45))

end
```

This code reads the binary data file of the binary symbol sequence and calls many function to run tests on the data. These tests include statistical measurements, NIST statistical tests, and NIST entropy estimators. This code also runs multiple times on corrected datasets. Results are written to a CSV file.

E.1.5 Bifurcation and Entropy

```
%% Bifurcation and Entropy Testing harness

% clear
close all
addpath("funnctions\")
addpath("NIST Entropy Tests\")
addpath("NIST Stat Function\")
addpath("Entropy Estimation\")
figure(1);
title('Bifuraction of Tent Circuit')
xlabel('Tent Slope')
ylabel('Possible Values')
slopes = [];
% Make multiple arrays for each entropy estimator
% Markov, Most Common Window, Tuple, Shannon, Topological
Mark_entropy = [];
slope_est = [];
MCV_entropy = [];
MCW_entropy = [];
Tuple_entropy = [];
Topological_entropy = [];
Block_ent = [];

% Desktop Sampled
volt_dir = 'C:\Users\user\Documents\folder_name\';
bit_dir = 'C:\Users\user\Documents\folder_name\';


volt_files = dir([volt_dir '*.bin']);
bit_files = dir([bit_dir '*.bin']);

T = struct2table(volt_files); % convert the struct array to a table
sortedT = sortrows(T, 'name'); % sort the table by 'DOB'
```

```
volt_files = table2struct(sortedT);

T = struct2table(bit_files); % convert the struct array to a table
sortedT = sortrows(T, 'name'); % sort the table by 'DOB'
bit_files = table2struct(sortedT);

% Loop through all collected data samples
for i = 1:length(volt_files)

    bin_file_name = fullfile(volt_files(i).folder, volt_files(i).name);
    fileID = fopen(bin_file_name);
    x = fread(fileID,'double')';
    fclose(fileID);

    %% IMPORT Binary Data
    bin_file_name = fullfile(bit_files(i).folder, bit_files(i).name);
    fileID = fopen(bin_file_name);
    bits = fread(fileID,'logical')'>0;
    fclose(fileID);

    %% create tent map from sampled data
    figure('DefaultAxesFontSize',24,'DefaultAxesFontName','Times New Roman', 'Color','White');
    scatter(x(1:end-1), x(2:end),'.');
    t = strcat("Tent Map ",string(i));
    title(t);%'Tent Map')
    xlabel('X(n)')
    ylabel('X(n+1)')
    axis tight
    hold on
    scatter(x,x,'.');
    hold off

```

```
%% measure the slope of the lhs
% Partition Tent
partition = FindPartition(x(1:400000));
fprintf('Partition: %f\n\n', partition)

% Slopes
[pL,pR,pA] = Calculate_Slopes(x(1:400000),partition
   );
fprintf(" Left Branch slope: %f\n Right Branch
   Slope: %f\n Overall Slope: %f\n\n",pL,pR,pA)

slope = abs([pL pR]);
% Take the unique values of the voltage samples
xu = unique(x);
figure(1);
hold on
scatter(min(slope(slope>1)).*ones(1,length(xu)),xu
   ,5,'blue','filled')
hold off

%% calculate the entropy from estimators

% Most Common Value Estimate
Tmcv = MostCommonValue(bits);
fprintf(" Most Common Value Estiamte: %f\n\n",Tmcv)

% Markov Estimate
Tmkv = MarkovEstimate(bits);
fprintf(" Markov Estimate: %f\n\n",Tmkv)

% Tuple Estimate
T_tup = TupleEstimate(bits);
fprintf(" Tuple Estimate: %f\n\n",T_tup)

% Shannon & Topological Entropy
% [Hs, Ht] = Calculate_Entropy(bits);
[Hs,block_lengths] = Block_Entropy(bits);
fprintf(" Stochastic Entropy: %f\n\n",Hs)
```

```
    % fprintf(" Topological Entropy: %f\n\n",Ht)
    Block_ent(i,:) = Hs;

    % store slope and entropy in array to plot later
    slopes(i) = min(slope(slope>1));
    slope_est(i) = log2(slopes(i));
    Mark_entropy(i) = Tmkv;
    MCV_entropy(i) = Tmcv;
    % Topological_entropy(i) = Ht;
    Shannon_entropy(i) = Hs(end);
    Tuple_entropy(i) = T_tup;



end
% end loop



%% Plot the entropy vs slope
figure('DefaultAxesFontSize',36,'DefaultAxesFontName','Times New Roman', 'Color', 'White');
plot(slopes, slope_est, '-diamond', ...
     slopes, Mark_entropy,'-r.', ...
     slopes, MCV_entropy, '-b*', ...
     slopes, Block_ent(:,7), '-k+', ...
     slopes, Tuple_entropy, '-^')

     title("Entropy Decay with Slope")
     xlabel("Measured Tent Slope")
     ylabel("Entropy (bits)")
     legend("Slope Estimate","Markov Estimate","MCV Estimate","Block Entropy","Tuple Estiamte")

%% 3d Block Entropy

figure('DefaultAxesFontSize',32,'DefaultAxesFontName','Times New Roman', 'Color', 'White');
```

```
for i = 1:length(volt_files)
    plot(block_lengths,Block_ent(i,:));
    hold on
end
legend(string(slopes),Location="southwest")
title("Entropy Measurements with Larger Block Sizes")
xlabel("Block Size")
ylabel("Entropy (bits)")
```

This code measures the bifurcation and entropy of data for tent maps at a variety of slopes. The code reads files in a fixed folder, generates a tent map from the data and calculates the slope. For each dataset, it plots a range of unique values on an orbit diagram and performs several entropy calculations. Any data stored in a binary file format can be input including simulated or measured data.

E.2 NIST 800-22 Statistical Tests

E.2.1 Frequency Monobit Test

```
function [P,pass] = Monobit(bits)
% Frequecny (Monobit) Test for randomness
%   This test determines randomness by looking at how equal the number of
%   ones and zeros are

% Count the number of bits
n = length(bits);

% Turn bits into +/-1
X = 2.*bits-1;

% Sum all values of x
Sn = sum(X);

% Test Statistic
Sobs = abs(Sn)/sqrt(n);

% Compute P-Value
P = erfc(Sobs/sqrt(2));

pass = P > 0.01;

end
```

The Frequency Test [25] and Frequency Test within a Block [25] measure how closely the dataset conforms to the ideal measure of mean. Half of all bits within a set should be one and the mean should be one-half. This property should be consistent for the entire set or subsets. Deviance from the ideal is measured with a chi-squared test. A computed P-value is compared to a known test statistic to determine if the dataset sufficiently matches the properties of an ideally random dataset.

E.2.2 Frequency Test within a Block

```
function [P,pass] = Block(bits)
% Freuency Test with a Block checks for even distribution of ones and zeros
%   Bits are goruped into blocks to check for an equal distribution of
%   values throughout the dataset. There should be fewer than than 100
%   blocks and the block size should be greater than 20
% M > 20
% N < 100
% n > MN

% Determine optimal Block size
M = 0;
N = 100;
while M < 20
    if M ~= 0
        N = N - 10;
    end
    M = floor(length(bits)/N);
end

% Loop through bits in blocks of size M, N times
proportions = [];

for i = 1:N
    proportions = [proportions sum(bits((i-1)*M+1:i*M))/M];
end

% Form the Chi-Squared distribution
chi_square = 4*M*sum((proportions-1/2).^2);

% Use inverse gamma function to determine P-value
P = gammainc(chi_square/2,N/2,'upper');

```

```
pass = P > 0.01;

end
```

E.2.3 Runs Test

```
function [P,pass] = Runs(bits)
%Runs counts the nuber of sequential bits that do not
   change
%   Detailed explanation goes here

% Number of bits in the array
n = length(bits);

% Number of runs in the bitstream
v = nnz(diff(bits))+1;

% Proportion of 1 bits in array
o = sum(bits)/n;

% Test statistic
P = erfc(abs(v-2*n*o*(1-o))/(2*sqrt(2*n)*o*(1-o)));

pass = P > 0.01;


end
```

The Runs Test [25] looks for substrings of consecutive matching values in the dataset. In a random distribution there is a precise likelihood of a run occurring, given the length of the dataset and the length of the run. Sets with more samples are expected to have runs with a longer maximum length as there are more opportunities for each sample to match its predecessor. Sequences with many long runs oscillate too slowly. Sequences with few runs oscillate too quickly. Both scenarios lead to highly predictable data with lower entropy. Similarly, the Test for the Longest Run of Ones in a Block [25] measures how commonly long runs occur in the set. The distribution of longest runs is compared to known probabilities of occurrence.

E.2.4 Longest-Run-of-Ones in a Block

```
function [P,pass] = LongestRun(bits)
%LongestRun counts counts the distribution of Runs in the array
%   The array is divided into blocks based on the length of the array.
%   Within each block the longest run of ones is counted. The distribution
%   of all the counted longest runs should fit a X-squared distribution.

% Length of the bit stream
n = length(bits);

% Choose Block Size and assign constants
if n >= 750000
    M = 1e4;
    K = 6;
    N = floor(n/M);
    count = [10 11 12 13 14 15 16];
    probs = [0.0882 0.2092 0.2483 0.1933 0.1208 0.0675 0.0727];
elseif n >= 6272
    M = 128;
    K = 5;
    N = floor(n/M);
    count = [4 5 6 7 8 9];
    probs = [0.1174 0.2430 0.2493 0.1752 0.1027 0.1124];
elseif n >= 128
    M = 8;
    K = 3;
    N = floor(n/M);
    count = [1 2 3 4];
    probs = [0.2148 0.3672 0.2305 0.1875];
else
    return
```

```
end

runs = [];

% Loop through array in blocks of size M
i = 1;
while i+M-1 < n

    % Separate bits into block
    block = bits(i:i+M-1);

    % Count of longest Run in block
    longest=max(accumarray(nonzeros((cumsum(~block)+1)
       .*block),1));

    % Apend value to array of runs
    runs = [runs longest];

    % Increment loop counter
    i = i + M;

end


% Create observed distribution

v = zeros(1,K+1);
for k=1:K+1
    % Count the occurences of runs
    if k == 1
        % min value
        v(k) = sum(runs<=count(k));
    elseif k == K+1
        % max value
        v(k) = sum(runs>=count(k));
    else
        % in between values
        v(k) = sum(runs==count(k));
```

```
    end
end

% Test Statistic
X = sum(((v-N.*probs).^2)./(N.*probs));

% Perform test
P = gammainc(X/2,K/2,'upper');

pass = P > 0.01;
end
```

The Longest Run of Ones in a Block test [25] is similar to the regular Runs test. It measures the longest run of ones in an M-bit block. The counts of runs should be distributed according to a predicted distribution. The test statistic is computed from the difference in the computed and measured distribution values. The test's P-value is the incomplete gamma function of the test statistic. Large deviations from the expected distribution of runs will result in a low P-value.

E.2.5 Binary Matrix Rank Test

```
function [P,pass] = MatrixRank(bits)
%MatrixRank Tests the linear dependence of fixed substrings of the sequence
%   Note: For this test, bits must be single or double

% Number of bits in array
bits = single(bits);
n = length(bits);

M = 32;      % Rows of Matrix
Q = 32;      % Columns of Matrix

N = floor(n/(M*Q));

% Loop through array of bits in MxQ length blocks determine Ranks
Ranks = zeros(1,N);

for i = 1:N
    A = reshape(bits((i-1)*(M*Q)+1:(i)*(M*Q)),M,Q);
    %Ranks(i) = rank(A);        OLD CODE

    % Forwards
    % loop through 1,1 to 31,31
    for j = 1:M-1
        % if A(j,j) == 0; then swap with another row that has a 1
        % if A(j,j) == 1; then XOR with other rows A(:,j) containing 1
        % else; continue the loop
        if A(j,j) == 0

            % Find any other rows with leading 1
            b = find(A(j+1:end,j)==1)+j;
            if ~isempty(b)
                % Swap row with first leading 1
```

```
                A([j b(1)],:) = A([b(1) j],:);
            end

        end
        if A(j,j) == 1

            % find other rows with ones
            b = find(A(j+1:end,j)==1)+j;
            if ~isempty(b)
                % XOR all found rows
                for k=1:length(b)
                    A(b(k),:) = xor(A(j,:),A(b(k),:));
                end
            end

        end
    end
    % Backwards
    % repeat from 32,32 to 2,2
    for j = M:-1:2
        % if A(j,j) == 0; then swap with another row
            that has a 1
        % if A(j,j) == 1; then XOR with other rows A(:,
            j) containing 1
        % else; continue the loop
        if A(j,j) == 0

            % Find any other rows with leading 1
            b = find(A(1:j-1,j)==1);
            if ~isempty(b)
                % Swap row with first last 1
                A([j b(end)],:) = A([b(1) j],:);
            end

        end
        if A(j,j) == 1

            % find other rows with ones
```

```
                b = find(A(1:j-1,j)==1);
                if ~isempty(b)
                    % XOR all found rows
                    for k=1:length(b)
                        A(b(k),:) = xor(A(j,:),A(b(k),:));
                    end
                end

            end
        end

        % r = sum(A,2);
        % Ranks(i) = sum(r~=0);
        Ranks(i) = sum(diag(A));
    end

    Fm = sum(Ranks == M);
    Fm_1 = sum(Ranks == M-1);
    Fm_rest = N - Fm_1 - Fm;

    % Calculate Test Statistic
    X = ((Fm-.2888*N)^2)/(.2888*N) + ((Fm_1-.5776*N)^2)
        /(.5776*N) + ((Fm_rest-.1336*N)^2)/(.1336*N);

    P = gammainc(X/2,1,'upper');

    pass = P > 0.01;

end
```

The Binary Matrix Rank Test [25] checks for linear dependencies in square matrices comprised of subsets of the original data. Rank is a measure of the number of columns in a matrix that are not linear combinations of other columns in the matrix. Most matrices should have a rank calculated near the maximum possible, with some having a lower rank with little linear dependence. The data should exhibit properties of linear independence to confirm the sampled random variable is uncorrelated and IID.

E.2.6 Discrete Fourier Transform (Spectral) Test

```
function [P,pass] = DFT(bits)
% Discrete Fourier Transform Test computes the number of peaks in the DFT of
% the sequence compared to the expected number

% Count number of bits in array
n = length(bits);

% Convert bit sequence
x = (2.*bits)-1;

% Disctrete Fourier Transform
S = fft(x,n);

% Absolute Vaue of first half of DFT values
M = abs(S(1:floor(n/2)));

% Calculate 95% peak height threshold
T = sqrt(log(1/.05)*n);

% Theoretical # of peaks
N0 = 0.95*n/2;

% Observed number of peaks under threshold
N1 = sum(M<T);

% Calculate Test statistic
d = (N1-N0)/sqrt(n*(.95*.05)/4);

% Compute P-value
P = erfc(abs(d)/sqrt(2));

% Plot FFT and Threshold
figure;
Fs = 358.45;         % change this to be dynamic variable
```

```
plot(Fs/n*(0:n-1),abs(S),[min(Fs/n*(0:n-1)) max(Fs/n
   *(0:n-1))],[T T])
xlabel('f (Hz)')
ylabel('|fft(bits)|')
title('Magnitude of FFT Spectrum')

pass = P > 0.01;

end
```

The Discrete Fourier Transform Spectral Test [25] utilizes analysis of the frequency domain to detect periodic features. These features can be observed graphically. A spectral graph with periodic components will have large magnitude spikes at certain frequencies. A sequence that conforms to the ideal will have a more uniform magnitude across all frequencies.

E.2.7 Non-overlapping Template Matching Test

```
function [P,pass] = LongestRun(bits)
%LongestRun counts counts the distribution of Runs in the array
%   The array is divided into blocks based on the length of the array.
%   Within each block the longest run of ones is counted. The distribution
%   of all the counted longest runs should fit a X-squared distribution.

% Length of the bit stream
n = length(bits);

% Choose Block Size and assign constants
if n >= 750000
    M = 1e4;
    K = 6;
    N = floor(n/M);
    count = [10 11 12 13 14 15 16];
    probs = [0.0882 0.2092 0.2483 0.1933 0.1208 0.0675 0.0727];
elseif n >= 6272
    M = 128;
    K = 5;
    N = floor(n/M);
    count = [4 5 6 7 8 9];
    probs = [0.1174 0.2430 0.2493 0.1752 0.1027 0.1124];
elseif n >= 128
    M = 8;
    K = 3;
    N = floor(n/M);
    count = [1 2 3 4];
    probs = [0.2148 0.3672 0.2305 0.1875];
else
    return
```

```
end

runs = [];

% Loop through array in blocks of size M
i = 1;
while i+M-1 < n

    % Separate bits into block
    block = bits(i:i+M-1);

    % Count of longest Run in block
    longest=max(accumarray(nonzeros((cumsum(~block)+1)
       .*block),1));

    % Apend value to array of runs
    runs = [runs longest];

    % Increment loop counter
    i = i + M;

end


% Create observed distribution

v = zeros(1,K+1);
for k=1:K+1
    % Count the occurences of runs
    if k == 1
        % min value
        v(k) = sum(runs<=count(k));
    elseif k == K+1
        % max value
        v(k) = sum(runs>=count(k));
    else
        % in between values
        v(k) = sum(runs==count(k));
```

```
    end
end

% Test Statistic
X = sum(((v-N.*probs).^2)./(N.*probs));

% Perform test
P = gammainc(X/2,K/2,'upper');

pass = P > 0.01;
end
```

E.2.8 Overlapping Template Matching Test

```
function [P,pass] = NonOverlapTemplate(bits)

n = length(bits);
m = 10;

B = zeros(1,m);
B(m) = 1;

% Calculate optimal block size
M = 0;
N = 100;
while M < .01*n
    if M ~= 0
        N = N - 4;
    end
    M = floor(length(bits)/N);
end

W = zeros(1,N);

% Double loop to search through blocks
for j = 1:N
    % Create block from bit array
    block = bits(M*(j-1)+1:M*j);
    k = 1;
    while k <= M-m
        % Search block for template
        if(block(k:k+m-1) == B)
            W(j) = W(j) +1;
            k = k + m;
        else
            k = k + 1;
        end
    end
end

```

```

mean = (M-m+1)/(2^m);

var = M*(1/(2^m)-(2*m-1)/(2^(2*m)));

% Compute Test Statistic
X = sum(((W-mean).^2)./var);

P = gammainc(X/2,N/2,'upper');

pass = P > 0.01;

end
```

The Non-Overlapping and Overlapping Template Matching Tests [25] measure the frequency of occurrence of predetermined strings in the data. Each arbitrary string should be found in the data roughly as many times as every other string. If certain strings occur far more frequently, then it indicates that the random number generator has a tendency to produce certain strings. This bias unwanted. The difference between the tests is in how they iterate their sliding window after finding a match. Both tests slide one bit if a match is not found. If a match is found, the overlapping test will slide one bit and the non-overlapping test will slide the length of the entire sequence.

E.2.9 Maurer's "Universal Statistical" Test

```
function [P,pass] = OverlapTemplate(bits)

n = length(bits);
m = 10;

% if n < 1e6
%     P = 0;
%     pass = 0;
%     return
% end

B = ones(1,m);

K = 5;

M = floor(sqrt(n)) + 2^K;

N = floor(sqrt(n)) - 2^K;

% Array to hold counts of occurences
V = zeros(1,6);

Blocks = reshape(bits(1:M*N),M,N)';

% Loop through blocks
for i = 1:N
    % Find occurences of B
    ind = strfind(Blocks(i,:),B);

    count = length(ind);
    % if count > 5, increment 5 counter
    if count > 5
        count = 5;
    end

    % increment counter in V
```

```
    V(count+1) = V(count+1) + 1;
end

% Compute Lambda and Mu

L = (M-m+1)/(2^m);

mu = L/2;

probs = zeros(1,6);

% Calculate theoretical probabilities
probs(1) = exp(-mu);

probs(2) = mu/2*exp(-mu);

probs(3) = mu/8*(mu+2)*exp(-mu);

probs(4) = mu/8*((mu^2)/6+mu+1)*exp(-mu);

probs(5) = mu/16*((mu^3)/24+(mu^2)/2+3*mu/2+1)*exp(-mu)
   ;

probs(6) = 1-sum(probs(1:5));


% Compute Test Statistic
X = sum(((V-N.*probs).^2)./(N.*probs));

P = gammainc(X/2,N/2,'upper');

pass = P > 0.01;

end
```

Maurer's Test [25] asserts that a quality random number generator should produce an output that is not compressible. A block is initialized with short sequences of bits. Measurements are taken to find the distance between the last occurrence of each sequence. The test statistic is the cumulative summation of the base-2 logarithm of the distances. This value is checked against known means and variances for a given sequence length.

E.2.10 The Serial Test

```
function [P1,P2,pass] = Serial(bits)

n = length(bits);
m = floor(log2(n)-log(n));
if m<3
    m=3;
end

V_count = zeros(3,2^m);


% Loop through each V array to look for occurences of bit sequences
for i = 0:2
    V = int2bit(0:2^(m-i)-1,m-i)';
    aug_bits = [bits,bits(1:m-i-1)];
    for j = 1:length(V)
        % find total occurences of each bit sequence in aug_bits
        inds = strfind(aug_bits,V(j,:));
        V_count(i+1,j) = length(inds);

    end
end

scale = [(2^m)/n; (2^(m-1))/n; (2^(m-2))/n];

psi = scale.*sum((V_count.^2),2) - n;

psi_del = psi(1)-psi(2);

psi_del2 = psi(1)-2*psi(2)+psi(3);

P1 = gammainc(psi_del/2,2^(m-2),'upper');
P2 = gammainc(psi_del2/2,2^(m-3),'upper');

```

```
pass = P1 > 0.01 && P2 > 0.01;

end
```

The Serial Test [25] observes the number of occurrences of small bit sequences. After appending the first two bits of the sequence to the end, a count of all overlapping combinations of three bit strings is taken. This is repeated for two bit and one bit string combinations. A test statistic is computed with the weighted sum of squares of the counts of each string. This tests for uniformity within the total dataset. Sets with large counts of a particular string will skew the test statistic and cause the test to fail.

E.2.11 The Approximate Entropy Test

```
function [P,pass] = ApproxEntropy(bits)

n = length(bits);

m = floor(log2(n)-log(n));

if m >= log2(n)-5
    m = floor(log2(n)-5);
end

count = zeros(2,2^(m+1));

% Perform steps for m and m+1
for i = 0:1
    % Augment bits by appending m-1 bits to the end
    aug_bits = [bits,bits(1:m+i-1)];

    % Array of bit sequences
    V = de2bi(0:2^(m+i)-1);
    for j = 1:length(V)
        % Count frequency of occurences of bit
           sequences
        inds = strfind(aug_bits,V(j,:));
        count(i+1,j) = length(inds);
    end

end

C = count./n;

C = C.*log(C);

C(isnan(C))=0;

phi = sum(C,2);

```

```
ApEn = phi(1)-phi(2);

X = 2*n*(log(2)-ApEn);

P = gammainc(X/2,2^(m-1),'upper');

pass = P > 0.01;


end
```

The Approximate Entropy test [25] checks the data's uniformity with a measure of Shannon entropy. The n-bit sequence is augmented by appending m-1 bits from the beginning onto the end so it can be broken into n overlapping m-bit strings. The frequency of each bit combination is measured and used to compute the Shannon entropy. The Process is repeated for m+1 length bit strings. The Approximate Entropy is the difference of the m-bit entropy and the (m+1)-bit entropy.

E.2.12 The Cumulative Sums (Cusums) Test

```
function [P,pass] = CumSum(bits,mode)

n = length(bits);

% convert to +/-1
X = 2*bits - 1;
z = 0;


% For mode 0
if mode == 0
    S = X(1);
    for i = 2:n
        S = S + X(i);
        if abs(S) > z
            z = abs(S);
        end
    end
end

% For mode 1
if mode == 1
    S = X(n);
    for i = 1:n-1
        S = S + X(n-i);
        if abs(S) > z
            z = abs(S);
        end
    end
end

% Compute P-value

zn = z/sqrt(n);
kStart = round((-n/z +1)/4);
kEnd = round((n/z -1)/4);
```

```
k = kStart:kEnd;
sum1 = 1/2*sum( ( 1-erf( -1*(4*k+1)*zn/sqrt(2) ) ) - (
   1-erf( -1*(4*k-1)*zn/sqrt(2) ) ) );

kStart = round( (-n/z -3)/4 );
k = kStart:kEnd;
sum2 = 1/2*sum( ( 1-erf( -1*(4*k+3)*zn/sqrt(2) ) ) - (
   1-erf( -1*(4*k+1)*zn/sqrt(2) ) ) );
P = 1 - sum1 + sum2;

pass = P > 0.01;

end
```

The Cumulative Sums test [25] is a measure of the uniformity of ones and zeros in a binary dataset with a random walk. The data is normalized from 1 and 0 to ± 1. A cumulative sum is taken over every bit in the sequence. The largest deviation from zero is tracked throughout the summation process. This summation should be tested twice, starting from each end of the sequence and summing forwards or backwards. A P-value is computed using the absolute maximum deviation in the Standard Normal Cumulative Probability Distribution Function.

E.2.13 The Random Excursions Test

```
function [P,pass] = RandomExcursion(bits)

x = 2.*bits -1;

S = cumsum(x);

S = [0, S, 0];


ind = strfind(S,0);
J = length(ind) - 1;

count = zeros(9,J);

% Loop through all crossings
for i=1:J
    % Loop through walks between crossings
    for j=ind(i):ind(i+1)
        % increment counter for levels -4,+4
        if S(j) ~= 0 && S(j) >= -4 && S(j) <= 4
            count(S(j)+5,i) = count(S(j)+5,i) + 1;
        end
    end
end

count = [count(1:4,:); count(6:9,:)];

cycles = zeros(8,6);
for k = 0:5
    if k ~= 5
        cycles(:,k+1) = sum(count == k,2);
    else
        cycles(:,k+1) = sum(count >= k,2);
    end
end

```

```
% Compute probability values
pi = zeros(9,6);

for k = 0:5
    for m = -4:4
        if k == 0
            pi(m+5,k+1) = 1-1/abs(2*m);
        elseif k == 5
            pi(m+5,k+1) = 1/abs(2*m)*(1-1/abs(2*m))^(4);
        else
            pi(m+5,k+1) = 1/(4*m^2)*(1-1/abs(2*m))^(k-1);
        end
    end
end

pi = [pi(1:4,:); pi(6:9,:)];

X = sum(((cycles-J.*pi).^2)./(J.*pi),2);

P = gammainc(X./2,5/2,'upper');

if J<500
    P=[0 0 0 0 0 0 0 0]';
end

pass = P > 0.01;

end
```

E.2.14 The Random Excursions Variant Test

```
function [P,pass] = RandomExcursionVariant(bits)

x = 2.*bits -1;

S = cumsum(x);
```

```
S = [0, S, 0];

states = [-9:-1,1:9];

ind = strfind(S,0);
J = length(ind) - 1;

o = zeros(1,18);

parfor i = 1:18
    o(i) = length(strfind(S,states(i)));
end

P = erfc(abs(o-J)./sqrt(2.*J.*(4.*abs(states)-2)));

if J<500
    P=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
end

pass = P > 0.01;

end
```

The Random Excursions and Variant tests [25] measure the distribution of values in a sequence as a function of the frequency of occurrence of values in the random walk. Similar to the Cumulative Sums test, the binary values are normalized and are summed. In the Random excursions test, the sum is broken into cycles based on where the sum crosses 0. A tally is made from the count of cycles where the values -4,-3,-2,-1,+1,+2,+3,+4 occur. A test statistic is computed for each of the eight states that were tallied. It is expected that values closer to 0 will occur more frequently than those farther from 0. The variant test differs in 2 main ways. The cumulative sum array is not broken down into cycles and it is tested against 18 states from -9 to +9. The counts are of each states occurrence in the whole cumulative array. We expect the distribution of states to fallow a normal distribution where states closer to 0 are measured more frequently than those farther away.

E.3 NIST 800-90B Entropy Estimators

E.3.1 Shannon Block Entropy

```
function [Hb,block_lengths] = Block_Entropy(bits)

%% Block Entropy

block_lengths = [1 2 4 6 8 12 16 20 24 36 48 64];

Hb = [];

% for each block size
for i=1:length(block_lengths)
    % Reshape vector of bits to matrix of non-overlapping blocks of length i in each
    % row
    Blocks = reshape(bits(1:block_lengths(i)*floor(length(bits)/block_lengths(i))),[],block_lengths(i));

    % Convert each row from a binary vector to a decimal value
    dec_blocks = binaryVectorToDecimal(Blocks);

    counts = [];
    % Find all unique values in the decinal array
    possible_vals = unique(dec_blocks);
    % for each unique value in the decimal array
    for k = 1:length(possible_vals)
        % count the number of occurrences
        counts(k) = sum(dec_blocks==possible_vals(k));
    end

    Hb(i) = 0;
    L = sum(counts);

    % Shannon's calculation of entropy
```

```
    Hb(i) = -1*sum(counts/L.*log(counts/L))/log(2^
       block_lengths(i));

end

end
```

This code implements Shannon's formula [26] for calculating symbol entropy of an arbitrary word length.

E.3.2 Most Common Value Estimate

```
function [T] = MostCommonValue(x)

L = length(x);
p = length(find(x==mode(x)))/L;


pu = min([1 p+2.576*sqrt(p*(1-p)/(L-1))]);

T = -log2(pu);

end
```

The Most Common Value Estimate [30] makes an approximation of entropy from the probability of the most common value in the sequence. The test statistic is calculated based on the distribution of the most common value and the length of the input sequence. As the sequence length increases to infinity the test statistic converges to the probability of the most common value and the entropy estimation converges to the negative base 2 log of that probability.

E.3.3 Most Common Window Estimate

```matlab
function [T] = MostCommonWindowEstimate(S)

S = S';
% S = S(1:2e6)';


%% Initialize variables
L = length(S);
% w = [3 5 7 9];
w = [63 255 1023 4095];
N = L - w(1);
correct = zeros(1,N);
scoreboard = zeros(1,4);
frequent = [];
winner = 1;

%% Generate Scorecard
for i = w(1)+1:L

    for j = 1:4
        if i>w(j)
            [M,F,C] = mode(S(i-w(j):(i-1)));
            mcv = cell2mat(C);
            if length(mcv) > 1
                % find which value came last
                lastloc = zeros(1,length(mcv));
                parfor k = 1:length(mcv)
                    lastloc(k) = find(S(i-w(j):(i-1))==mcv(k),1,'last');
                end
                frequent(j) = sum(mcv'.*(lastloc==max(lastloc)));
            else
                frequent(j) = mcv(1);
            end
        else
```

```
            frequent(j) = NaN;
        end
    end

    prediction = frequent(winner);
    if prediction == S(i)
        correct(i-w(1)) = 1;
    end

    for j = 1:4
        if frequent(j) == S(i)
            scoreboard(j) = scoreboard(j) +1;
            if scoreboard(j) >= scoreboard(winner)
                winner = j;
            end
        end
    end

end

%% Compute Performance indicators
C = sum(correct);
P = C/N;

if P == 0
    Pglobal = 1-0.01^(1/N);
else
    Pglobal = min(1,P + 2.576*sqrt(P*(1-P)/(N-1)));
end

% CRCT = 2.*correct-1;
M = max(accumarray(nonzeros((cumsum(~correct)+1).*
   correct),1));
r = M + 1;  % Holds the max run length.

Plocal = 0:1e-6:1;
m = floor(length(Plocal)/2);
l = 0;
```

```
k = 1;

while abs(0.99-l) > 1e-5
    q = 1-Plocal(m);
    x = 1;
    for j = 1:10
        x = 1 + q*Plocal(m)^r*x^(r+1);
    end

    l = (1-Plocal(m)*x)/((r + 1 -r*x)*q)*1/(x^(N+1));
    if 0.99-l > 1e-5
        m = m - round(length(Plocal)*1/(2^(k+1)));
    elseif 0.99-l <-1e-5
        m = m + round(length(Plocal)*1/(2^(k+1)));
    else
        break
    end

    k = k+1;
end

T = -log2(max([Pglobal Plocal(m) 1/L]));

end
```

The Multi Most Common in Window Estimate [30] uses an algorithm to track the accuracy of four different windowed prediction. As the four windows slide across the entire dataset, the most common value in the window will be set as the prediction. The algorithm counts how many times each window successfully predicts the next value. It also tracks which window most frequently guesses the next value correctly, calling it the *winner*. Every iteration, the winner gets to choose the prediction and every time it is correct a count is made. Two statistics are calculated. P_{global} is a measure of the number of correct guesses divided by the total number of samples. P_{local} is a measure calculated from the longest run of correct guesses. The smaller of the two statistics will be used to compute the entropy estimation as the negative log base 2 of the minimum statistic.

E.3.4 Markov Estimate

```
function [T] = MarkovEstimate(bits)

L = length(bits);
p1 = sum(bits)/L;
p0 = 1-p1;

C00 = length(strfind(bits,[0 0]));
C01 = length(strfind(bits,[0 1]));
C11 = length(strfind(bits,[1 1]));
C10 = length(strfind(bits,[1 0]));

P00 = C00/(C00+C01);
P01 = C01/(C00+C01);
P11 = C11/(C11+C10);
P10 = C10/(C11+C10);


probs = [p0*P00^127 p0*P01^64*P10^63 p0*P01*P11^126 p1*
    P10*P00^126 p1*P10^64*P01^63 p1*P11^127];

pmax = max(probs);

T = min(-log2(pmax)/128,1);

end
```

The Markov Estimate [30] uses the properties of a Markov process to test the entropy of a given binary dataset. A Markov process is only dependent on the previous value for the next one in the sequence. First the probabilities of each binary symbol are calculated, followed by the probability of all four state transitions. The likelihood of occurrence of six outcomes of 128 symbols are computed. The six sequences are all ones, all zeros, leading one followed by all zeros, leading zero followed by all ones, alternating bits starting with one, and alternating bits starting with zero. The entropy estimation is the negative base 2 logarithm of the most probable outcome divided by 128. For the case where this is larger than 1, the entropy estimation is 1.

E.3.5 Tuple Estimate

```
function [T] = TupleEstimate(S)

Q = [];
P = [];
Pm = [];

% Find way to make algorithm only check unique bit sequences (I don't need
% it to find the # of 1's and 0's 4 million times.

for u = 1:64
    b = buffer(S,u,u-1)';
    y = b(u:end,:);
    dec = binaryVectorToDecimal(y);
    counts = [];
    possible_vals = unique(dec);
    parfor k = 1:length(possible_vals)
        counts(k) = sum(dec==possible_vals(k));
    end
    if max(counts) < 35
        break;
    end
    Q(u) = max(counts);
    P(u) = Q(u)/(length(S)-u+1);
    Pm(u) = P(u)^(1/u);

end


Phat = max(Pm);
Pu = min(1, Phat+2.576*sqrt((Phat*(1-Phat))/(length(S)-1)));

T = -log2(Pu);

end
```

The Tuple Estimate [30] measures the frequency of subsequences of certain block lengths to estimate the entropy per sample. It uses an algorithm to determine the largest block size containing a sequence that occurs at least 35 times in the entire dataset. A maximum statistic is computed from the lengths of the most frequent subsequences for each block length. The negative base 2 logarithm of the statistic is taken as the entropy estimation.

E.3.6 Collision Estimate

```
function [T] = CollisionEstimate(bits)

L = length(bits);
v = 0;
index = 1;
t = [];

while index < L % step through sequence until you find a repetition
    j = 1;
    while length(bits(index:index+j-1)) == length(unique(bits(index:index+j-1)))
        j = j+1;
        if index+j>L
            break
        end
    end
    if index+j >L
        break
    end
    v = v+1;
    index = index + j;
    t(v) = j; %-index+1;

end

X = mean(t);
sig = std(t);

Xp = X - 2.576*sig/sqrt(v);

%% Simplified

r = roots([1 -1 (Xp/2-1)]);

p = max(r);
```

```
if ~isreal(p)
    p=1/2;
end

T = -log2(p);
```

The Collision Estimate [30] measures the distance between equivalent values. For binary data, this will commonly be a distance of only 2 or 3 symbols. The mean and standard deviation of the distances are computed. From the mean and standard deviation, the lower bound of the 99% confidence interval, X. In the original specification for the test, the confidence interval is used in a binary search for the parameters p and q which will be used to compute the entropy. A simplification can be made to forgo the binary search and speed up the computation of the entropy estimator [22]:

$$p = \frac{1 \pm \sqrt{1 - 2X}}{2}$$

. In the case that the polynomial has no real solutions, the entropy estimation is 1.

E.4 Miscellaneous Functions

E.4.1 Auto-correlation

```matlab
function AutoCorr(bits)
% Correlation will be 1 at zero as that is correlating the data with itself
% the shifted correlations should be close to zero. The more bits used the
% lower the noise floor.

% cross correlation with itself measures the statistical probability between two points
[cc,lags] = xcorr(bits-mean(bits), bits-mean(bits), 'coeff');
figure('DefaultAxesFontSize',10,'DefaultAxesFontName','Times New Roman', 'Color', 'White');
plot(lags, abs(cc));
title('Auto-Correlation')
xlabel('Shift Amount')
ylabel('Overlap')

count = 0;

for i = 1:length(cc)
    if abs(cc(i)) >= 0.01
        count = count + 1;
    end
end

count = count - 1; % remove the point at 0

percent = count/length(cc)*100;

fprintf(" %f%% of overlaps have over 1%% correlation\n\n",percent)
```

```
end
```

This code takes an array of data as an input, performs an autocorrelation operation and then plots the result.

E.4.2 Tent Slope Calculation

```
%% Calculate Slopes
function [slope_L,slope_R,slope_A] = Calculate_Slopes(x,partition)

% Import x sampled points
% Import partition

% Convert to a single Matrix of x(1:end-1) and x(2:end)
    % Transpose x matrix
x_array = [x(1:end-1); x(2:end)]';

% sortrows of x matrix and transpose
A = sortrows(x_array)';

% use partition to find last location of a 0
split = sum(A(1,:) < partition);

% split the x matrix into left and right halves
left = A(:,1:split);
right = A(:,split+1:end);

% use function polyfit to determine the approximate slope of the line
    % for the right side, use the abs value of the y-axis
pL = polyfit(left(1,:),left(2,:),1);
slope_L = max(abs(pL));
pR = polyfit(right(1,:),right(2,:),1);
slope_R = -max(abs(pR));

% Create continuous line
all = [[left(1,:); left(2,:)-max(left(2,:))] [right(1,:); abs(right(2,:))+max(right(2,:))]];

pA = polyfit(all(1,:),all(2,:),1);
slope_A = max(abs(pA));
```

```
% Create x and y vectors for linspace to plot slope
   lines
xL = linspace(min(left(1,:)),max(left(1,:)));
xR = linspace(min(right(1,:)),max(right(1,:)));

yL = polyval(pL,xL);
yR = polyval(pR,xR);

end
```

This code takes data from the chaotic oscillator and the partition of the tent map. It uses the known partition to separate data into two arrays, one for each half of the tent map. A polyfit function is run on each half of the tent map's data to approximate the slope of the map.

E.4.3 Partition Tent Map

```
function partition = FindPartition(x)

middles = [];
for r = 1:30
    % find indices of highest 30 samples
    [M,ind] = max(x(2:end));
    middles = [middles,x(ind)];
    x(ind+1) = -Inf;
end

partition = mean(middles);

end
```

This code computes the partition of the tent map. It finds the 100 highest points on the y-axis of the map and averages the corresponding x-axis values. The average of the x-axis points is returned as the map's partition.

E.4.4 Longest Repeated Sequence

```
function [last_size] = LongestSequence(s,t,vc,indeces,partition)
% Pass in bits, time, and sampled voltages
% Return the last size of the sequnces

start = 1;
size = 1;
locations = [];


% Search for longest repeated sequence
while start+size < length(s)
    % initial search
    locations = strfind(s(start+1:end),s(start:start+size-1));

    while ~isempty(locations)
        % increment size of sub-array
        size = size + 1;
        locations = strfind(s(start+1:end),s(start:start+size-1));

        if isempty(locations)
            % hold last location of longest run
            last_location = strfind(s(start+1:end),s(start:start+size-2));
            last_start = start;
            last_size = size-1;
        end
    end
    % shift sub-array over by 1
    start = start + 1;
end

% Find start and end locations of longest sequences
a_start = last_start;
```

```
a_end = last_start+last_size-1;

b_start = last_start+last_location;
b_end = last_start+last_location+last_size-1;

a_time_start = indeces(a_start-3);
a_time_end = indeces(a_end+3);

b_time_start = indeces(b_start-3);
b_time_end = indeces(b_end+3);

%%
% Plot longest repeated sequence
figure('DefaultAxesFontSize',18,'DefaultAxesFontName','
    Times New Roman', 'Color', 'White');
subplot(2,1,1)
hold on
% signals
plot(t(a_time_start:a_time_end)-t(a_time_start),vc(
    a_time_start:a_time_end),'-.','LineWidth',2.5)
plot(t(b_time_start:b_time_end)-t(b_time_start),vc(
    b_time_start:b_time_end),'LineWidth',2.5)

% partition line
plot([t(a_time_start)-t(a_time_start) t(a_time_end)-t(
    a_time_start)],partition.*[1 1],'-c',LineWidth=2)

% start/stop lines
plot([t(indeces(a_start))-t(a_time_start) t(indeces(
    a_start))-t(a_time_start)],[min(vc(b_time_start:
    b_time_end)) max(vc(b_time_start:b_time_end))],'-r','
    LineWidth',2)
plot([t(indeces(a_end))-t(a_time_start) t(indeces(a_end
    ))-t(a_time_start)],[min(vc(b_time_start:b_time_end))
     max(vc(b_time_start:b_time_end))],'-r','LineWidth'
    ,2)

% sampled points
```

```
plot(t(indeces(a_start-3:a_end+3))-t(a_time_start),vc(indeces(a_start-3:a_end+3)),'or','MarkerSize',10)
plot(t(indeces(a_start-3:a_end+3))-t(a_time_start),vc(indeces(b_start-3:b_end+3)),'ob','MarkerSize',10)
hold off
axis tight
legend('Signal 1','Signal 2','Partition',Location='southeast')
title('Synchronization Region')
xlabel('Time (s)')
ylabel('Voltage (v)')

% plot the differences between the two signals
subplot(2,1,2)
plot(t(a_time_start:a_time_end)-t(a_time_start),abs(vc(a_time_start:a_time_end)-vc(b_time_start:b_time_start+(a_time_end-a_time_start))),'-.','LineWidth',2)
axis tight
title('Difference in signals')
xlabel('Time (s)')
ylabel('Voltage (v)')

end
```

This code takes the symbol sequence, time series, time data, partition and an array of the indices of the samples from the times series as inputs. It recursively finds the longest symbol sequence that is repeated in the array. This repeated sequence corresponds to regions of the time series where the waveforms almost perfectly overlap and then diverge over time. The two time series waves and the difference between them is plotted.

E.4.5 Binary Data Binning

```
function bn = MakeBins(bits)

SL = 12; % Length of bit sequeces [1 0 1 1 0 0 1 1 1 1 0 0]
b = (1/2).^(1:SL);  % Array of fractional decimal values 0.5 to (0.5)^12
bn = []; % # of SL length sequences

% Loop through sets of 12 bits and multiply by their corresponding decimal
% values to get a final decimal value for a given set of bits

% Helps to show the likelyhood of certain bit combinations
for i = 1:SL:length(bits)-SL
    bn = [bn b*bits(i:i+SL-1)'];
end

fprintf(" Bins Average: %f\n Bins Variance: %f\n Bins Deviation: %f\n\n", mean(bn),var(bn),std(bn));

end
```

This code takes the binary symbol sequence as an input. It returns a set of fractional decimal values converted from an overlapping 12-bit window. It also computes the mean, variance, and standard deviation of the binned decimal values.

E.4.6 Von-Neumann Bit Correction

```
function vnbits = VNB_Correction(bits)

% This whitening algorithm removes instances where the circuit output can get "stuck"
% and have a long run. It will remove about 75% of the data, but the
% reamining bits will be highly uncorrelated.

% It will compare bits two at a time with no overlap and remove any 11 or
% 00 combinations 10 will become 0 and 01 will become 1.

k = 1;
for j = 2:2:length(bits)
    if bits(j-1) == 1 && bits(j) == 0
        vnbits(k) = 0;
        k=k+1;
    elseif bits(j-1) == 0 && bits(j) == 1
        vnbits(k) = 1;
        k=k+1;
    end
end


% Mean of the bitstream should be close to 0.5 or have an equal amount of
% 1s and 0s
fprintf(' Mean of Bits: %f\n', mean(vnbits))
fprintf(' Efficiency: %f%%\n\n', length(vnbits)/length(bits)*100)

end
```

This code takes the binary symbol sequence as an input and returns a binary sequence that has undergone Von-Neumann bit correction.

E.4.7 Voltage Histogram

```
function VoltHist(x)

% histogram of sampled voltage values

figure('DefaultAxesFontSize',20,'DefaultAxesFontName','Times New Roman', 'Color', 'White');
histogram(x,100)
title('Histogram of Sampled Voltages')
xlabel('Measured values')
ylabel('Count')
```

This code plats a histogram of the input voltage samples with 100 bins.

E.5 NIST 800-90B IID Permutation Testing

The following tests are taken from the NIST 800-90B document [30]. They create statistical measures of entropy sources. This data can be used to compare relative performance between two unique systems.

E.5.1 Conversion 1

```
function [x] = Conversion1(b)
% Convert a Binary sequence into integers
%   Detailed explanation goes here

b = [b zeros(1,8-mod(length(b),8))];
x = zeros(1,length(b)/8);

for i = 1:length(b)/8
    x(i) = sum(b((i-1)*8+1:i*8));
end

end
```

E.5.2 Conversion 2

```
function [x] = Conversion2(b)
% Converts binary blocks to decimal representation
%   Detailed explanation goes here

b = [b zeros(1,8-mod(length(b),8))];
x = zeros(1,length(b)/8);

for i = 1:length(b)/8
    x(i) = binaryVectorToDecimal(b((i-1)*8+1:i*8));
end

end
```

E.5.3 Excursion Test

```
function [T] = Excursion(x)
% This function measures the maximum deviation from the
    average value.

L = length(x);
m = mean(x);

k = (1:L).*m;

T = max(abs(cumsum(x)-k));
```

E.5.4 Number of Directional Runs

```
function [d] = DirectionalRunNum(x)

% Binary conversion I
if min(x)==0 && max(x)==1
    x = Conversion1(x);
end

S = 2.*(x(1:end-1)-x(2:end)<=0)-1;

% find number of runs in S
d = sum((S(1:end-1)+S(2:end)) == 0)+1;
end
```

E.5.5 Length of Directional Runs

```
function [T] = DirectionalRunLen(x)

% Binary conversion I
if min(x)==0 && max(x)==1
    x = Conversion1(x);
end

S = 2.*(x(1:end-1)-x(2:end)<=0)-1;

% find longest run of +/-1
[M,V] = regexp(sprintf('%i',[0 diff(S)==0]),'1+','match');
[M,I] = max(cellfun('length',M));
T = M + 1;  % Holds the max length.
end
```

E.5.6 Number of Increases and Decreases

```
function [T] = NumUpDown(x)

if min(x)==0 && max(x)==1
    x = Conversion1(x);
end

S = 2.*(x(1:end-1)-x(2:end)<=0)-1;

T = max([sum(S==1) sum(S==-1)]);

end
```

E.5.7 Number of Runs Based on the Median

```
function [T] = MedianRunNum(x)

if min(x)==0 && max(x)==1
    m = 0.5;
else
    m = median(x);
end

S = 2.*(x>=m)-1;

% find number of runs in S
T = sum((S(1:end-1)+S(2:end)) == 0)+1;
end
```

E.5.8 Length of Runs Based on the Median

```
function [T] = MedianRunLen(x)

if min(x)==0 && max(x)==1
    m = 0.5;
else
    m = median(x);
end

S = (x>=m);%2.*(x>=m)-1;

% find longest run of +/-1
[M,V] = regexp(sprintf('%i',[0 diff(S)==0]),'1+','match
   ');
[M,I] = max(cellfun('length',M));
T = M + 1;  % Holds the max length.
end
```

E.5.9 Average & Maximum Collision Test

```
function [aT, mT] = Collisions(x)

% Binary conversion II
if min(x)==0 && max(x)==1
    x = Conversion2(x);
end

L = length(x);
%%
i = 1;
C = [];

while i<L
    j = 2;
    while length(unique(x(i:i+j-1)))==length(x(i:i+j-1)) && i+j-1 < L
        j=j+1;
    end
    C = [C j];
    i = i+j;
end


aT = mean(C);
mT = max(C);

end
```

E.5.10 Periodicity Test

```
function [T] = Periodicity(x)

% Binary conversion I
if min(x)==0 && max(x)==1
    x = Conversion1(x);
end

p = [1 2 8 16 32];
T = [0 0 0 0 0];

for k = 1:length(p)
    count = 0;
    for i = 1:length(x)-p(k)
        if x(i)==x(i+p(k))
            count = count + 1;
        end
    end

    T(k) = count;

end

end
```

E.5.11 Covariance Test

```
function [T] = Covariance(x)

% Binary conversion I
if min(x)==0 && max(x)==1
    x = Conversion1(x);
end

p = [1 2 8 16 32];
T = [0 0 0 0 0];

for k = 1:length(p)
    T(k) = sum(x(1:end-p(k)).*x(1+p(k):end));
end

end
```