True Random Number Generation from a High Frequency Chaotic Jerk Circuit

by

Remington Chase Harrison

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
December 14, 2019

Keywords: chaos theory, nonlinear dynamics, random number generation, Dieharder, jerk chaos

Copyright 2019 by Remington Chase Harrison

Approved by

Robert N. Dean, Chair, McWane Endowed Professor of Electrical and Computer Engineering
Edmon Perkins, Assistant Professor of Mechanical Engineering
Lloyd Riggs, Professor of Electrical and Computer Engineering
Thaddeus Roppel, Associate Professor of Electrical and Computer Engineering

Abstract

Shown in this work is a method for true random number generation by directly sampling a high frequency chaotic jerk circuit. A method for determination of the maximum Lyapunov exponent, and thus the maximum bit rate for true random number generation, of the jerk system of interest is shown. The system is tested over a wide range of sampling parameters in order to simulate possible hardware configurations. The system is then implemented in high speed electronics on a small printed circuit board to verify its performance over the chosen parameters. The resulting circuit is well suited for random number generation due to its high dynamic complexity, long term aperiodicity, and extreme sensitivity to initial conditions. Also, a framework for evaluating other random number generation schemes based on chaotic systems is been given and is applicable to a wide variety of potential RNG solutions. This specific system passes the Dieharder RNG test suite at 3.125 Mbps.

Acknowledgments

I would like to thank my wife, Abbie, for supporting me over the course of our time at Auburn. We have made so many new friends here that have helped us and made our time here really special.

I would also like to thank my advisor, Robert Dean, for guidance through the graduate and doctoral process, and his help on a multitude of details relating to this work.

I would like to thank my advising committee of Thad Roppel, Lloyd Riggs, and Edmon Perkins, in their aid as well.

A special thanks goes to all of the members of the Nonlinear Dynamics Lab. I've enjoyed working with all of you.

Table of Contents

Abstract ii

Acknowledgments iii

1 Information, Entropy, and Randomness 1

1.1 Compression 2

1.2 The Coin Toss 6

1.3 Entropy 8

2 Overview of Chaos and Random Number Generators 11

2.1 Chaos 11

2.2 Random Number Generation 13

2.3 Laser Chaos and Random Number Generation 15

3 The Chaotic Jerk System 18

3.1 Background 19

4 Hardware Circuit Design 23

4.1 Circuit Simulation 23

4.2 Circuit Hardware 26

4.3 Compact and Low Power Implementation 32

4.3.1 Efficient Design of Chaotic Oscillators 32

4.4 Smaller Board Design 33

4.4.1 PCB Implementation 34

4.4.2 Low Power Improvements 38

5 Hardware Random Number Generation Results 41

6 Other Methods Of Obtaining Randomness From Chaos 48

6.1 Von Neumann Corrector 48

6.2 XOR Operation 55

7 Conclusion 59

8 Future Work 60

References 62

Appendices 70

A Compiling and Using Dieharder on Windows 71

B Creating a Printed Circuit Board using KiCad 75

C Code Sections for Various Functions 96

D Dieharder Output 114

List of Figures

1.1 A 256x256 pixel black image, containing very little information 4

1.2 A 256x256 pixel image in a checkerboard pattern, also containing very little information 5

1.3 A 256x256 pixel image generated randomly, containing much more information 6

1.4 A screenshot of the `ent` program running on two different files 10

3.1 Sensitivity to initial conditions in the jerk equation. The red asterisk denotes the point of divergence. 21

4.1 Block diagram of (3.2) 23

4.2 LTSpice simulation of an ideal circuit implementation of (3.2) 24

4.3 Time domain plot of $x(t)$ with $G = 1$ 24

4.4 Time domain plot of $x(t)$ with $G = 10\ M$ 25

4.5 Phase space of the chaotic equation 25

4.6 Real component implementation of (3.2) 27

4.7 LTSpice simulation of (3.2) with real components 28

4.8 Time domain plot of $x(t)$ for the simulated system 29

4.9 Phase space ($\dot{x}$ vs x) for the simulated system 29

4.10 Populated printed circuit board 30

4.11 Time domain plot of $x(t)$ 30

4.12 Phase space ($\dot{x}$ vs x) 31

4.13 $x(t)$ and $sgn(x(t))$ 31

4.14 Schematic of the resistor divider scaling the signum function 34

4.15 Resulting signum output showing bleedthrough 35

4.16 Simulated phase space $\dot{x}(t)$ vs. $x(t)$ 35

4.17 System schematic of the improved oscillator 36

4.18 Photograph of the assembled board 37

4.19 Captured continuous output $x(t)$ 37

4.20 Captured phase space of the assembled circuit board 38

4.21 Captured phase space of the assembled circuit board operating at 2.5 VDC . . . 39

4.22 The front and back of the populated circuit board which implements the jerk equation at 4 MHz. 40

5.1 Partial output of Dieharder testing. More tests and results are given in Appendix D than are shown here. 42

5.2 (a) 6th bit of an ADC sample of the system at 0.2 Hz with a full scale range of 2.5V. (b) 8th bit of an ADC sample of the system at 0.2 Hz with a full scale range of 2.5V. (c) 9th bit of an ADC sample of the system at 0.2 Hz with a full scale range of 2.5V. (d) 10th bit of an ADC sample of the system at 0.2 Hz with a full scale range of 2.5V. (e) 12th bit of an ADC sample of the system at 0.2 Hz with a full scale range of 2.5V. (f) 12th bit of an ADC sample of the system at 0.5 Hz with a full scale range of 2.5V. (g) 12th bit of an ADC sample of the system at 1 Hz with a full scale range of 2.5V. 44

5.3 The hardware circuit board sampled by the Handyscope HS6 inside of Multi-Channel. 46

5.4 Results from Dieharder testing of the hardware circuit. All bits were collected at 3.25 MHz with a full scale voltage of $\pm$2V. (a) 9th bit of each 16 bit sample (b) 10th bit of each 16 bit sample (c) 11th bit of each 16 bit sample (d) 12th bit of each 16 bit sample (e) 13th bit of each 16 bit sample (f) 14th bit of each 16 bit sample (g) 15th bit of each 16 bit sample (h) 16th bit of each 16 bit sample . 47

6.1 Results from simulated data with Von Neumann correction (bits 1-8) 51

6.2 Results from simulated data with Von Neumann correction (bits 9-16) 53

6.3 Results from hardware data with Von Neumann correction (bits 9-16) 54

6.4 Results from simulated data with XOR correction (bits 1-8) 56

6.5 Results from simulated data with XOR correction (bits 9-16) 58

A.1 The select packages screen within Cygwin Setup. Make sure that at least these packages are installed. 72

A.2 Cygwin's terminal 73

A.3 Dieharder output with no commands given 74

B.1 The homescreen of KiCad after a new blank project has been created 76

B.2 A blank EESchema schematic, with labelled items 76

B.3 Picking and placing a part withing EESchema 77

B.4 A basic EESchema schematic of the op amp circuit 77

B.5 A better EESchema schematic of the op amp circuit 78

B.6 Annotating the schematic 79

B.7 A screenshot of the Design Rules Check output with some warnings 79

B.8 A screenshot of the blank footprint assignment screen 81

B.9 A screenshot of the filled footprint assignment screen 81

B.10 A screenshot of the "Generate Netlist" screen in EESchema 82

B.11 A screenshot of the blank PCB layout in PCBnew 82

B.12 A screenshot of the read netlist window in PCBnew 83

B.13 A screenshot of the components ready to be placed in PCBnew 83

B.14 A screenshot of the layer setup in PCBnew 84

B.15 Beginning step of creating the edges of the PCB 85

B.16 A completed border that defines the edges of the PCB 85

B.17 The components moved to their desired locations, but unrouted 86

B.18 Selecting a copper zone to begin drawing 88

B.19 A copper zone drawn on inner layer 1 88

B.20 A copper zone drawn on the bottom layer 89

B.21 A copper zone drawn on the second inner layer for the power planes 89

B.22 A completed layout where all components are connected with no unrouted nets 90

B.23 Changing the names of the connectors on the PCB 91

B.24 The front silk screen with designators, references, and values 92

B.25 The front silk screen in 3D 93

B.26 The back silk screen in 3D 94

B.27 The completed board in 3D 94

B.28 Plotting each of the layers as a gerber file 95

B.29 Plotting the drill locations for the PCB 95

List of Tables

1.1 Summary of file sizes resulting from the previous exercise. 5

6.1 Bit Pair Probabilities of an Unbiased Sequence 49

6.2 Bit Pair Probabilities of a Biased Sequence 49

6.3 Von Neumann Corrected Sequence 49

6.4 The exclusive-OR truth table 55

Chapter 1

Information, Entropy, and Randomness

In order to discuss the technical aspects of a random number generator based on a chaotic system, the term random needs to be investigated first. Unfortunately, trying to simply define “random in a meaningful way proves easier said than done. A more apt approach perhaps is to look how randomness is derived from information. This somewhat philosophical method of approach is more roundabout than most, but hopefully will be fruitful in attempting to delve into this topic appropriately.

Information, for the purposes of this discussion, can be assumed to be infinitely precise and available. More specifically, any value that can be described, measured, or transferred can be done so to infinite precision, and there are an infinite number of such values. Of course, this assumption is highly debatable and not meant to build into a proof of anything. It is merely a baseline upon which to build an understanding of how humans understand and interpret information. One could argue that the total information contained in any fixed space is by definition not infinite by quantum limits, but even then the total information is so large that it is effectively freely available for the systems of interest. By making this assumption, a gradual building of this understanding into a framework for a random number generator can be achieved.

What, then, is information? In the modern digital age, information is usually measured in bits and bytes. There is a constant demand for both a higher throughput of data (colloquially referred to as “bandwidth”, and measured in megabits per second), and reducing the amount of data needed (referred to as compression). Often the abundance of available data for consumption gives rise to an “information overload” where it is difficult to sort through what is desired and what is not needed. Information itself is valuable to certain parties, most recently in the form of user data sold to third parties without consent. “Big Data” has become a rising area

of innovation in order to sort large amounts of data and see trends that otherwise could not be seen without accumulating these massive databases.

All of these are examples of what information can be, but none really capture the essence of what information is in a meaningful way as far as a random number generator. Information is an intuitive idea that most would be able to identify, yet would struggle to adequately and concretely describe without a reference. However, by building upon an understanding of this reference, certain properties of the reference will come to light and help form an understanding of information itself. For this discussion, data compression will serve as a technical reference to aid in realizing this definition.

1.1 Compression

Compression, in terms of a digital system, is the process of reducing the amount of data needed to store a particular piece of information, whether it be a picture, music, or text. There are a multitude of different algorithms for compression of data; a good resource for covering these algorithms specifically is [1].

First, consider some data that is uncompressed. This data need not be infinite or capture every minute detail of the pre-digitized work. This data is considered “raw”, meaning there has been no futher processing of this data other than the original digitization into bits. The data is more than likely a direct conversion using an analog to digital converter with a set resolution. The resulting data file is merely a “list” of these samples in sequential order, perhaps with some header information and metadata. Of course, this data format is not the most efficient form for storing the data since these raw formats do not take into account information about the data itself.

In order to increase the storage efficiency of this data, a compression algorithm is used to condense the stored data down to a smaller size. Instead of going into specifics on how these algorithms work, we will focus on what these algorithms achieve: lower storage requirements by simply describing the data in a different way. We will also consider what is left and what is discarded in the compression process.

Some compression algorithms are "lossless", meaning that they reduce the required amount of data needed to store all of the information without losing any information. This is most readily seen in the PNG and FLAC storage formats for images and music, respectively. This is achieved by removing redundant data that can be exactly replicated in the future or by storing differences in sequential bytes (which are usually much smaller than the full byte value). There are also "lossy" compression algorithms, such as JPG and MP3. These algorithms remove some information contained in the data in exchange for a much lower storage requirement. The "discarded" information is usually high frequency information that is at the extreme end of human hearing (in the case of music) or hard edge information in pictures, resulting in some blurring and "compression artefacts", but when viewed from a distance are not noticeable. Note that information is always lost in the original digitization process.

As an example, consider a completely black square image to be stored digitally (shown in Fig.1.1). This image was generated in Matlab by converting an array of zeros into a figure using `imshow`, and then saved in various file formats. The first file format, .bmp, is the stored sequential array of the values, and is uncompressed. The second file format, .png, is the image stored with lossless compression specifically for images. The third file format, .jpg, is a lossy compression algorithm for images. The fourth file format, .zip, is a generic lossless compression function given in Windows as `Send to -> Compressed (zipped) folder` in the .bmp file's right-click context menu. The .bmp format takes 65 kB of data to store digitally (close to 256*256*8 bits per pixel). The .jpg format is able to reduce this file size down to 1100 bytes. The .png format further reduces the file size to merely 160 bytes. The .zip format achieves a file size slightly higher than the .png format at 221 bytes (the extra data coming from increased header information). From the drastic reduction in file size by using these various compression algorithms, we can see that the original image contained very little information. This makes sense intellectually as well, since the entire image could be described with a few words (e.g. "256 pixel square entirely black"), which is a small amount of information.

Next, consider the same size image with a black and white checkerboard pattern (i.e. every pixel is opposite the ones next to it, with colors limited to black and white), as shown in Fig.1.2. This image was generated similarly to the previous one and again saved in the four different

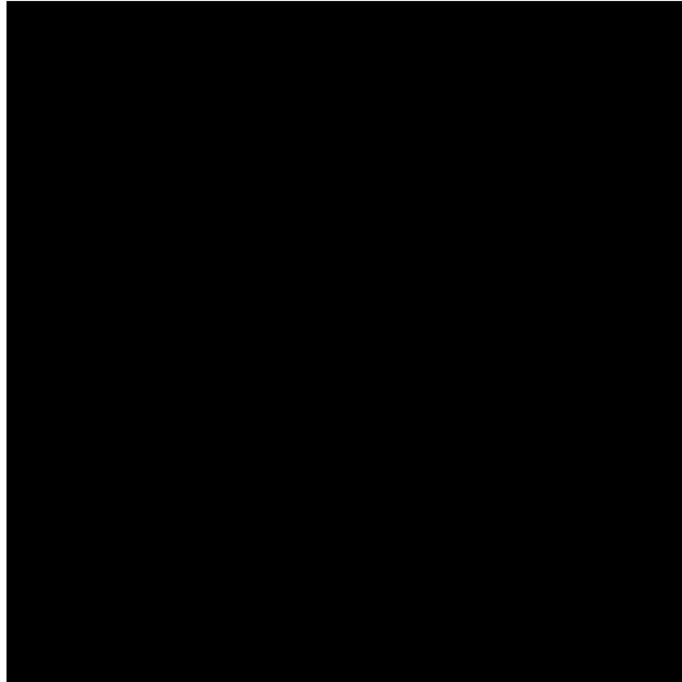

Figure 1.1: A 256x256 pixel black image, containing very little information

formats. The .bmp format takes 8.1 kB to store uncompressed. The .jpg format is unable to compress the picture and ends up taking 28.8 kB to store. In addition, it introduces artifacts (e.g. colors other than black and white) into the picture. However, the .png format is able to losslessly compress the image down to 135 bytes, with the .zip format taking up 211 bytes. Even though there is some "information" in this image (more so than the completely black image), the compression algorithms are able to reduce this image into a very small amount of "real" data that can completely replicate the picture. Again, this exact pattern is easily described in words, so the only unique properties of the image are the size and starting pixel color. These properties are able to precisely define the image.

Finally, consider an image with a random choice of black and white in each pixel (shown in Fig.1.3), the pattern being created using the `rand` function in Matlab and then processed in the same way as with the previous two images. The uncompressed .bmp again takes 8.1 kB to store. The .jpg format, much like the checkerboard pattern, fails at compressing the image and takes 44 kB to store (over 5 times the original size). Interestingly, the .png and .zip formats also both fail to compress the image, taking 8.2 kB to store. These results are summarized in Table 1.1.

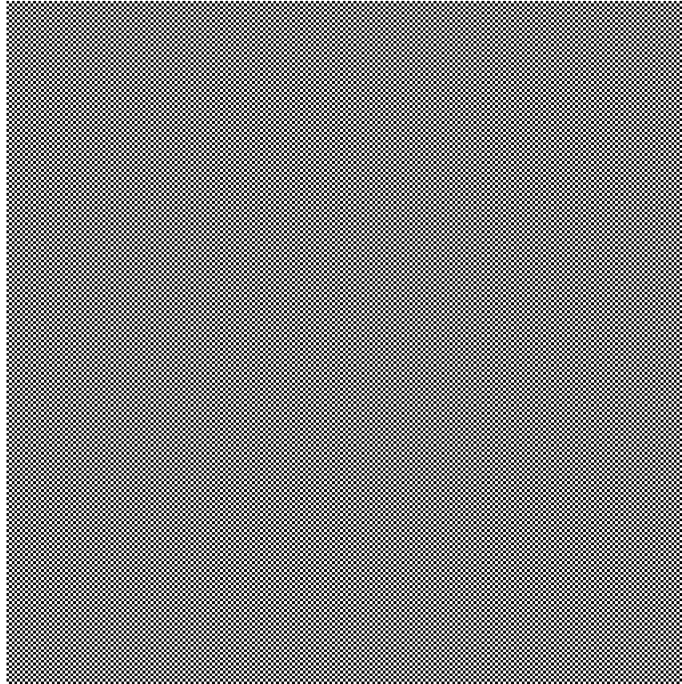

Figure 1.2: A 256x256 pixel image in a checkerboard pattern, also containing very little information

Table 1.1: Summary of file sizes resulting from the previous exercise.

	.bmp	.jpg	.png	.zip
black square	65 kB	1100 B	160 B	221 B
checkerboard	8.1 kB	28.8 kB	135 B	211 B
random	8.1 kB	44 kB	8.2 kB	8.2 kB

Why do all of the compression attempts fail on the last image? It is the same size as the other images and even has the same ratio of pixel color as the checkerboard image. Of course, it is because of the pattern of the individual pixels. In the checkerboard image, information about one pixel completely defines every other pixel. In the random image, there is no information gained about one pixel from examining any other (or all!) of the other pixels. Thus, the best that the compression algorithms can do it simply list each pixel in order. Hence, the storage requirement is very near the uncompressed .bmp size.

At this point, it can be said that information about something can be split into two parts. First, there is information that is unknown, unique, or "random". This is the information that in incompressible and cannot be reduced without removing properties of the item. Second,

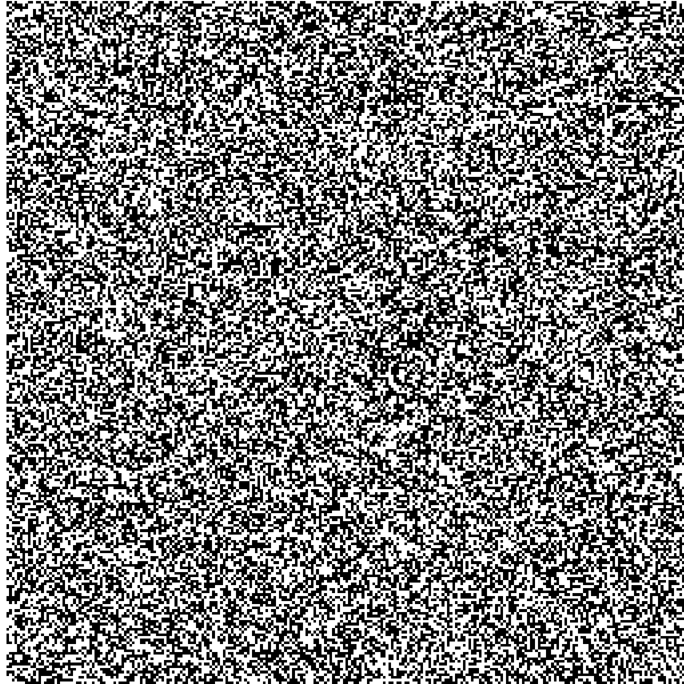

Figure 1.3: A 256x256 pixel image generated randomly, containing much more information

there is information that is known, predictable, or patterned. This is the part that compression algorithms attempt to reduce to the base “unique” part of the information, which can then be decompressed later. Some items, such as the black square, are almost entirely predictable. Others, like the checkerboard, are some balance between unique and patterned. Finally, there is the ”random”, which is the focus of much interest and research. The main goal of studying randomness and random phenomena is to learn what parts can be understood as patterned, and what parts are truly random. If one is able to remove the patterned part and only keep the random, then a random number generator can be built.

1.2 The Coin Toss

Perhaps the most iconic random number generator known today is the simple coin toss. This “heads or tails” device has decided outcomes with varying degrees of importance throughout history, from choosing between places to eat with friends or deciding team placement in sporting events, all the way to determining presidential candidates and other elected positions in the United States. A large emphasis is placed on the fairness of the chosen coin, with a

weighted or even “double-headed” coin universally seen as unfair as the outcome is skewed from a fifty-fifty distribution.

Despite the extensive use of the coin toss as a random number generator, its dynamics have been thoroughly studied and have been found to be quite predictable [2][3]. In order to perform a coin flip, the coin is placed either heads up or heads down on the thumb, and then the thumb flicks it upward. After the point at which the coin loses contact with the thumb, the “initial conditions” are set, meaning that the dynamics of the coin flip are allowed to propogate forward without further input. Air resistance, gravity, and impact with the ground are the forces that act upon the coin to ultimately determine the ending heads or tails decision.

By measuring the initial conditions of the coin (e.g. side initially up, rotational velocity, upward velocity, mass of the coin, etc.) with enough accuracy, and by knowing certain properties of the environment (e.g. coefficients of friction and restitution of the ground and coin, initial height above ground, etc.), the outcome of the coin flip can be confidently predicted [4]. With enough sophistication in equipment, multiple sets of initial conditions can be measured and mapped to their final outcomes, allowing rapid predictions in real time while the coin is still in the air. These heads or tails “basins” of initial conditions show that various ranges in specific initial conditions end up with the same outcome, meaning that measurement errors or inaccuracy within these ranges do not affect the predicted outcome.

Given that the coin toss is so readily predictable by modern standards, it seems odd that it is still so widely used with the assumption that the outcome is fifty-fifty. Most would realize that the coin toss somehow “predictable”, but would continue to trust in the process for decision making. Of course, in reality there is an understanding also that no one is measuring the coin’s initial conditions (beyond the the side up perhaps), and no one has ready access to equations of dynamics for the coin into which to input the conditions. Hence, there is a mutual *lack* of information during the coin toss among parties that prevents one from gaining a predictive advantage over the other. The information is technically available to any observer, and the final outcome *could* be calculated, but rarely would anyone pursue attempting to perform the prediction in real time.

So is the coin toss random? On one hand, there are accurate models for the coin toss dynamics that take into account a multitude of different toss variables and scenarios, and measuring these variables is fairly straight forward. With modern equipment, real time prediction is certainly feasible. In this sense, the coin toss is not random: Given enough information about the initial system, the outcome is confidently known. On the other hand, the dynamics of the coin toss map the initial conditions to an even distribution of heads and tails, and the initial conditions of the system are mostly unknown to the parties and unable to be accurately measured or influenced by a human to confidently move them between the basins of initial conditions. Even if steps were taken to measure the initial conditions, uncertainty in these measurements could be enough to change the predicted outcome such that the confidence in the prediction is no better than the original fifty-fifty chance.

In essence, there are parts of the coin toss that are "known", and there are parts that are "random". Specifically, the equations governing the trajectory of the coin while in the air and when impacting the ground have been thoroughly researched and can be considered known. The unknown part of the coin toss is only the initial condition of the system. The initial conditions could be known with enough determination, and thus an accurate prediction could be made, but in most situations this aspect of the coin toss remains hidden from all parties. This is the random part that most rely on in order to make decisions. Even though the coin toss is not precisely the ideal fifty-fifty chance that it is expected to be, it is good enough for most purposes.

1.3 Entropy

"Entropy" is a term that is sometimes interchangeably used with randomness and information in order to describe a sense of disorder or unpredictability. For this discussion, we will focus instead on the more tangible aspects of entropy as it relates to random number generation.

A system that has entropy tends to move from states of order to states of disorder. Order is denoted by stability, periodicity, or predictability. Disorder is denoted by instability, aperiodicity, or unpredictability. The entropy of a system is then a measure of the amount of that disorder compared to the order in the system, or the tendency of the order to transition into

disorder. Entropy has a well established background using thermodynamic principles. When considering each individual molecule of a system and the properties of them (known as “microstates”) as they relate to the properties of the overall system (known as “macrostates”), the thermodynamic definition of entropy S can be written as 1.1.

$$S = k_B \ln(\Omega) \tag{1.1}$$

In this definition, k_B is Boltzmann’s constant ($1.381 \times 10^{-23} J/K$), and Ω is the number of equiprobable microstates. It follows from this definition that a system with a large number of microstates has a high entropy, regardless of the macrostate of the system. Likewise, given the macrostate of the system, the entropy S is a measure of how much more information is needed to completely describe the system down to the microstate level.

This thermodynamic definition is very similar to the communication theory definition given by Shannon [5, 6]:

$$H = -\sum_{i=1}^{n} p_i \log_2(p_i) \tag{1.2}$$

Here, H is the Shannon entropy, and the p_i are the individual probabilities of each symbol. For a binary system, the entropy is maximized when $p_0 = p_1 = 0.5$. That is, when all probabilities are equally likely, then the least amount of information is known about the system until it is sampled. Likewise, once it is sampled, the maximum amount of new information is learned. For example, a communication scheme that only sends 0’s ($p_0 = 1, p_1 = 0$) has zero entropy since no information is actually communicated.

In a technical sense, the entropy of a certain bit sequence is a measure of the information contained in that sequence compared to the length of that sequence. This is usually measured in bits per byte, with 8 bits per byte being the maximum entropy measure possible. `ent` is a program available on Linux to quickly examine a bit stream or file and evaluate various aspects related to its randomness [7]. An example of this program running is shown in Fig. 1.4.

In the first file tested, the data in the file could technically have been represented with 6% less data (i.e. 6% of the data is redundant). This amounts to an entropy of approximately 7.44

```
C:\Windows\system32\cmd.exe

C:\Users\rch0012\Desktop\hardwaredata>random\ent simdata_B1.bin
Entropy = 7.441912 bits per byte.

Optimum compression would reduce the size
of this 2000000000 byte file by 6 percent.

Chi square distribution for 2000000000 samples is 1637486661.65, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 127.4995 (127.5 = random).
Monte Carlo value for Pi is 3.125246247 (error 0.52 percent).
Serial correlation coefficient is 0.006362 (totally uncorrelated = 0.0).

C:\Users\rch0012\Desktop\hardwaredata>random\ent simdata_B16.bin
Entropy = 8.000000 bits per byte.

Optimum compression would reduce the size
of this 2000000000 byte file by 0 percent.

Chi square distribution for 2000000000 samples is 251.54, and randomly
would exceed this value 54.96 percent of the times.

Arithmetic mean value of data bytes is 127.5004 (127.5 = random).
Monte Carlo value for Pi is 3.141563883 (error 0.00 percent).
Serial correlation coefficient is 0.000039 (totally uncorrelated = 0.0).

C:\Users\rch0012\Desktop\hardwaredata>
```

Figure 1.4: A screenshot of the `ent` program running on two different files

bits per byte. The second file, however, is uncompressible. The data within this file has the maximum entropy possible at 8 bits per byte. For both files, other metrics of randomness are given in addition to the entropy. The arithmetic mean of the file is the mean of all of the 8 bit byte values from 0 to 255 that are represented in the file. The Monte Carlo value for π is a calculation based on using successive byte values as 24-bit XY coordinate pairs within a unit square, and then measuring how many of those pairs land within the unit circle. The serial correlation is a measure of how each byte relates to the previous byte; correlation denotes predictability in the bit stream and thus redundant data.

Chapter 2

Overview of Chaos and Random Number Generators

2.1 Chaos

In the 1960's, in an attempt to create a model for weather patterns that had phenomena that were unexplained by previous linear models, Edward Lorenz discovered a system that had very peculiar properties [8]. When his system was simulated with very close initial conditions, the future outputs were vastly different. Robert May, when describing population densities, found an equation that did not have a steady state solution, was not periodic, and yet did not become globally unstable [9]. These, along with other discoveries in the field of nonlinear dynamics, ultimately led to the coining of the term "chaos" in order to describe these phenomena.

Today, chaos can be found in multiple areas, including phenomena in both natural and artificial events. For example, weather patterns are chaotic in the sense that they are able to be accurately predicted a few days in advance, but forecasts longer than a week suffer a drastic decrease in confidence. A normal pendulum has a simple periodic orbit dependent only on gravity and the length of the pendulum, but when an extra joint is added to the pendulum to make it a double pendulum, the motion of the end mass is complex and can complete full revolutions around the second joint. Fluid mixing results in an overall homogeneous mixture but has chaotic trajectories for the individual particles. Chaos also has strong similarities to fractal geometry found in many plants and rock formations.

Chaos, before the formal recognition that it currently has, was mostly bundled into the term "noise" and its effects were disregarded as unavoidable. While chaos does share a number of characteristics with noise, some properties set chaos apart as a unique phenomenon. There are also qualities of chaos that can be measured in a system of interest to quantify the existence of chaos in that system.

One of the first properties of chaos is known as sensitivity to initial conditions. As the designation implies, chaotic systems' future states have an extreme dependence on the starting conditions of the systems. Small changes in these conditions quickly multiply into large deviations in the eventual trajectories that the system will take, even in the absence of any other influence such as noise. Furthermore, extremely small perturbations in the system state at any time will cause the same divergence in trajectories.

Chaotic systems also exhibit long term aperiodicity even though they are cyclic. Specifically, chaotic systems can have a number of orbits that trajectories of the system will follow. Certain trajectories will take different amounts of time to complete one cycle of an orbit. Other trajectories will switch from one orbit to another for a time and then switch back. Each cycle of an orbit takes a slightly different amount of time, and thus there is no fixed period of the system. Phase space portraits and Poincare sections can shed greater insight into different orbits that a chaotic system can have.

From this, chaotic systems have a spread spectrum power density. Whereas normal periodic systems have their spectral power mostly at a natural or resonant frequency, chaotic systems have this power over a large frequency band or set of bands. Often this can extend from near 0 Hertz to the fundamental frequency of the system without having a large defined peak at any one point in that range.

Chaotic systems also exhibit what is known as topological mixing. This is an extension of both the properties of sensitivity to initial conditions and aperiodicity. A set of points close together on any trajectory of the system will, given enough time, become spread out over all of the orbits of the system. An analogy is mixing food coloring into a pudding: once it starts to be mixed, the color quickly becomes dispersed throughout the pudding until the mixture is homogeneous. At this point, the color and the pudding cannot be unmixed. Likewise, it is extremely difficult to determine the starting positions of the points in a chaotic system after this mixing has occurred.

All of these properties gives rise to another useful characteristic of chaotic systems: they are impossible to predict in the long term. For chaotic systems that can be defined as differential equations, the short term behavior can be predicted with high accuracy. This accuracy quickly

drops off due to sensitivity to initial conditions and aperiodicity to the point that long term behavior is essentially a random guess. At that point, new information needs to be gathered in order to be able to predict the system again.

Thus, chaotic systems are ideal for random number generation. They are naturally unstable and are aperiodic, which prevents prior information about the system from being used to determine long term behavior in the output. However, care must be taken so that random numbers are extracted from the system correctly such that statistical randomness is preserved.

2.2 Random Number Generation

Most randomly generated numbers today are generated with an algorithm designed to provide these numbers in a statistically uniform order. These algorithms are referred to as pseudorandom number generators. The increase in the reliance on random numbers has caused pseudorandom number generators to be present in almost every digital electronic system. Pseudorandom number generators are extremely fast and efficient, only limited by the clock speed of the algorithm and the power budget of the system. There are multiple different designs of these generators, including linear feedback shift registers, block ciphers, and stream ciphers [10][11].

However, as the name implies, pseudorandom number generators are not actually random. Pseudorandom number generators are meant to provide a source of statistical randomness, not true randomness. These generators are suitable for most everyday applications given that most generators are isolated enough from user interface, and most consumers are only concerned with the statistical randomness of random numbers. For extremely secure systems, this compromise is not acceptable.

There are a number of weaknesses in pseudorandom number generators that prevent them from being truly random. One of the most prominent issues of a pseudorandom number generator is that it will eventually repeat itself. Some of these generators have extremely long cycle lengths. Hence, there is a set amount of bits that can be taken from the generator before it is guaranteed to start its output sequence over. Also, some configurations of a pseudorandom number generator will not produce statistically random numbers. This is most prominently seen

in early configurations of linear feedback shift registers, where not only did the output repeat itself, it did so before all combinations of output were given. Pseudorandom number generator designs today go to great lengths to ensure that the output of the generator is as uniform as possible and as long as possible before it repeats itself.

Ultimately the output of every pseudorandom number generator can be predetermined given the initial states of the generator and the configuration of the generator. Since there are no outside states or influences to the generator, only an algorithm stored in memory, the future output is fixed. If an outside entity wanted to predict the output sequence from a pseudorandom number generator before they were actually generated, all that would be required would be to take the state of the generator and run the algorithm faster until the future output of the first generator was given by the second; no further information would be required. Normally, it is very difficult to get any information from a pseudorandom number generator aside from the output sequence, but once that information is known, the generator can be broken.

On the other hand, true random numbers are not based on a digital algorithm but rather on a physical process. The true randomness comes from the inability to get full information about either the system or the system state. For example, even if the dynamics of a physical system were hypothetically known exactly, the system output would still be unable to be predicted because the exact initial conditions cannot be measured. Likewise, if hypothetically the initial conditions could be exactly known, the future output would still be unable to be predicted because the system itself is never exactly known. In this way, physical processes provide a source of true randomness in that nothing can be fully known or predicted.

In order to be useful random number generators, physical processes still must provide statistical randomness as well as providing true randomness. Usually, the random numbers taken straight from the system will be biased in one way or another, and thus must be compensated for in a post-processing step. The most famous of these steps is the Von Neumann corrector, which takes a number sequence with a bit level bias and returns a new sequence where the bits are weighted equally.

Statistical randomness can be examined using a number of tests designed to evaluate large bit sequences for uniformity. Some programs that are publicly available include NIST and

Dieharder [12][13]. Ideally, a true random number generator will pass all of the performed tests with every bit sequence tested. A large amount of data must be given to these programs in order to assess randomness, and the physical process must be swept over its operating parameters in order to ensure that the system maintains its randomness throughout operation.

2.3 Laser Chaos and Random Number Generation

Lasers have provided incredible improvements and applications to a multitude of different areas, including manufacturing, medical, chemical, entertainment, and metrology [14][15][16][17][18]. Lasers' naturally small wavelengths allow extremely precise resolution when controlling the laser output spatially, while still being able to lase with a high output power. Usually, in order to keep the laser within a desired frequency range, a feedback mechanism is required; however, this is readily achieved with various techniques [19][20][21].

Random number generation is one of the foundations of modern cryptosecurity. Without a reliably random number source, encryption schemes for monetary and communication purposes would be much easier to break. As such, random number generators need to be inherently unstable, unable to be predicted, and yet easy to implement in a real system such that they can efficiently be used.

Stability in laser frequency is normally desired for most applications, since for these applications the laser energy needs to be confined to a small bandwidth compared to the natural frequency of the laser. However, for random number generation, the natural instability of the laser can be exploited. In addition, a laser can be made more unstable by placing certain feedback paths in the system that multiply the natural instability.

Note that laser instability for random number generation does not mean instability in the sense of lasing. Instability in normal laser systems means that there is no coherent beam because certain frequencies that the laser is designed to operate at are not supported by the optical geometry of the system or by any electronics providing amplification of the beam. Instability in these systems refers to a phenomenon where the system is globally stable but not locally stable, and is also not periodic. In a sense the laser output is random over a set of parameters, usually frequency or amplitude. Laser systems that exhibit this behavior are referred to as chaotic

systems. Chaotic phenomena in lasers have been reported and studied extensively in recent decades [22][23][24][25]. Chaos can be present in laser systems through various means, most of which involve a feedback mechanism [26]. In normal laser operation, feedback is used to keep laser frequency locked into a very narrow bandwidth. Chaotic laser operation requires this feedback to enhance the instability of the laser to the point that the nonlinear dynamics of the system overwhelm the normal operation. A large amount of research in laser chaos has been performed to mitigate or eliminate the effects of chaos in normal laser systems [27][28][29].

One method to induce chaos in a laser system is to couple a coherent, but much weaker, output of the laser back into the system [30]. By doing this, coherence in the system collapses and the laser is now no longer one specific frequency, but rather a broad range of frequencies, ultimately resulting in a noiselike spectra. The gradual increase in frequency content of a signal, noting higher order periodicitiy, is also known as period doubling [31].

Laser radar using a chaotically modulated carrier has been shown to provide a number of advantages over conventional radar, including low probability of intercept and increased resolution [32]. Creating a wideband signal for modulation at microwave frequencies presents a challenge for most radar systems, but chaotic signals have the necessary properties to give these benefits. Chaotic radar systems also have the ability to be synchronized with each other under certain conditions, which allows for a transmitter and receiver pair to operate theoretically undetected while still maintaining high resolution [33][34].

A lasers' naturally high frequency allows the chaotic phenomena to have a very high bandwidth, on the order of 1 GHz. This allows for random numbers to be generated more quickly than chaos found in other systems at lower frequencies, since a high throughput of random numbers in a random number generator is desirable. Some schemes for extracting random numbers from chaotic laser systems are shown below. Importantly, these schemes pass the random number generator tests to ensure that the output is statistically random in addition to being truly random.

One method of obtaining random numbers from a chaotic laser signal is to sample it with an analog to digital converter. This method is suitable for high speed random number generation because no post processing is necessary with a well designed system, which reduces system

complexity and increases robustness [35][36]. In these systems, the signal is sampled at a high sample rate in order to acquire a number of bits, e.g. an 8-bit sample. One of these bits is extracted and taken as a random bit. Then, another sample is taken and another bit extracted. The continuous operation of sampling and extraction creates a bit stream that can then be used as random numbers. Bit rates on the order of 10 Gbps are reported using this method. Additionally, by taking more than one bit per sample, higher bit rates can be achieved, but some statistical tests fail with the inclusion of more bits.

Another method is to sample the laser signal and then compare it with a time delayed version of the laser signal [37]. This requires storage of the laser signal for later processing, which might prohibit real time operation. First, the derivative of the laser signal amplitude is taken as the new signal in order to avoid biases in the final output. Then, a bit-wise XOR operation is performed between the signal sample and its time delayed derivative sample, and then a subset of the bits are taken as the random bits. In this system, where the lowest 5 bits of the XOR operation are taken at a sampling rate of 2.5 GHz, bit rates of 12.5 Gbps are achieved while passing all statistical tests.

Additionally, by sampling both the clockwise and counterclockwise lasing modes within a single cavity, multi-bit samples can be taken and then post processed similarly as above. This is an improvement to other similar systems that use two separate laser cavities to achieve the same statistical results [38]. Moreover, the optical feedback path chosen can be easily constructed in a semiconductor, which allows ready implementation with an analog to digital converter and any digital communication necessary. By taking 4 bits per sample, 40 Gbps is achieved in practice.

Chapter 3

The Chaotic Jerk System

Electronic implementations of chaotic systems offer a variety of useful applications. The deterministic yet unpredictable behavior of these types of systems lends well to inclusion in areas such as acoustic ranging [39] and automotive collision avoidance radar systems [40][41], random stimulating noise sources for MEMS [42], spread spectrum communication systems [43], and random number generation (RNG)[44] [45]. These chaotic systems have shown promise in having continuous power spectral density [46] as well as potential security against various attacks under which other systems could be susceptible [47][48].

The primary application of interest for the oscillator in this paper is true RNG. RNGs with high bit rates are increasingly needed as various areas of security become more reliant on electronic means. Chaotic oscillators are especially well suited for use as RNGs due to the long term unpredictable nature of the waveforms that are generated [49]. Moreover, these oscillators can be used as seeds for pseudo-RNG (PRNG). This allows for the relatively higher speed PRNG systems to be used more effectively [50, 51]. Different implementations of chaotic oscillators into RNGs have been shown to pass the NIST test [52]. In addition, chaotic circuits can be realized using minimal components and system footprint. [53].

Specifically, "double scroll" systems with discrete states similar to the one developed by Saito et. al [54] have an advantage in that one "scroll" can be mapped to a "1" and the other to a "0". These types of chaotic oscillators potentially could require deskewing [55]; however many developed systems achieve high bit rates without post processing any bits, thereby decreasing the delay in the desired bit stream [56].

3.1 Background

This work explores chaotic jerk systems, which are defined by a third order ordinary differential equation (ODE). These autonomous dissipative systems contain the lowest order derivative that can produce continuous timed chaos. The term jerk comes from the definition of successive derivatives in mechanical systems where x is the displacement, $\dot{x}$ is the velocity, $\ddot{x}$ is the acceleration, and $\dddot{x}$ is called the "jerk" [57].

Chaotic systems must include a nonlinear term in order to generate chaos. The intended application of interest is a RNG, so careful consideration was taken with regard to the potential nonlinearity. In order to be able to more easily produce random bits, discrete nonlinearities were considered. A candidate choice of a discrete nonlinearity can introduce discrete states to a continuous time equation, making it easier to generate a random bit stream from an autonomous system [58]. Sprott [58] gives many third order equations that exhibit chaos, but most contain nonlinearities that are difficult to implement accurately in electronics, such as multiplicative relationships, exponential functions, and trigonometric functions. However, one specific nonlinearity listed is the nonlinear switching event defined by the signum function, as defined in (3.1).

$$sgn(x) = \begin{cases} +1, & x \geq 0 \\ -1, & x < 0 \end{cases} \tag{3.1}$$

By using this nonlinearity in a continuous time system, this system is less complex and yet more robust than an equivalent discrete time system, even though continuous time systems can potentially undergo synchronization using a master-slave oscillator system [59].

Taking this specific nonlinearity into consideration, one of the equations given in [58], is much simpler than the others:

$$\dddot{x} = -0.5\ddot{x} - \dot{x} - x + sgn(x) \tag{3.2}$$

This system was carefully chosen because these functions can easily be implemented in mixed signal electronic circuitry. The advantage of this discrete nonlinearity is that it can be implemented in electronic circuitry by using a single comparator. This system's structure and its state feedback topology are composed of a weighted summation of the system's states and the output from the discrete nonlinearity. These mathematical functions can be implemented in electronics by using a string of operational amplifier (op amp) integrators, a comparator, and an analog summing amplifier. Implementation of this system in MOS has been proposed and designed with desirable simulation results in the kHz frequency range, but has yet to be physically realized [60]. These circuits could potentially require specialized MOS fabrication techniques in order to reach high frequencies. The electronic design presented here requires a small number of commercial off-the-shelf (COTS) components in order to be implemented. This is advantageous in scaling the frequency of the design up to 4 MHz since the propagation delay required to complete the feedback path is minimized.

Next, the maximum Lyapunov exponent (MLE) of the system needs to be estimated so that the maximum bit rate for random number generation can be found. The method chosen to accomplish this is a direct measurement of the divergence rate of many pairs of almost identical trajectories in simulation. The sensitivity of these chaotic systems to initial conditions causes trajectories that are different only by an amount well below measurement thresholds of real systems to quickly diverge. The MLE is then calculated by comparing the inital offset between the two systems with the time it takes for the difference in trajectories to reach a chosen threshold. This calculation is given with the following equation

$$MLE = \frac{ln(\frac{threshold}{offset})}{time} \tag{3.3}$$

where threshold is the chosen divergence limit, offset is the initial difference in states of the two trajectories, time is the final time taken to reach the threshold, and ln is the natural logarithm. Then, this translates into a theoretical maximum bit rate as follows

$$bitrate = \frac{MLE}{ln\,2} \approx 1.443 * MLE \tag{3.4}$$

The bit rate given in (3.4) has units of s^{-1}.

For the jerk system, many time domain simulations were performed in order to determine the MLE by using a MATLAB script to implement the differential equations with a fixed time step. Initial conditions for both systems (the x, $\dot{x}$, and $\ddot{x}$ states) were randomized such that they were between -1 and $+1$ so that the trajectories remained in the chaotic region of the attractor rather than becoming globally unstable. Then an very small offset (between 10^{-12} and 10^{-8}) was applied to the second system's x state. The threshold limit was chosen to be between 10^{-4} and 10^{-1}. An example plot of divergence in the two trajectories for one set of the chosen offset and threshold is shown in Fig. 3.1. In this figure, an initial condition for the systems is chosen and then one is separated by a small amount ($d0 = 5{*}10^{-12}$ for this example). Then the systems are simulated forward in time until they reach the specified threshold (here, $thresh = 1 * 10^{-2}$ and is marked with an asterisk). After this point, the systems visibly diverge quickly.

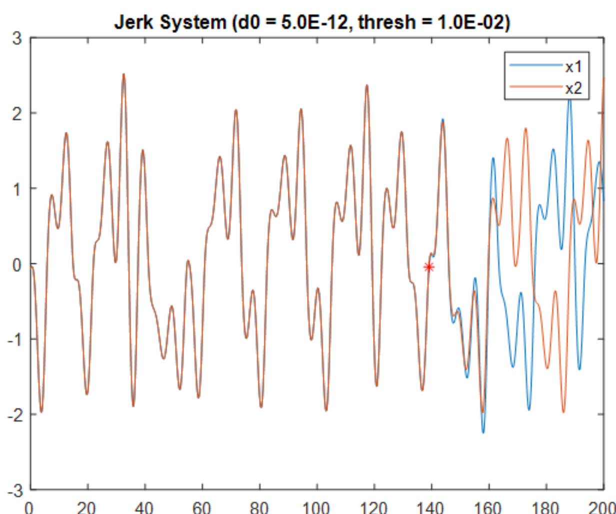

Figure 3.1: Sensitivity to initial conditions in the jerk equation. The red asterisk denotes the point of divergence.

The resulting MLE from many simulations with various initial conditions, thresholds, and offsets was between 0.152 and 0.153 for the jerk system. This then gives a bit rate of 0.218 and 0.221 bits per second. Sprott reaches a very similar result for an MLE of the same system when using another method with some approximations [61]. It can be seen visually that the system has a natural "frequency" of oscillation (when considering the time to complete one orbit around half of the attractor) of approximately $0.2Hz$. Thus, the bit rate of the system

itself is around 1 bit per cycle. By framing the bit rate this way, the system can be scaled to any frequency and the bit rate will remain constant with respect to the system. Specifically, when the system is implemented at high frequency in an electronic circuit, the circuit will be able to generate random bits at this higher natural frequency, as long as the system is represented accurately.

Chapter 4

Hardware Circuit Design

4.1 Circuit Simulation

A block diagram of the chaotic jerk system in state feedback form is shown in Fig. 4.1. The system's ideal dynamics were first simulated using ideal component models in LTSpice as shown in Fig. 4.2. The parameter "G" sets the overall loop gain of each of the integrating stages. Figs. 4.3 and 4.4 show the time domain of $x(t)$ for various values of G. The phase space ($\dot{x}\ vs.\ x$) of the chaotic equation is shown in Fig. 4.5. Demonstration that the fundamental frequency of oscillation increases as the loop gain G increases is shown in Figs. 4.3 and 4.4.

Thus, in order to achieve the highest possible frequency in the real circuit, the gains of the integrating stages must be very high. A standard operational amplifier integrator has a transfer equation of (4.1).

$$v_o = \frac{-1}{RC}\int_0^t v_{in} dt \tag{4.1}$$

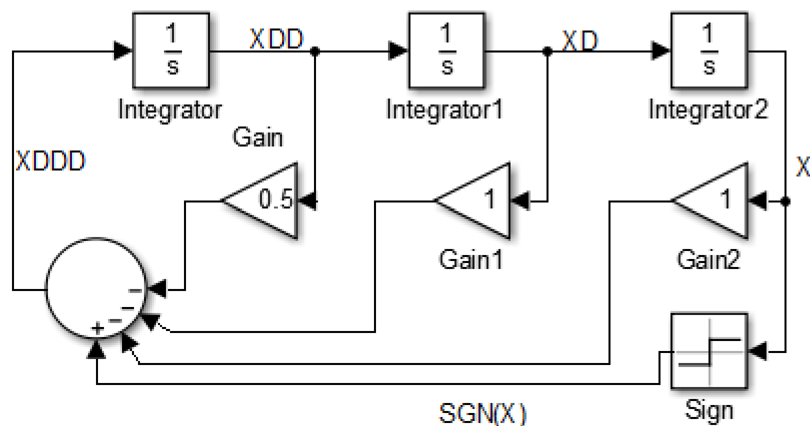

Figure 4.1: Block diagram of (3.2)

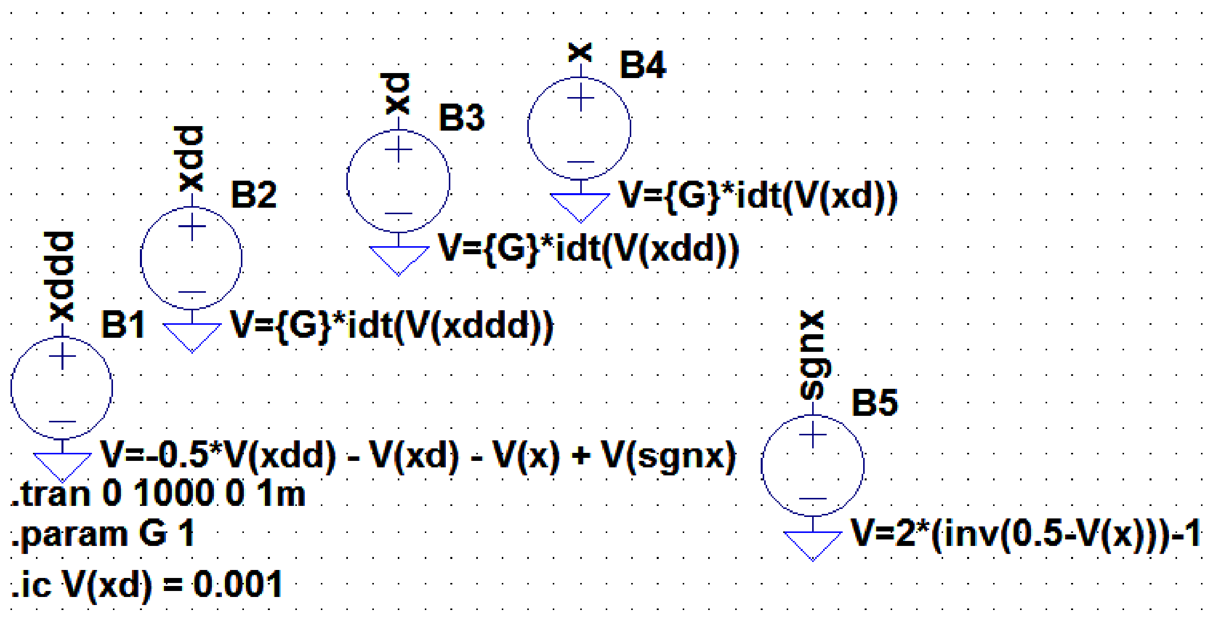

Figure 4.2: LTSpice simulation of an ideal circuit implementation of (3.2)

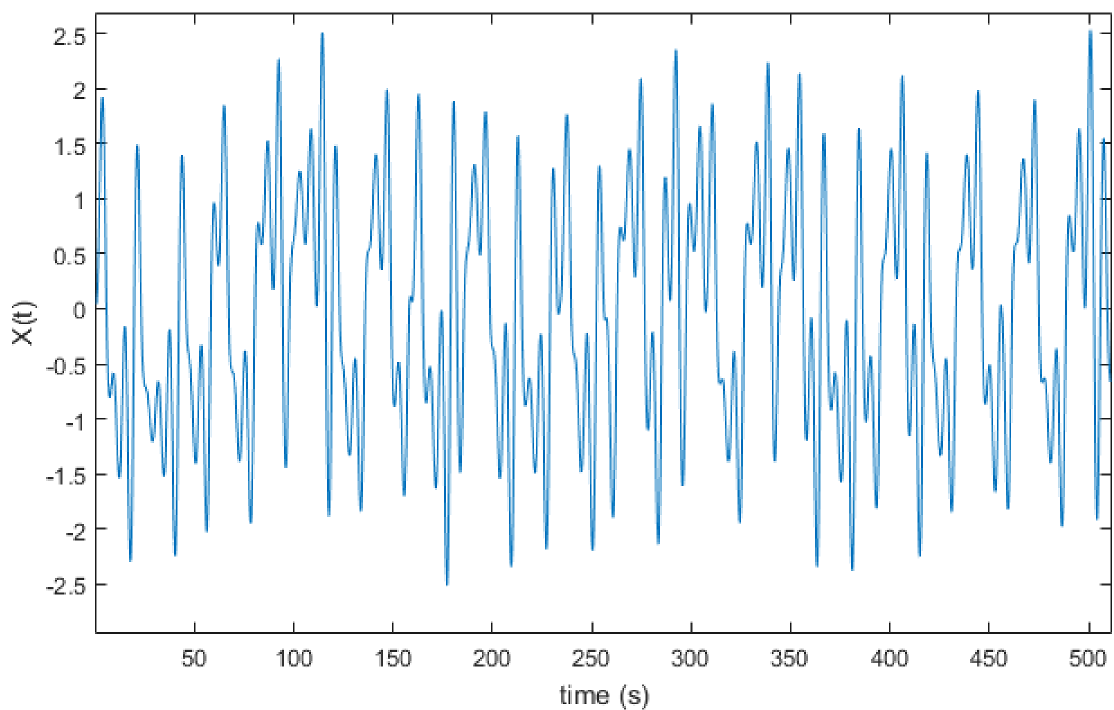

Figure 4.3: Time domain plot of $x(t)$ with $G = 1$

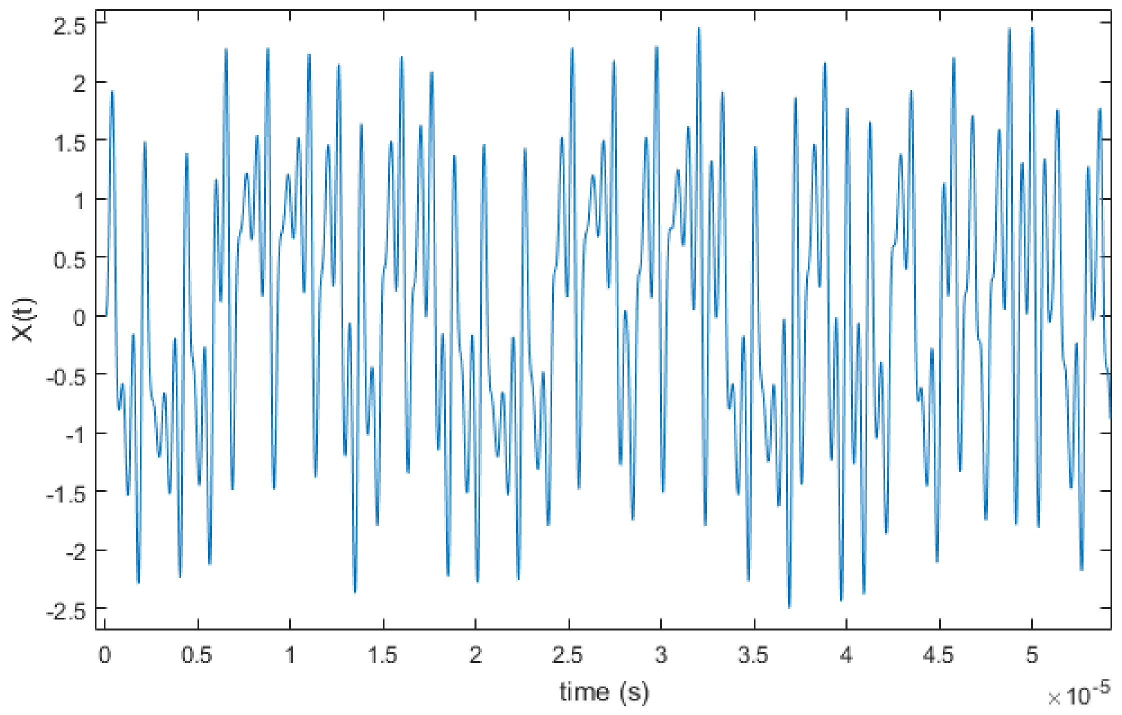

Figure 4.4: Time domain plot of $x(t)$ with $G = 10\ M$

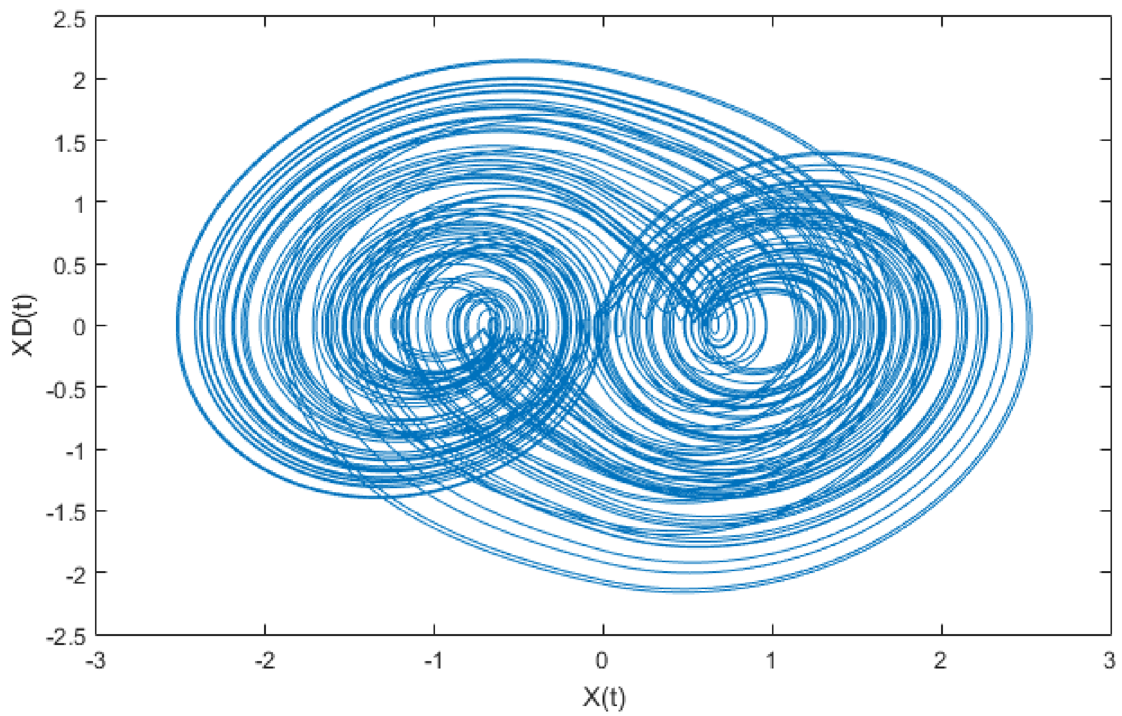

Figure 4.5: Phase space of the chaotic equation

Hence, the gain of each of the integrating stages is set by the values of the input resistor (R) and feedback capacitor (C), and also is always negative. This is accounted for in the summing amplifier as the polarity of each stage will be the opposite of the stage before, but the polarity of each term in (3.2) is referenced to $\dddot{x}$.

However, the frequency of oscillation cannot be increased indefinitely. Real operational amplifiers have finite gain bandwidth products (GBP) and slew rate limitations. Since the gain of the op amp integrators must be high, the primary frequency limitation of this implementation is the GBP of the op amp. Equation (3.2) assumes that there is no propagation delay through the string of integrators and comparator. In practice, the propagation delay through these devices can cause signals that are assumed to be "in sync" to be "out of sync". Additionally, non-ideal op amp phase responses become more pronounced at higher frequencies that could potentially be detrimental to the desired chaotic operation.

A simulation with models for real components is shown in Fig. 4.6. This circuit was simulated in LTSpice as shown in Fig. 4.7. LT1818 op amps were chosen for their high bandwidth, low propagation delay, and ability to operate from differential power supplies. The LTC6752-2 comparator has separate input and output power supply power rails. This means that the comparator's output can be scaled to $\pm 1V$ to achieve proper scaling of the signum function. Also, feedback resistors were added in parallel with the feedback capacitors in order to prevent integrator latch up. Figs. 4.8 and 4.9 show the time domain and phase space of this circuit simulation, respectively. These component model simulations are in agreement with the ideal simulations at the target frequency of 4 MHz. Unfortunately, when the resonant frequency is made any higher by decreasing the feedback capacitor value, the phase space becomes increasingly distorted until the circuit is no longer chaotic. Thus, although the circuit could potentially be made to oscillate at a higher frequency, the component values were chosen so that the circuit maintains a clean phase space and time domain signal.

4.2 Circuit Hardware

A 4-layer printed circuit board (PCB) was developed using KiCad. A tutorial for creating PCBs within KiCad is given in Appendix B. This prototype maintains the core components

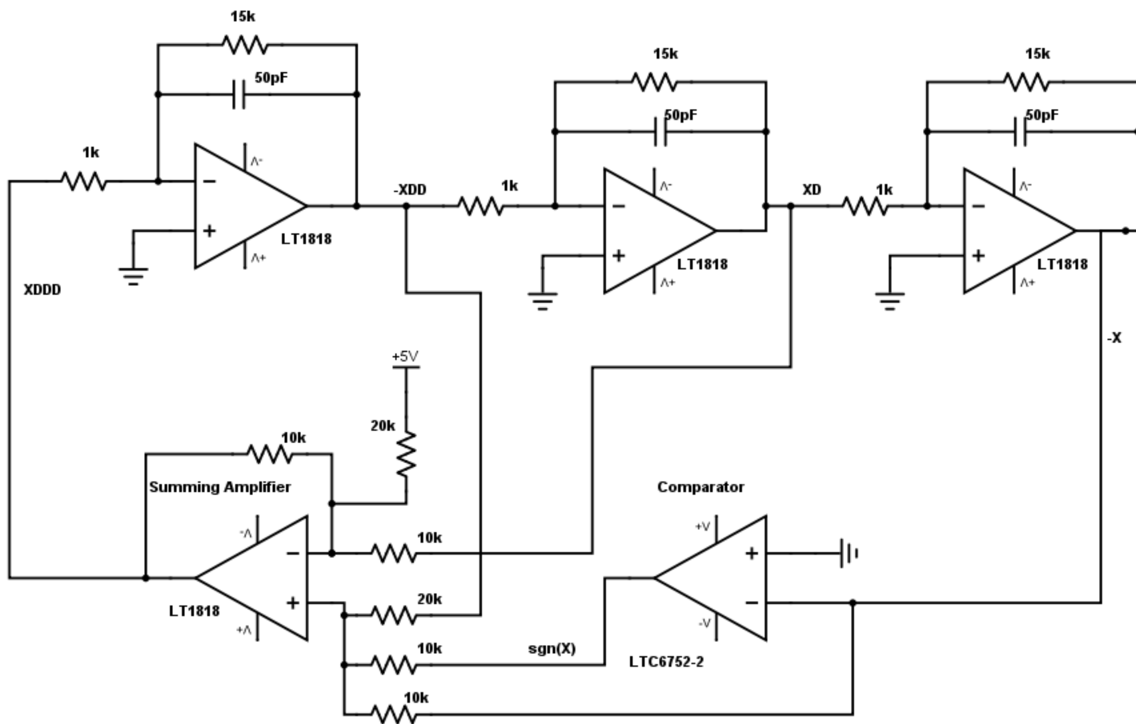

Figure 4.6: Real component implementation of (3.2)

from the schematic in Fig. 4.7; however, additional components such as headers and decoupling capacitors were added for probing and filtering the power supply. A photograph of the populated PCB is shown in Fig. 4.10. Time domain and phase space plots of the hardware circuit are shown in Figs. 4.11 and 4.20, respectively. An oscilloscope screen capture of the signum function output of the circuit is shown in Fig. 4.13. This circuit is shown populated to oscillate at the target 4 MHz fundamental frequency.

The hardware circuit is in close agreement with both the ideal simulation and the real component simulation when comparing both the time domain and phase space plots. Moreover, the signum output shown can be used as the random bit stream for a RNG. The circuit draws approximately 40 mA from both the +5V and -5V power supplies, and draws approximately 2mA from both the +1V and -1V power supplies.

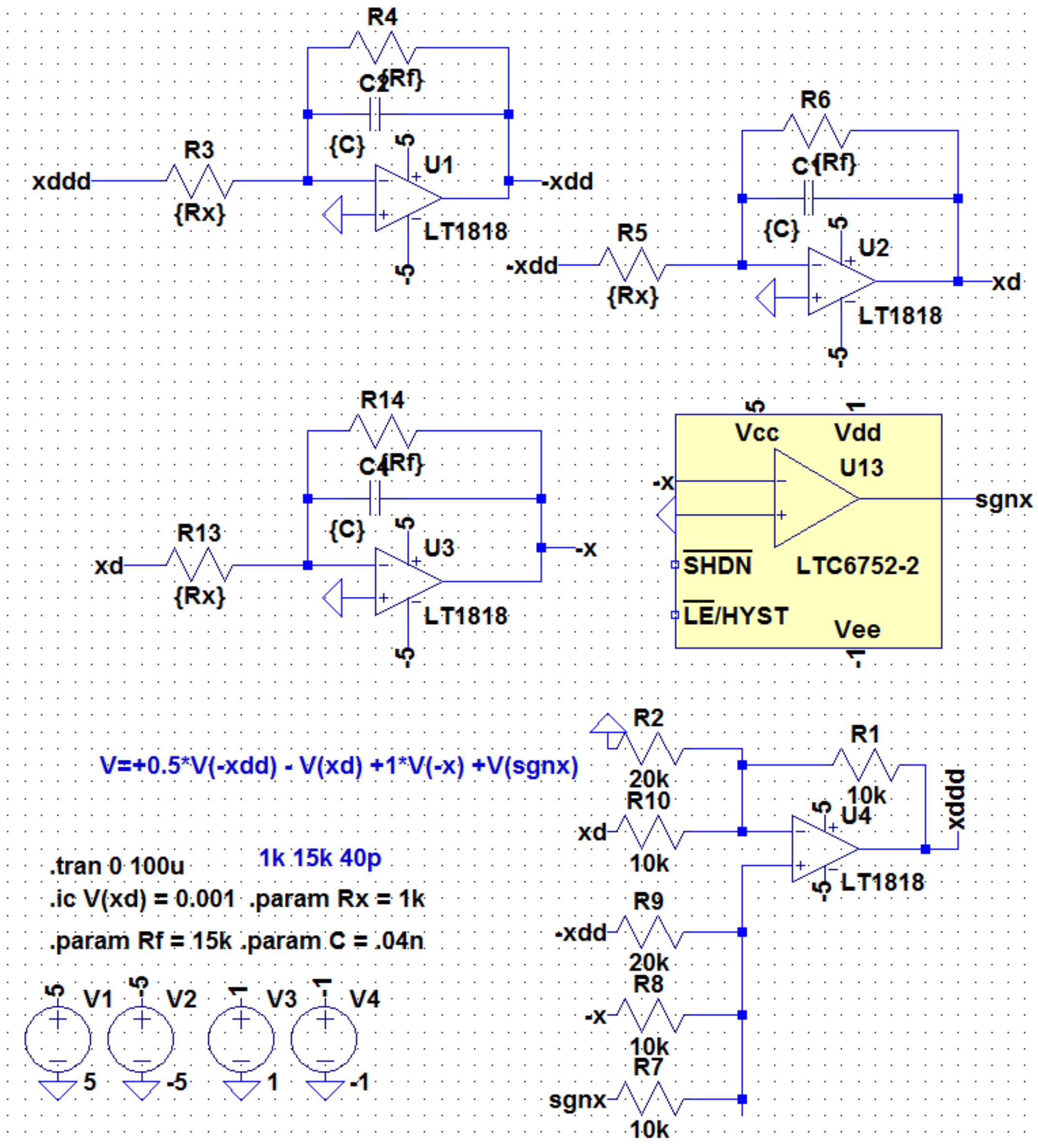

Figure 4.7: LTSpice simulation of (3.2) with real components

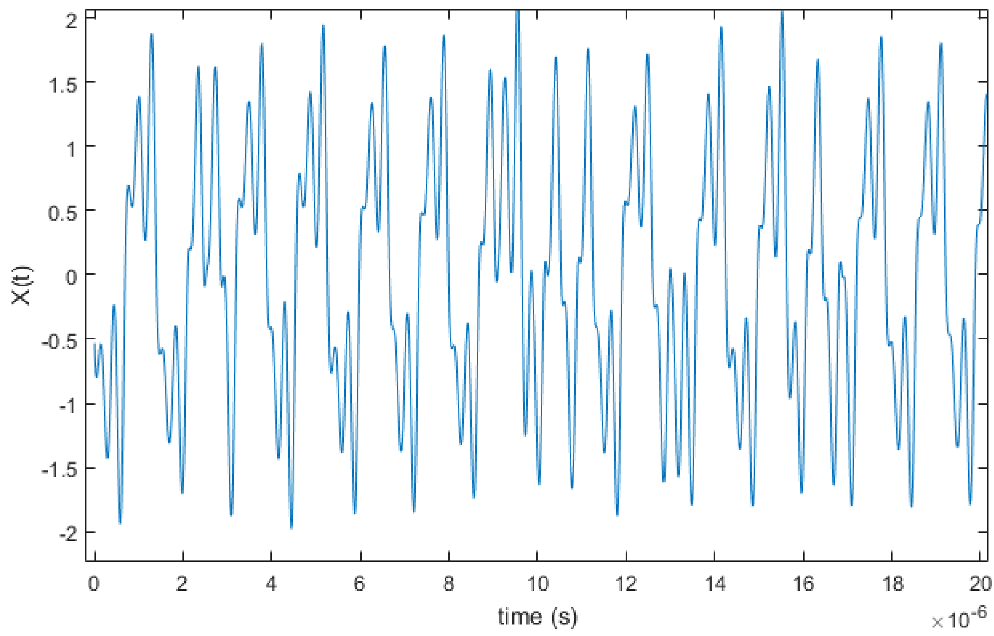

Figure 4.8: Time domain plot of $x(t)$ for the simulated system

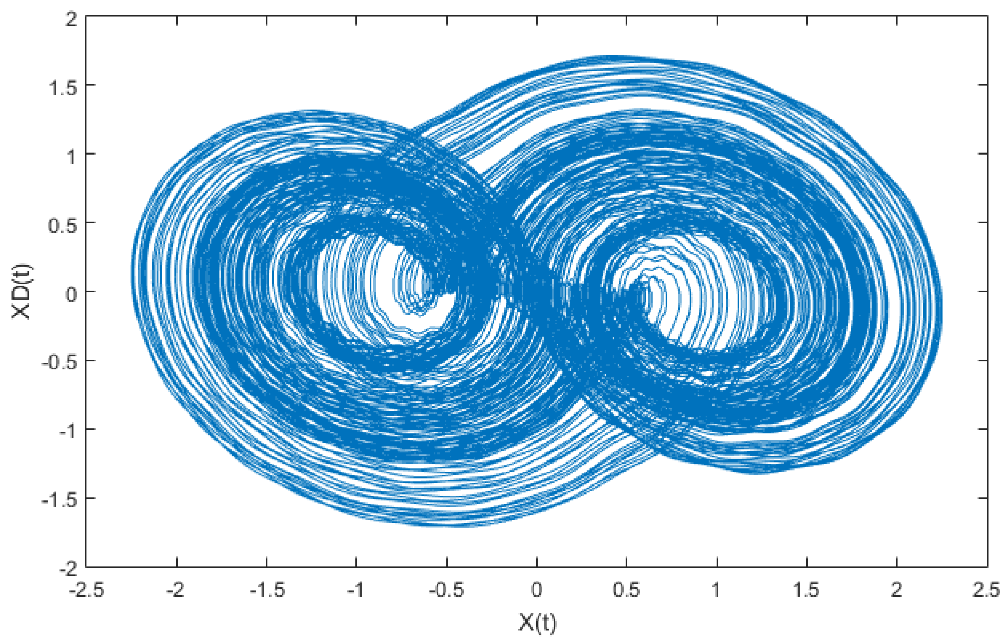

Figure 4.9: Phase space ($\dot{x}$ vs x) for the simulated system

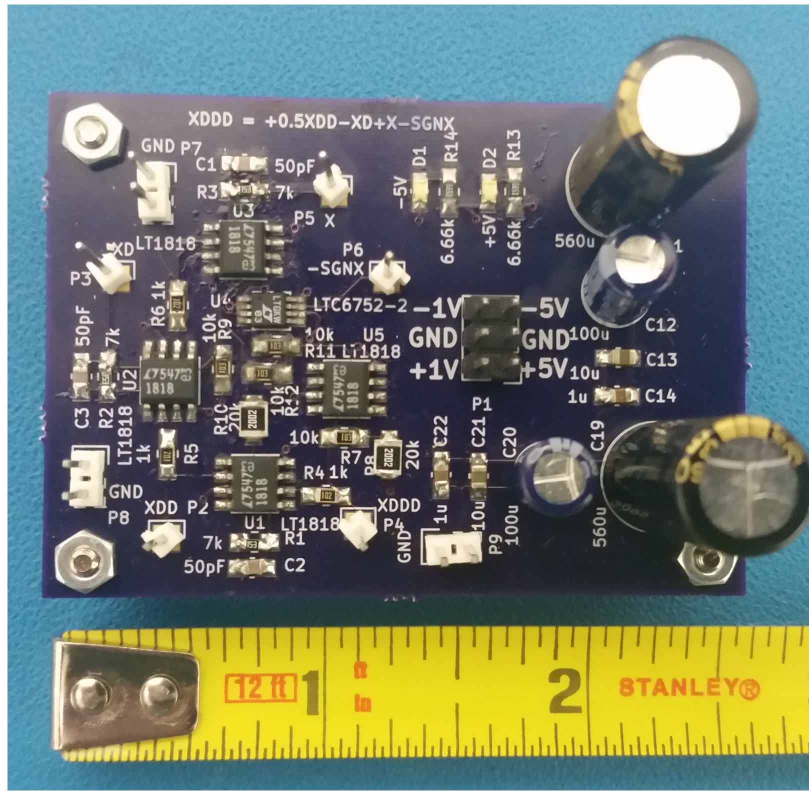

Figure 4.10: Populated printed circuit board

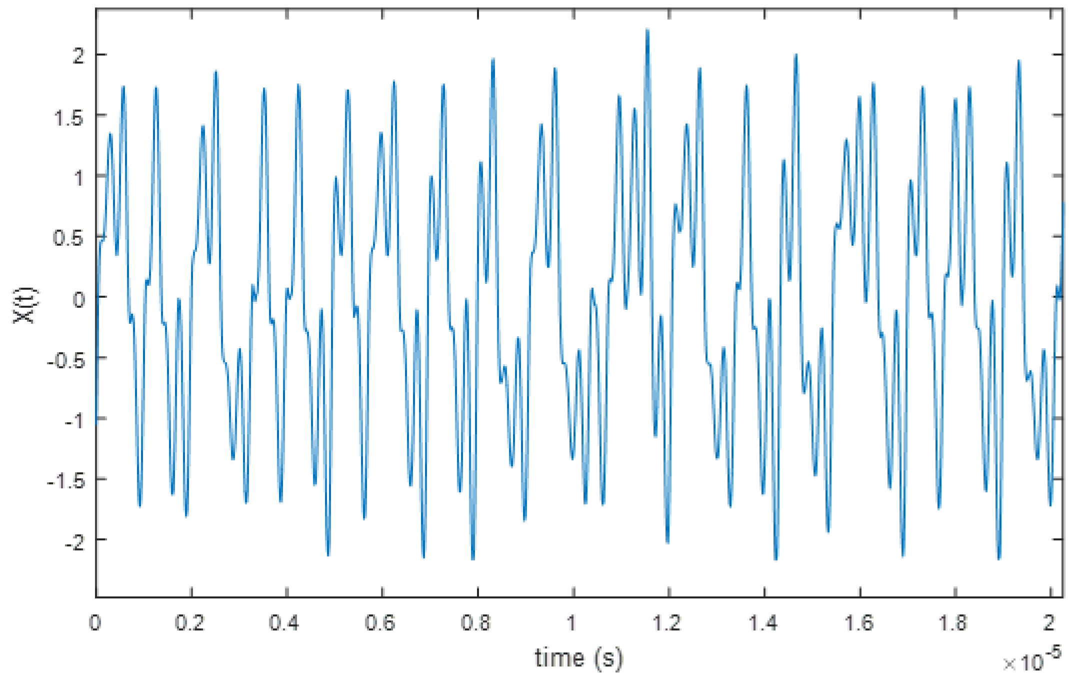

Figure 4.11: Time domain plot of $x(t)$

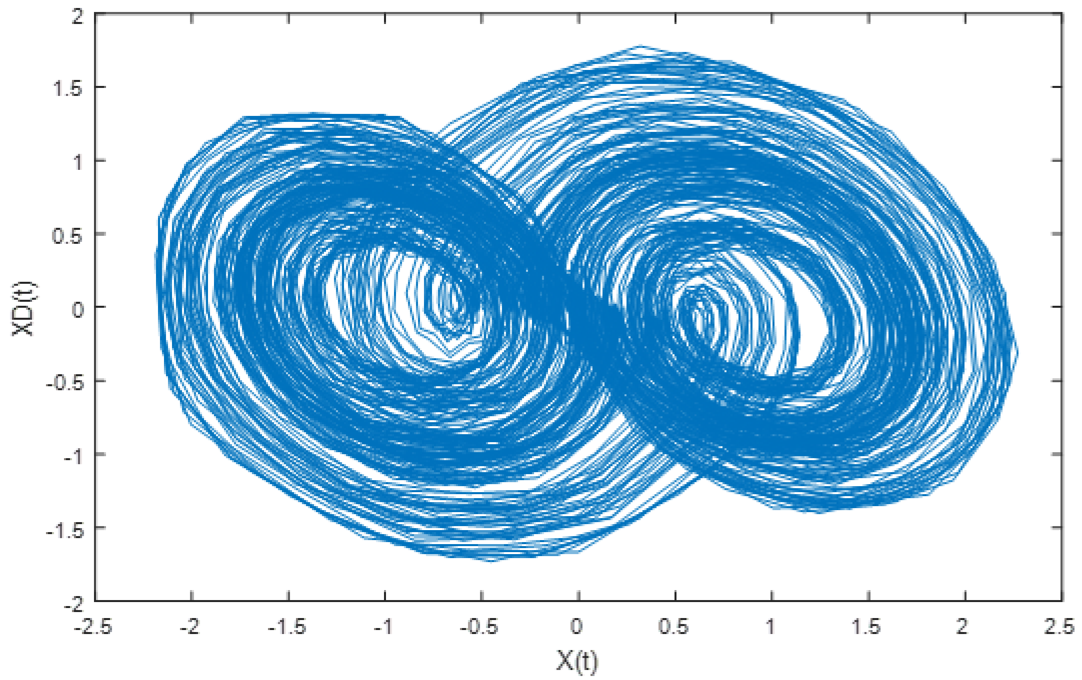

Figure 4.12: Phase space ($\dot{x}$ vs x)

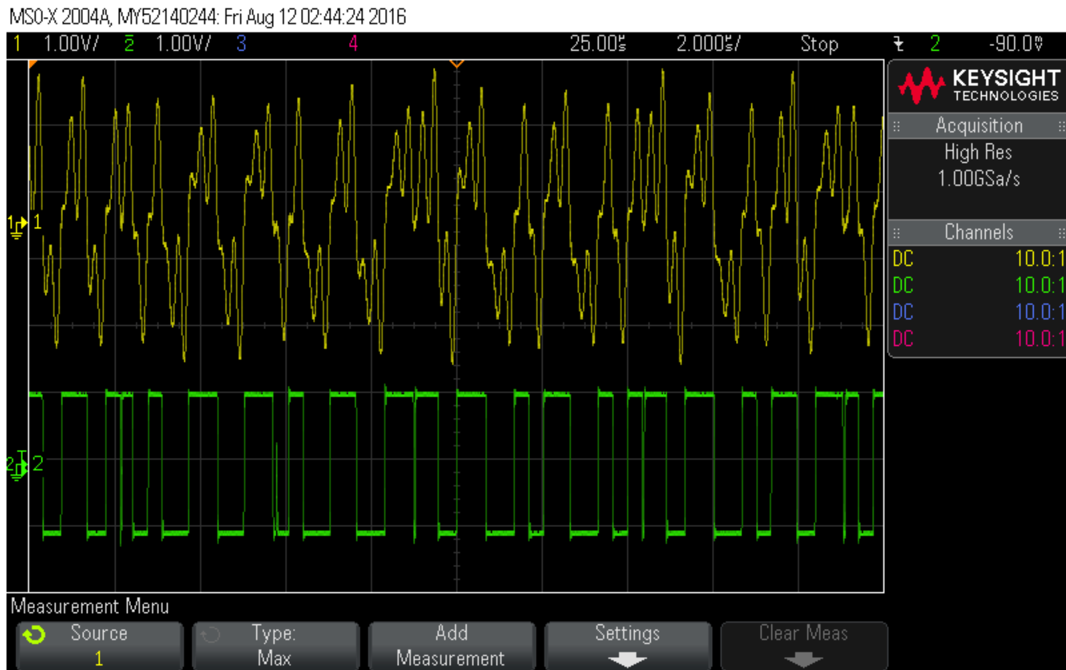

Figure 4.13: $x(t)$ and $sgn(x(t))$

4.3 Compact and Low Power Implementation

The circuit board previously described was improved in order to reduce the form factor and to decrease the required power consumption. The system requires both a positive and a negative supply voltage, which would require buck-boost converters in low power/embedded systems to provide these additional rails. This design utilized the LTC6752-2 comparator that required separate 1V and -1V rails; the coefficients of (3.2) are not kept in proportion when the +5V and -5V rails lower from their nominal value. This could result from either an increase in current requirements elsewhere or from battery voltage dropping during discharge. While the circuit may still chaotically oscillate, it would no longer be an accurate representation of the design equation.

4.3.1 Efficient Design of Chaotic Oscillators

As electronics and manufacturing technology improves, a premium is placed on cost, component count, and the physical dimensions of the overall system. Electronic subsystems should occupy the minimum area possible while still achieving high functionality. This includes consolidation of component values, using smaller size components, component layout optimization, and careful design of feedback paths for signal integrity. Chaotic circuits have the advantage of rich dynamics in a relatively small electronic implementation when compared to other traditional topologies. This simplicity should be taken advantage of as much as possible. Specifically, chaotic circuits requiring multiple operational amplifiers should aim to group these devices into a single package, and standard component values should be used instead of nonstandard values if the design requirements of the system can still be met. Also, component values should be reused throughout the design to reduce both component and manufacturing cost. However, the precision and functionality of the chaotic oscillator should not be sacrificed for the sake of cost. If the system design dictates that a specific component is necessary, it should not be replaced with another component that would impact the performance of the system negatively.

4.4 Smaller Board Design

The previous oscillator used discrete operational amplifier (op amp) chips in order to implement the three integrators and summing amplifier. These four chips have been replaced with two LT1819 dual op amp chips in micro small outline packages (MSOP). These are functionally the same as the LT1818, but the reduced size allows for a much smaller trace length in the feedback path and fewer power filtering capacitors. The feedback path of the circuit is routed in a counterclockwise direction in order to take advantage of the pinout of these smaller chips.

In addition, the LTC6752-2 comparator was swapped with a LT1713 rail-to-rail output comparator. This design choice has a number of improvements over the previous comparator. First, the LT1713 does not require any extra power supplies, so the +1V and -1V supplies can be eliminated. Furthermore, since the comparator output is rail-to-rail, the signum function will scale with the rest of the system outputs if the supply voltage changes. However, since the comparator output is not at the correct amplitude (nominally 5V instead of 1V required by the signum function), it needs to be scaled down. One potential solution is to alter the summing amplifier ratios such that the coefficient of the signum function was one fifth. This solution is not suitable for this application due to the LT1819 not having rail to rail inputs; the op amps became overloaded when the +5V signum function was added to the summing amplifier directly.

The proposed solution is to attenuate the signum function before it reaches the summing amplifier. This is achieved by using a resistor divider at the comparator output; a schematic is shown in Fig. 4.14. By having a passive resistor divider, the scaling of the output only depends on the ratio of the two resistors if care is taken to make the output path high impedance with respect to the resistors. In simulation, a ratio of $7k\Omega$ to $2k\Omega$ produced the desired 1V amplitude signum function from the comparator output.

One notable drawback of using this method is the bleedthrough of the other signals into the signum function through the summing amplifier resistors, as seen in Fig. 4.15. This is not desirable; a clean squarelike wave should be the actual signal that the signum function

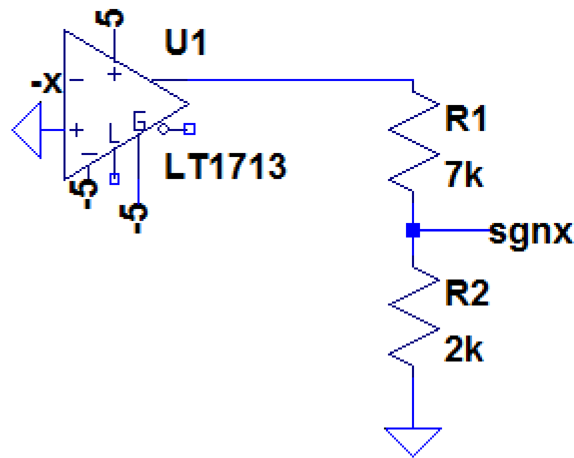

Figure 4.14: Schematic of the resistor divider scaling the signum function

produces. However, simulation results show that neither the continuous time signal $x(t)$ nor the phase space ($\dot{x}(t)$ vs. $x(t)$) is noticeably altered, seen in Fig. 4.16.

A potential solution to this issue would be to use an active attenuator, such as an op amp in an inverting amplifier configuration. This would require both another op amp and extra passive components around it to implement this solution, so it was not pursued.

These improvements are reflected in an updated system schematic shown in Fig. 4.17. A string of integrators forms a feedback loop with a summing amplifier which adds the integrator outputs in proportion to the coefficients of (3.2). The LT1713 comparator is used to generate the signum output, which is then attentuated before being added with the rest of the system outputs. The frequency of oscillation is set by the op amp integrator gains ($G = 1/RC$) which are very high (20Meg) in this implementation. The op amp's finite gain bandwidth product prevents the frequency from being increased indefinitely, so the system frequency is set at the upper limit of 4 MHz.

4.4.1 PCB Implementation

A 4-layer circuit board was designed in KiCad. In order to reduce the physical size of the design, 0603 packaged components were used. This could be further improved by using 0402 or smaller components, but this prototype board was hand assembled, so the size of these components was kept reasonable. Routing of the feedback paths was done on the top

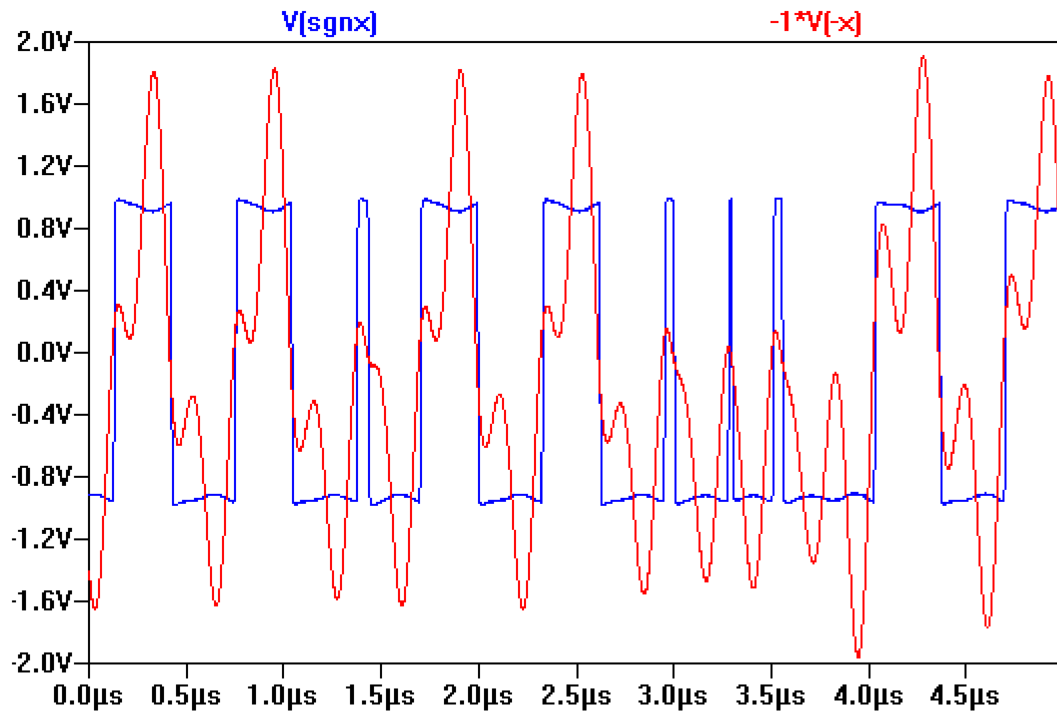

Figure 4.15: Resulting signum output showing bleedthrough

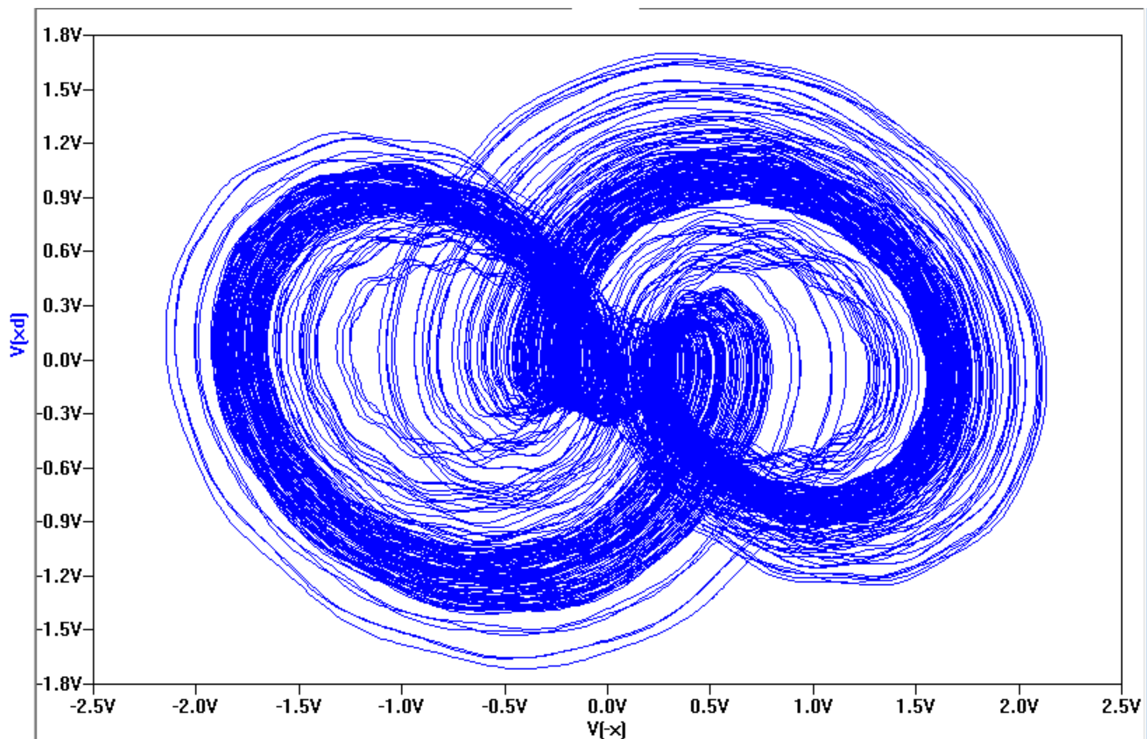

Figure 4.16: Simulated phase space $\dot{x}(t)$ vs. $x(t)$

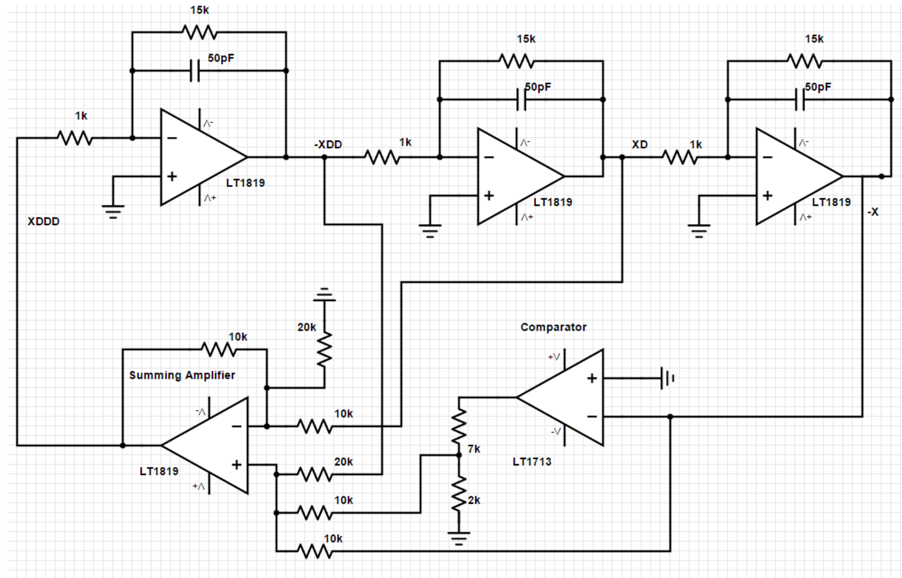

Figure 4.17: System schematic of the improved oscillator

and bottom layers of the four layer board, with power and ground planes routed on the the inner layers. This was done to have signal lines easily accesible for troubleshooting, as well as additional distributed capacitance between the power and ground planes. Moreover, having the power and ground planes in the middle prevents crosstalk between signals on opposite sides of the board. A photograph of the assembled board is shown in Fig. 4.18.

The chaotic circuit outputs are shown in Fig. 4.19 and 4.20. These are very similar to the previous circuit's outputs ($\dot{x}(t)$ and $x(t)$) in [62]. The proposed improved design still maintains the fundamental frequency of 4 MHz, but in a much smaller footprint. The final size of the circuit board is 1.5 cm x 1.5 cm (0.6 in x 0.6 in), and the current draw is 40 mA from both +5V and -5V supplies.

One issue with this new design is that the circuit is much more sensitive to the transient effect of power supply voltage start up. Turning on a power supply usually causes the integrating op amps to rail, and the system is then unable to recover and fall into the chaotic regime. Adding more bypass capacitance to the supply at the board connections mitigated this problem somewhat, but the problem still persisted on the majority of turn ons. By manually controlling a PMOS power transistor placed on the positive power rail, chaotic operation is achieved every

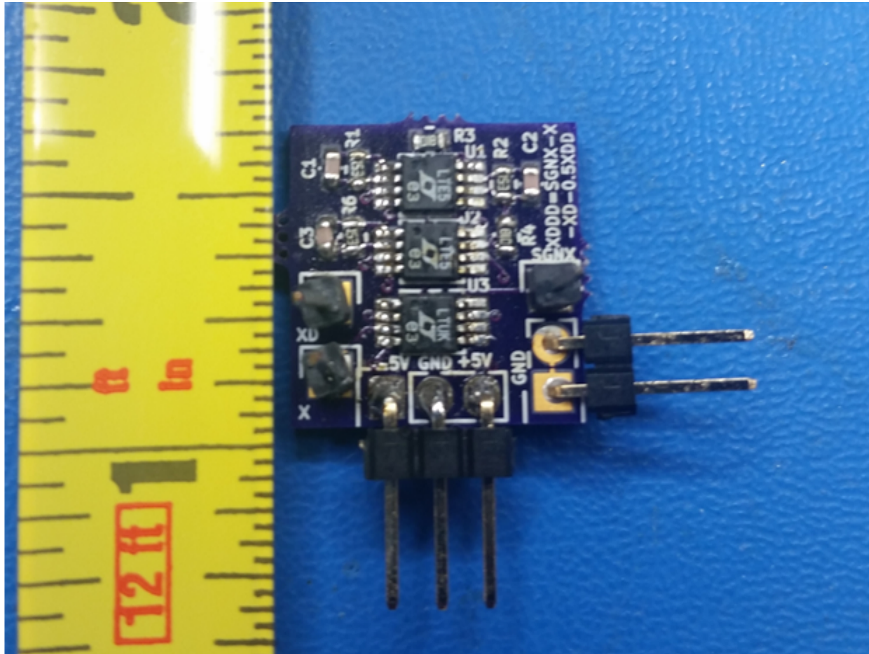

Figure 4.18: Photograph of the assembled board

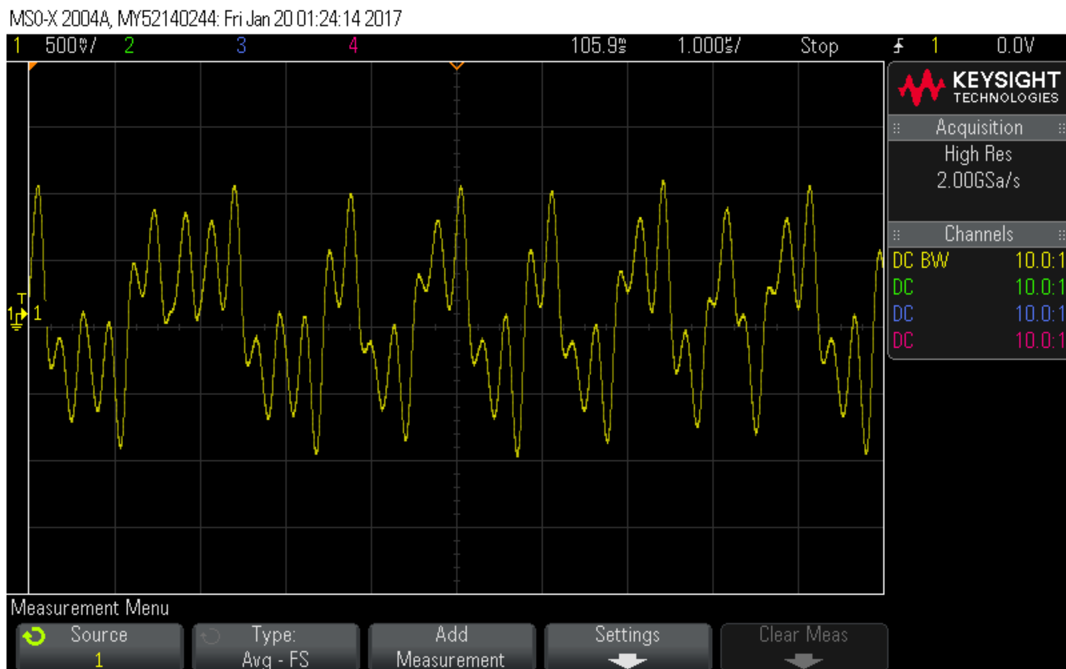

Figure 4.19: Captured continuous output $x(t)$

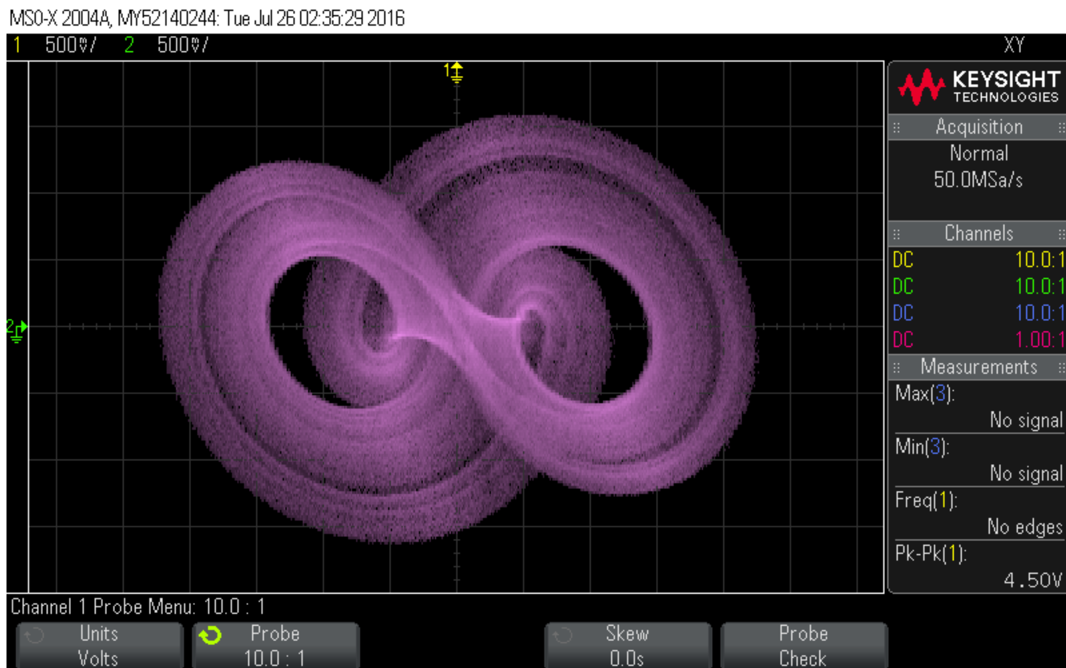

Figure 4.20: Captured phase space of the assembled circuit board

time the system power is enabled by the transistor. Allowing the power supply voltage to stabilize and then enabling the power to the board ensures that the initial conditions of the system remain in the chaotic regime.

4.4.2 Low Power Improvements

In order for the circuit to be easily integrated into other systems, the -5V supply needs to be eliminated as well. Fortunately, all of the chips on the board can work down to lower voltages than the designed -5V to 5V. By reducing each power rail to magnitude of ±2.5V, and then moving all of the circuit voltages up by 2.5V such that the center of the oscillation is around 2.5VDC instead of 0V, the circuit could conceivably be powered in a nominally 5V system (e.g. a computer peripheral). However, in order to do this, the center 2.5VDC that each of the integrators and comparator uses for a reference must be generated and not provided by an external 2.5VDC source.

A resistor divider in a 1:1 ratio will divide the supply voltage in approximately half. By placing this resistor divider from the positive supply to the negative supply and then connecting the "ground" of the board to the resistor divider output, the entire board is now referenced to

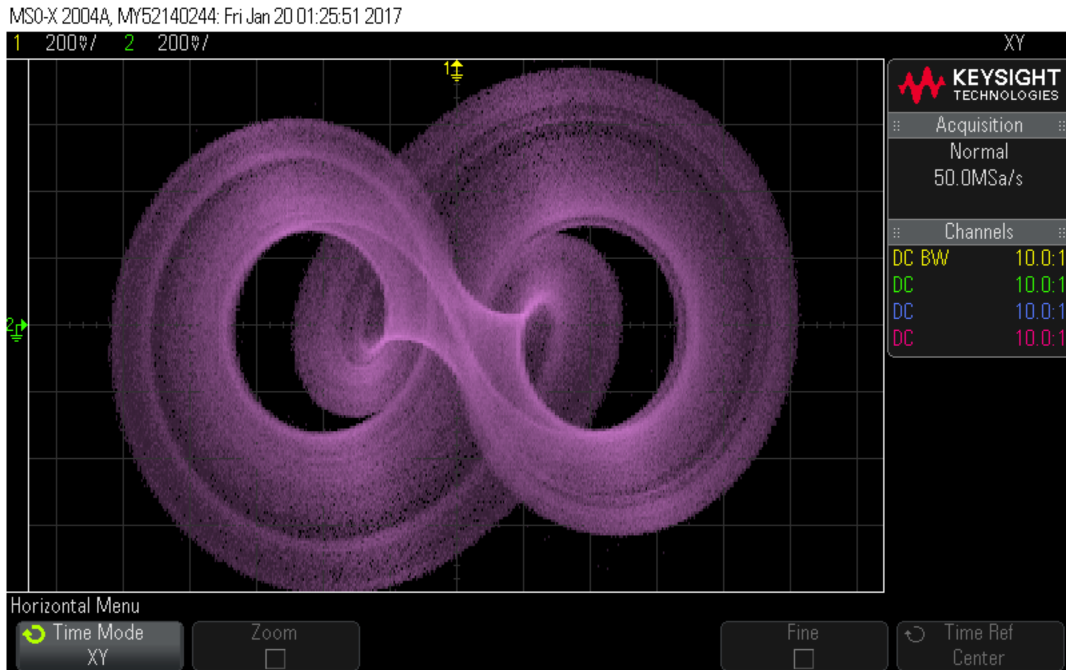

Figure 4.21: Captured phase space of the assembled circuit board operating at 2.5 VDC

2.5VDC. This divider was constructed using $1k\Omega$ resistors. In reality, the actual voltage at the resistor divider output is slightly less than 2.5VDC because there is a current path through the other resistor divider used for the signum function; however, this does not noticeably alter the system's outputs. Care should always be taken such that any load that a resistor divider is connected to is high impedance with respect to the resistor divider values; otherwise, the input impedance of the load needs to be taken into account.

The supply voltage of the final board arrangement was lowered until the circuit no longer stayed chaotic; the lowest voltage of chaotic operation was approximately 2.5 VDC ($\pm$1.25VDC for split operation) at a supply current of 20 mA, for a continuous power draw of 50mW. A capture of the circuit operating at this low voltage is shown in 4.21.

The improved low power circuit had additional components included to improve ease of use of the circuit board and so that testing more easily could be performed. Specifically, the pin headers used to power the board were exchanged with a micro USB connector to enable power to come directly from a number of readily available power sources, including a host computer with an open USB port or a mobile battery bank for testing inside of an enclosed space. A PMOS transistor was placed after this connector in order to apply power to the rest

of the circuit in a more controlled fashion (via unshorting a jumper on the board from ground) than just plugging in the connector. Also, in testing previous revisions of the board, it was discovered that the circuit could enter a periodic railing state upon being powered up, but could then be forced into the normal chaotic state by temporarily shorting the x node of the op amp integrators to ground. A small momentary switch was added to this node to facilitate this correction. Finally, the signum output was taken from the $-Q$ pin on the comparator so as to not interfere with the Q signal in the feedback path. The fundamental frequency of the board was maintained at 4 MHz. A photograph of the front and back of the updated board are shown in Fig. 4.22.

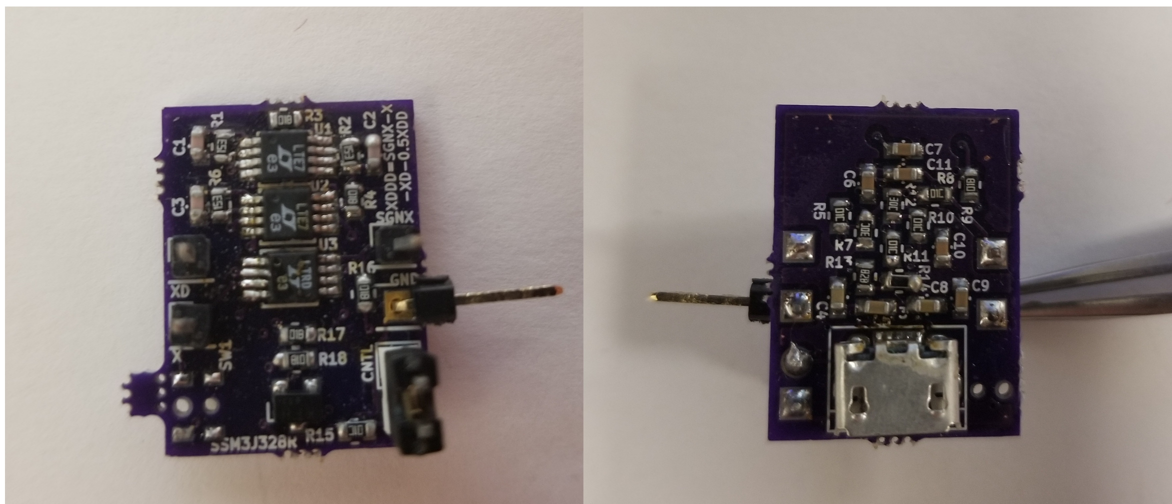

Figure 4.22: The front and back of the populated circuit board which implements the jerk equation at 4 MHz.

Chapter 5

Hardware Random Number Generation Results

In order to obtain random bits from the system, the system is sampled at a fixed rate using an analog-to-digital converter (ADC). There are a number of different parameters for an ADC that can be chosen in regards to sampling, including voltage range, sampling frequency, and bits of resolution. In order for the jerk system to be a true random number generator, the sampling parameters of an ADC sampling the system which produces bits that are statistically random must be determined. These parameters will then be replicated in hardware and will sample a physical electronic circuit in order to get truly random bits.

Simulation of an ADC sampling the system was achieved in MATLAB by implementing the jerk system equation with a 0.01s time step. The 32-bit floating point value for x (since the x variable will be sampled in hardware) is then converted to an n-bit sample value based on the chosen resolution of n bits and the ADC's maximum voltage. Successive samples are taken at approximately the fundamental frequency of the jerk system (i.e. at $0.2Hz$). For each sample, every bit of the sample except the lowest bit is discarded, and the lowest remaining bits are concatenated to form 8-bit random bytes.

The Dieharder test suite was used to evaluate the bit sequences generated from this simulation [13]. A tutorial on compiling Dieharder for Windows is given in Appendix A. There are 114 tests of randomness in this suite, but some tests are small variations on other tests. However, in evaluating the sequences for randomness, the 114 tests are viewed as separate. Each test returns a P-value between 0 and 1, which is interpreted as follows: a P-value that is between 0.005 and 0.995 is considered to have passed that test, and a P-value of exactly 0 or 1 is considered to have failed. P-values that are under 0.005 and above 0.995 are "weak" and can be futher resolved to either pass or fail through more testing. Due to the amount of data

that Dieharder requires, some sequences that are actually random will produce weak P-values in around 1% of tests. An example output of Dieharder is shown in Fig. 5.1.

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
 file_input_raw|           hard_3p125MHz_B16.bin|  1.92e+07  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.97980759|  PASSED
      diehard_operm5|   0|   1000000|     100|0.65329807|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.84539595|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.16086332|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.52860159|  PASSED
        diehard_opso|   0|   2097152|     100|0.53328270|  PASSED
        diehard_oqso|   0|   2097152|     100|0.96756200|  PASSED
         diehard_dna|   0|   2097152|     100|0.13417533|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.96201967|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.28801364|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.75190252|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.88077548|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.71120484|  PASSED
     diehard_squeeze|   0|    100000|     100|0.16043836|  PASSED
        diehard_sums|   0|       100|     100|0.04247707|  PASSED
        diehard_runs|   0|    100000|     100|0.10396408|  PASSED
        diehard_runs|   0|    100000|     100|0.40512514|  PASSED
```

Figure 5.1: Partial output of Dieharder testing. More tests and results are given in Appendix D than are shown here.

The P-values returned by the Dieharder tests are such that if the tested bit sequence is statistically random, the P-values should be uniformly distributed from 0 to 1. This allows for both individual test results and the results from the Dieharder suite as a whole to be analyzed quickly. This can be accomplished for all of the results visually by plotting them versus a uniform distribution to see how well they agree. The P-values are sorted in ascending order (the type of test for each P-value is not taken into account) and then plotted against a straight line (the uniform distribution). The resulting plots from this process are shown in Fig. 5.2 for different parameters of the simulation. The bit taken from each sample is swept from the 6th bit to the 12th bit, and shows a generally increasing resemblance to the uniform distribution as the lower bits are taken. The time between samples is then swept from 1 bit per cycle (i.e. $0.2Hz$) to 5 bits per cycle ($1Hz$). Although this is only simulation data and thus by definition not truly random, it provides a baseline from which to build a hardware system that closely matches the parameters from the simulation equation and the analog-to-digital converter. As shown in Fig. 5.2, bit sequences gathered from ADC resolutions below 10-bits do not pass the Dieharder suite

as there are a multitude of failures ($P \approx 0$) of statistical tests (noted by the line staying close to 0 instead of increasing with the uniform distribution). This indicates that these sequences are not statistically random. Above the 10-bit resolution mark, these plots show that the P-values from the generated bit sequences are close to the desired uniform distributions since the lines show closeness to the uniform distribution.

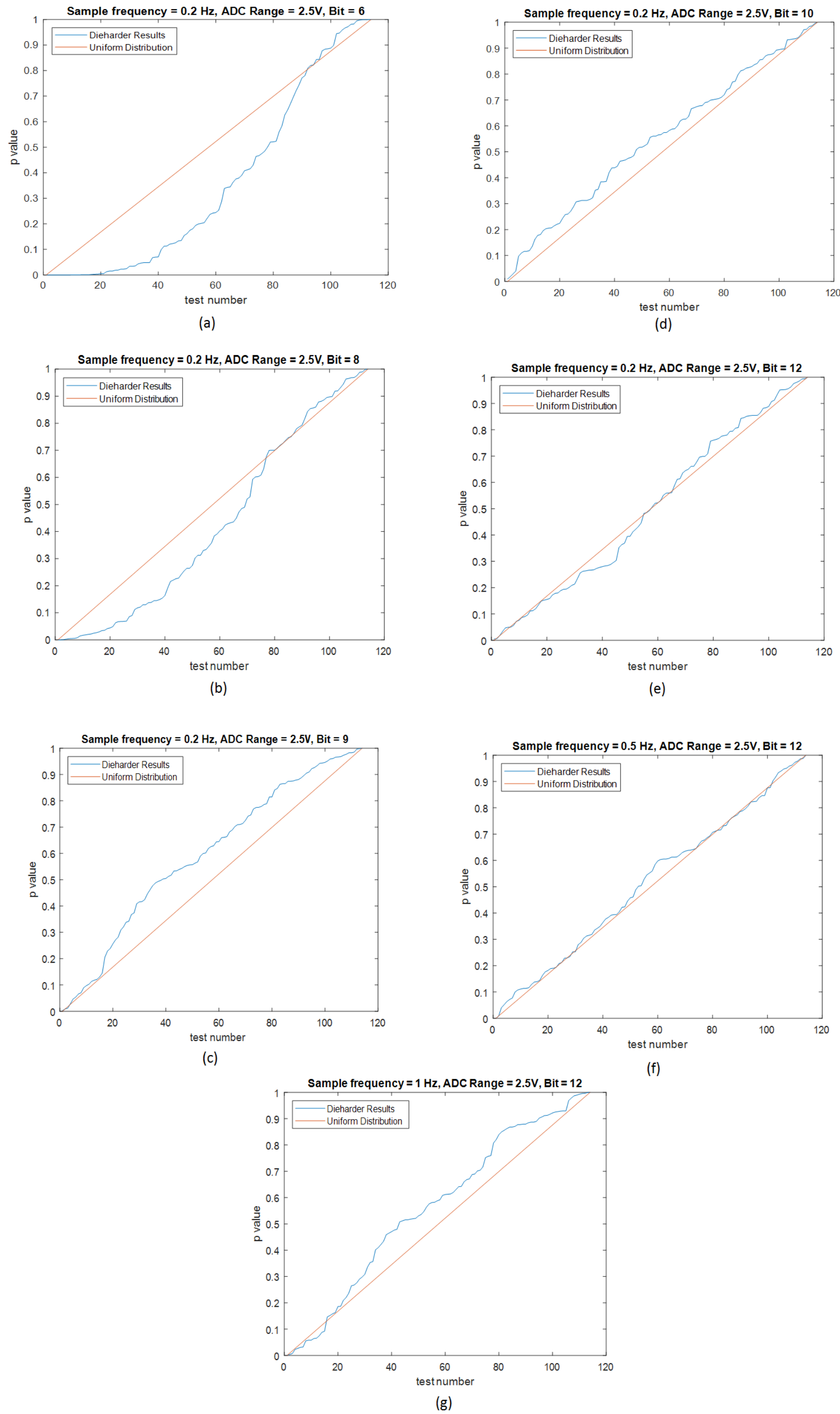

Figure 5.2: (a) 6th bit of an ADC sample of the system at 0.2 Hz with a full scale range of 2.5V. (b) 8th bit of an ADC sample of the system at 0.2 Hz with a full scale range of 2.5V. (c) 9th bit of an ADC sample of the system at 0.2 Hz with a full scale range of 2.5V. (d) 10th bit of an ADC sample of the system at 0.2 Hz with a full scale range of 2.5V. (e) 12th bit of an ADC sample of the system at 0.2 Hz with a full scale range of 2.5V. (f) 12th bit of an ADC sample of the system at 0.5 Hz with a full scale range of 2.5V. (g) 12th bit of an ADC sample of the system at 1 Hz with a full scale range of 2.5V.

The previous test results were performed by sampling at 1 bit per cycle, and almost ideal results are obtained at the 12-bit resolution level. The last two plots in Fig. 5.2 show test results when the sampling frequency is increased to 2.5 bits and 5 bits per cycle at 12-bits of resolution, respectively. These plots indicate that statistically random bits can be obtained from a system that is sampled faster than the maximum Lyapunov exponent dictates for true random number generation. Thus, it is imperative that the implementation of the random number generation take into account the theoretical limitation for true randomness, else the system as a whole cannot be truly random, even though statistical randomness might be achieved at a higher bit rate.

For testing of this circuit as a true random number generator, ADC sampling was achieved using a Handyscope HS6 USB Oscilloscope from TiePie Engineering connected to a host computer running the provided MultiChannel software. Other USB oscilloscopes tested did not support the high speed continous streaming of data required beyond 8 bits. The circuit was powered from a USB port on the same computer. The HS6 allows for up to 16 bit streaming ADC sampling, but at that resolution the sampling speed is limited to 3.25 MS/s, slightly less than the desired 4 MS/s to achieve a 1 bit per cycle random output. The full scale voltage of the ADC was chosen to be $\pm$2V since the circuit has peaks of approximately $\pm$1V when AC coupled to the oscilloscope and powered using the single 5V supply from the USB port. This results in a loss of 1 bit of resolution when compared to a full scale voltage that is smaller, but the next lowest supported by the MultiChannel software is 800mV. A screenshot of the circuit being sampled in this software is shown in Fig. 5.3.

32 GB of data was collected from the x node of the circuit at 3.25 MS/s and then split using a Matlab script into sixteen 2 GB files, one with each separate bit of the 16-bit samples concatenated together. These files were then subjected to Dieharder testing in the same manner as the jerk equation simulation data. The results from the testing are summarized as before in Fig. 5.4.

For the hardware circuit, the bit sequences pass most of the Dieharder tests starting at the 13th bit and do not fail any tests at the 16th bit of the ADC sample. At the 12th bit and above, the sequences systematically fail certain sets of tests, most notably the sts_serial and

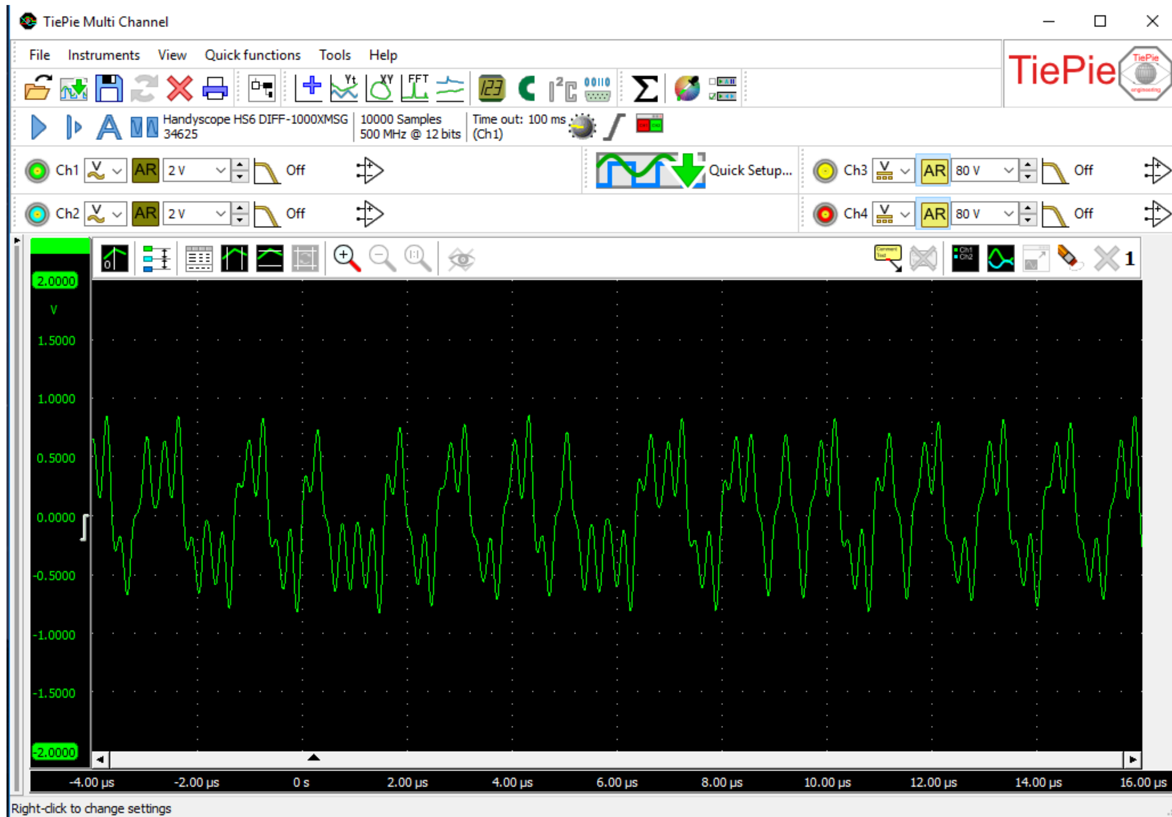

Figure 5.3: The hardware circuit board sampled by the Handyscope HS6 inside of MultiChannel.

rgb_lagged_sum series of tests. These tests involve skipping many bits in a row and thus the input file is rewound multiple times for each of these tests. This can potentially make the bit sequence seem like it is repeating itself, but this is unavoidable with this setup of Dieharder without a much larger input file size.

Although there is no real way to truly prove randomness, the results from the Dieharder RNG test suite indicate that the jerk circuit is able to provide statistically random bits from a hardware source under various circumstances. This test suite is widely used to stringently test pseudorandom number generators with much larger bit sequences available on demand. Since the chaotic jerk circuit is implemented in a small form factor with commercial off the shelf components, it can easily be integrated into other systems requiring truly random bits at high speeds. The results from the simulation RNG testing and hardware RNG testing are in agreement with each other, so overall the jerk system and circuit are ideal candidates for random number generation.

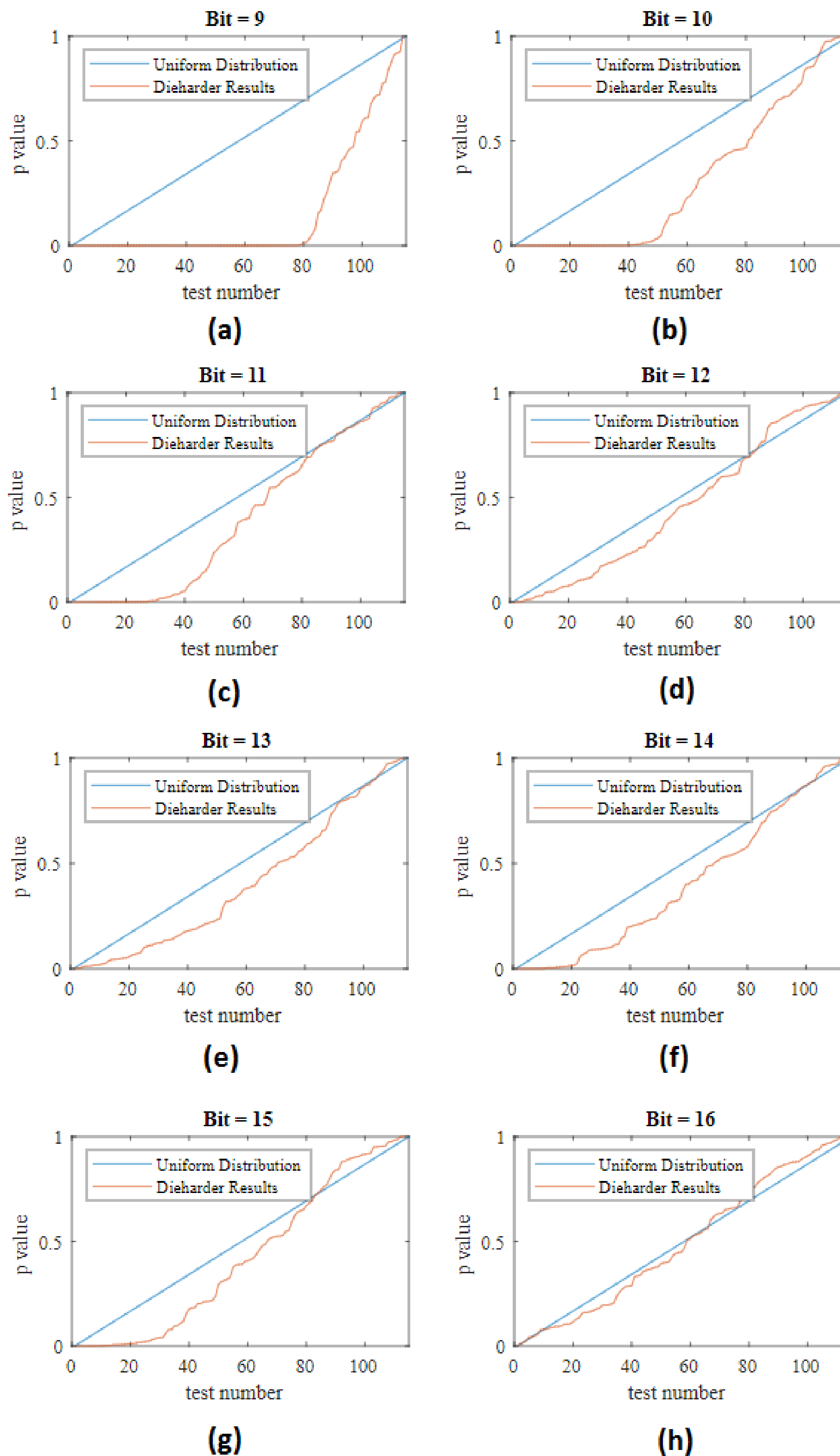

Figure 5.4: Results from Dieharder testing of the hardware circuit. All bits were collected at 3.25 MHz with a full scale voltage of ±2V. (a) 9th bit of each 16 bit sample (b) 10th bit of each 16 bit sample (c) 11th bit of each 16 bit sample (d) 12th bit of each 16 bit sample (e) 13th bit of each 16 bit sample (f) 14th bit of each 16 bit sample (g) 15th bit of each 16 bit sample (h) 16th bit of each 16 bit sample

Chapter 6

Other Methods Of Obtaining Randomness From Chaos

So far, this work has investigated one method of extracting random bits from a chaotic system and circuit. There are a plethora of other methods and systems of interest that are available for random number generation. This section will show some preliminary results from testing these other methods and systems, but will not go into as much detail for each one as has been previously discussed.

The method shown in this work notably had no post-processing of the bit stream performed on it; the bits were extracted directly from the chaotic waveform and then concatenated together. Often this is not the implementation used in practice; hardware random number generators are susceptible to biases in the output bit stream that must be corrected in order to pass statistical testing. This correction or "whitening" stage is usually in the form of a post-processing step before the bit stream is passed onto the random number sink. Note that many types of biases can exists within a bit sequence which will cause the sequence to fail some of the statistical tests within Dieharder. These biases include bit-wise biases (e.g. more ones than zeros), multibit-wise biases (e.g. uneven distribution of the 2^n combinations of n bit sequences), and dependency or correlation between bits.

6.1 Von Neumann Corrector

The most popular correction algorithm for biased bit sequences is known as the Von Neumann corrector (VN). This algorithm is designed to correct a bit-wise bias in the original sequence and return a sequence that has a fifty-fifty split between ones and zeros. VN takes bit pairs from the original sequence and then compares them. If the two bits in the pair are the same (i.e. 00 or 11) then no output bit is given. If they are different (i.e. 01 or 10), then either

a 0 or 1 is given as the output bit, depending on the implementation. For the results given here, 01 was mapped to a 1 and 10 was mapped to a 0; these were arbitrarily chosen and do not affect the results of statistical testing.

VN operates using the principle that, given two probabilities of events H and T (or 1 and 0) as $P(H)$ and $P(T)$, respectively, then $P(HT) = P(TH)$, even when $P(H) \neq P(T)$. To illustrate this, consider the resulting probabilities of unbiased and biased bit pair sequences, given below in Tables 6.1 and 6.2, respectively. An example of a bit sequence resulting from VN is shown in Table 6.3.

Table 6.1: Bit Pair Probabilities of an Unbiased Sequence

event	P(event)
H	0.5
T	0.5
HH	0.25
TT	0.25
TH	0.25
HT	0.25

Table 6.2: Bit Pair Probabilities of a Biased Sequence

event	P(event)
H	0.7
T	0.3
HH	0.49
TT	0.09
TH	0.21
HT	0.21

Table 6.3: Von Neumann Corrected Sequence

00	00	10	01	10	11	01	11	11	10	10	11
no bit	no bit	0	1	0	no bit	1	no bit	no bit	0	0	no bit

Notice that even though the individual probabilities of events H and T are unequal, their combined successive probabilities are equal. This is true for all values of the probabilities if

the two events are independent. Thus, the VN corrector is able to transform a biased sequence into an unbiased sequence based on these equal successive probabilities.

The disadvantage of VN correction lies in the amount of bits in the original bit sequence that are discarded. In the best case scenario, the resulting bit sequence is only 25% of the length originally. For example, take a bit sequence that has an even distribution of ones and zeros. In 50% of bit pairs, no output is given. In the other 50%, only one bit from the pair is taken. Generally, the length of the resulting bit sequence is $P(H)P(T)$ times the length of the original sequence, where H and T are the individual bit probabilities. Obviously, for highly biased sequences, the VN corrector reduces the overall length dramatically. This presents an issue when running tests through Dieharder since Dieharder rewinds the file sizes so often. These rewinds could cause the sequence to repeat itself and fail a test when otherwise it would pass.

To evaluate the improvement, if any, that Von Neumann correction provides for the chaotic jerk system previously discussed, the bit sequences generated in simulation from the jerk system were passed through VN and then the resulting corrected sequence were sent to Dieharder for testing. The ultimate evaluation for improvement from the original sequence to the corrected sequence remains mostly visual. The resulting lists of p-values are plotted against a uniform distribution as before; if the VN corrector has improved the randomness of the original sequence, then the line created by the p-values should lie closer to the uniform distribution. This simulated data was generated using a sampling frequency of 0.2 Hz and a full scale voltage of 2.5V. The original lengths of the bit sequences were 2 GB (i.e. 2 billion 8-bit samples). After Von Neumann correction, these file sizes shrunk by almost exactly 75% to approximately 500 MB. The resulting plots are shown in Figs. 6.1 and 6.2 for bits 1-8 and 9-16, respectively.

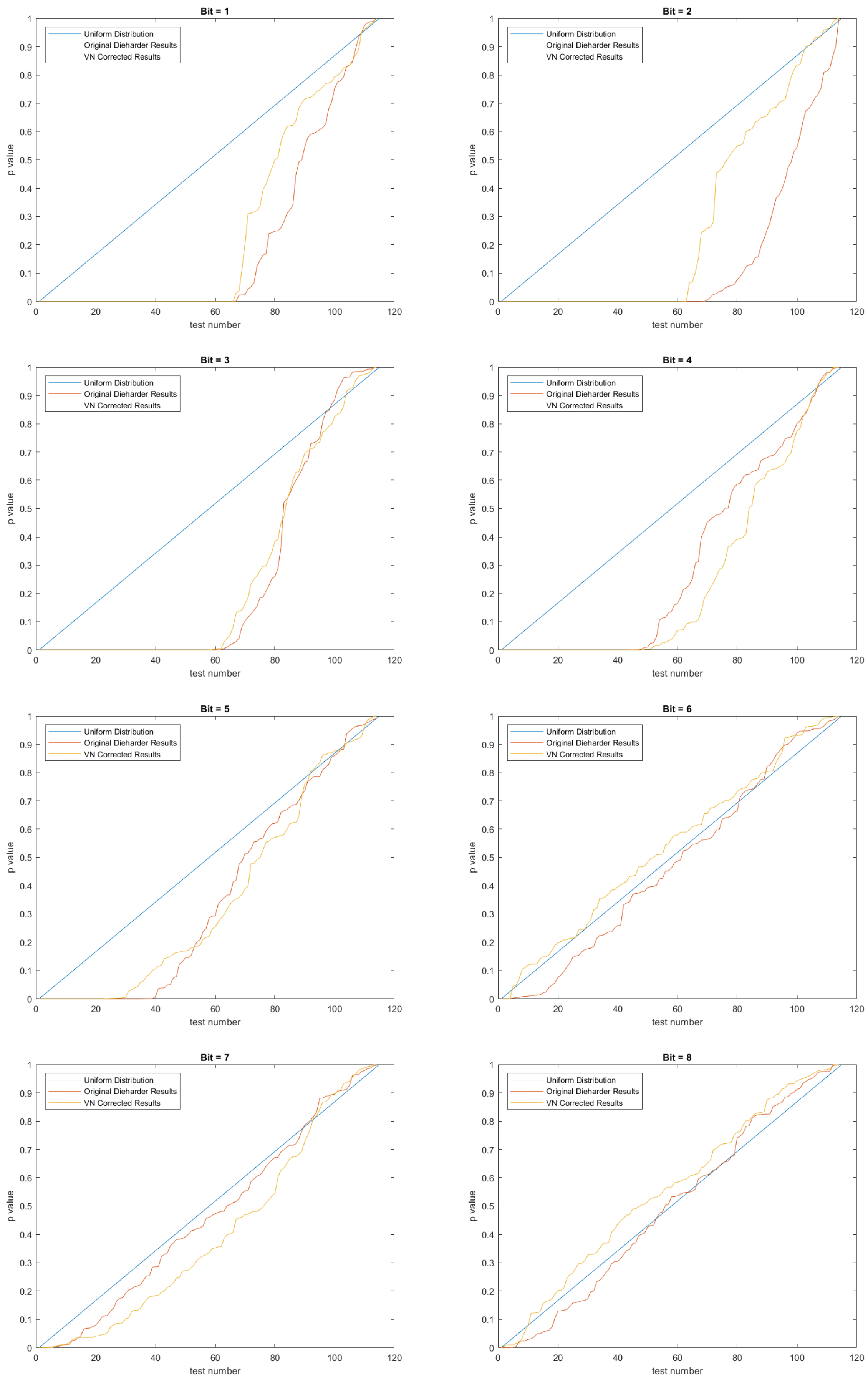

Figure 6.1: Results from simulated data with Von Neumann correction (bits 1-8)

These results do not indicate that the Von Neumann correction provides a meaningful improvement over the original bit sequences. This makes sense given that the tested bit sequences are all simulated: there are no inherent bit-wise biases in the sampling scheme in software. That is, the ADC in software is "perfect" in that it converts the high-precision x value into a lower precision sampled value without introducing any additional error. Thus, the VN corrector is unnecessarily throwing away bits that aren't actually bit-wise biased.

A real test of the VN corrector is to apply it to a bit sequence generated from hardware sources, where the biases and imperfections are inherent to the system. The bit sequences previously generated from the hardware circuit were also passed through VN and then to Dieharder for testing. These results are shown below in Fig. 6.3.

The hardware sequences benefit much more from VN correction. In general, the VN p-values are closer to the uniform distribution than the original p-values. Note that the original sequences were also 2 GB (after VN correction approximately 500MB). Despite the smaller file size, the VN sequences are able to pass more of the Dieharder tests, especially at the higher bits. The lower bits do not benefit as much from this correction since they already pass most of the test suite.

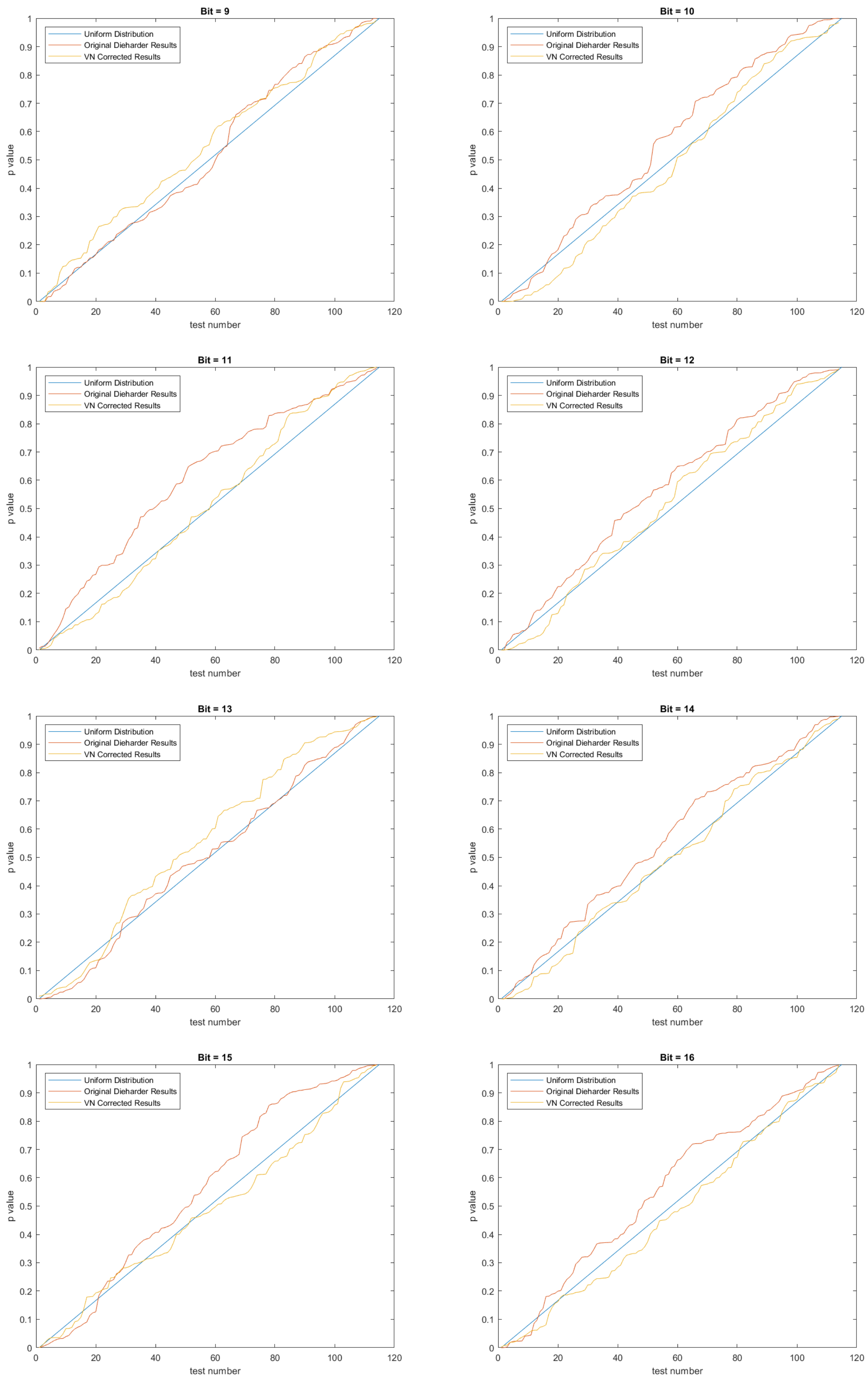

Figure 6.2: Results from simulated data with Von Neumann correction (bits 9-16)

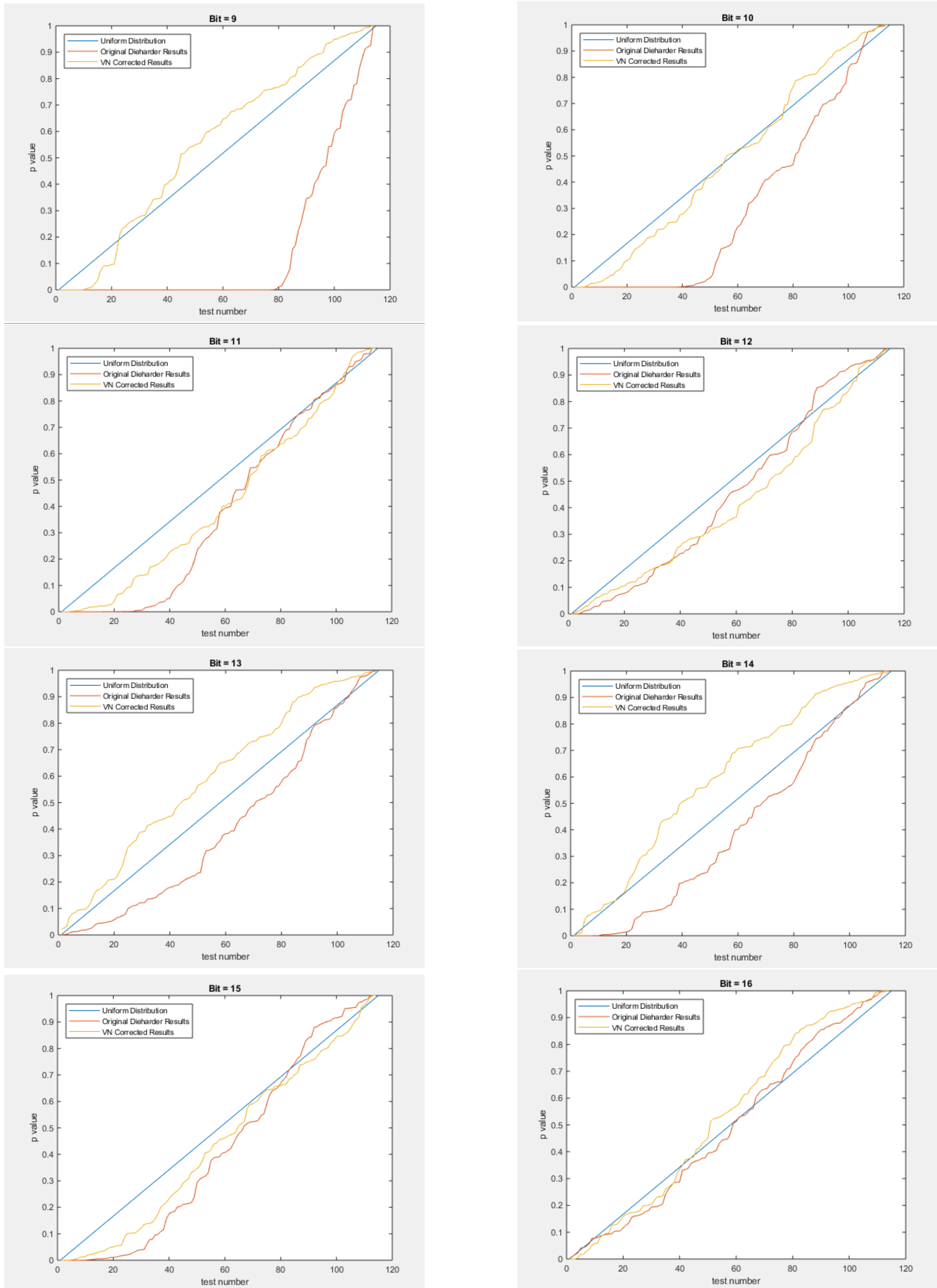

Figure 6.3: Results from hardware data with Von Neumann correction (bits 9-16)

6.2 XOR Operation

Another bit manipulation scheme commonly used is performing an exclusive-OR (XOR) operation on two simultaneous bit streams. Normally these bit streams come from two oscillators that are too biased to be used by themselves. By performing the XOR operation, it is hoped that the biases “cancel” each other out to get an overall better bit stream on the output. The XOR truth table is given below in Table 6.4.

Table 6.4: The exclusive-OR truth table

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

It is also worth noting that, for an XOR operation performed in hardware, only the first “bit” is available to be operated on. If two oscillators are placed on the inputs of, for example, a 7400 series XOR logic chip, then the two voltages on the inputs are “converted” to a high or low value for the purposes of the XOR. Usually voltages below $VDD/2$ are “0” and voltages above are “1”. However, for the purposes of testing, we will assume that all bits of a sampled oscillator are able to be accessed and the XOR operation performed.

It is important that the two oscillators being sampled are completely uncorrelated. As noted by the truth table given above, if the two bit streams are consistently giving the same output, then the XOR output will be more weighted with zeros. This correlation can happen through different mechanisms, but generally there needs to be some sort of coupling between the two oscillators for long term synchronization.

In order to evaluate the XOR operation as a potential improvement for reducing bit stream bias, two sets of binary data were generated as before for each of the 16 bits of a 16 bit sample if two oscillators were sampled in hardware. The two files for each bit were then XORed with each other to produce a final output file to then evaluate with Dieharder as described previously. These results are shown in 6.4 and 6.5 for bits 1-8 and 9-16, respectively.

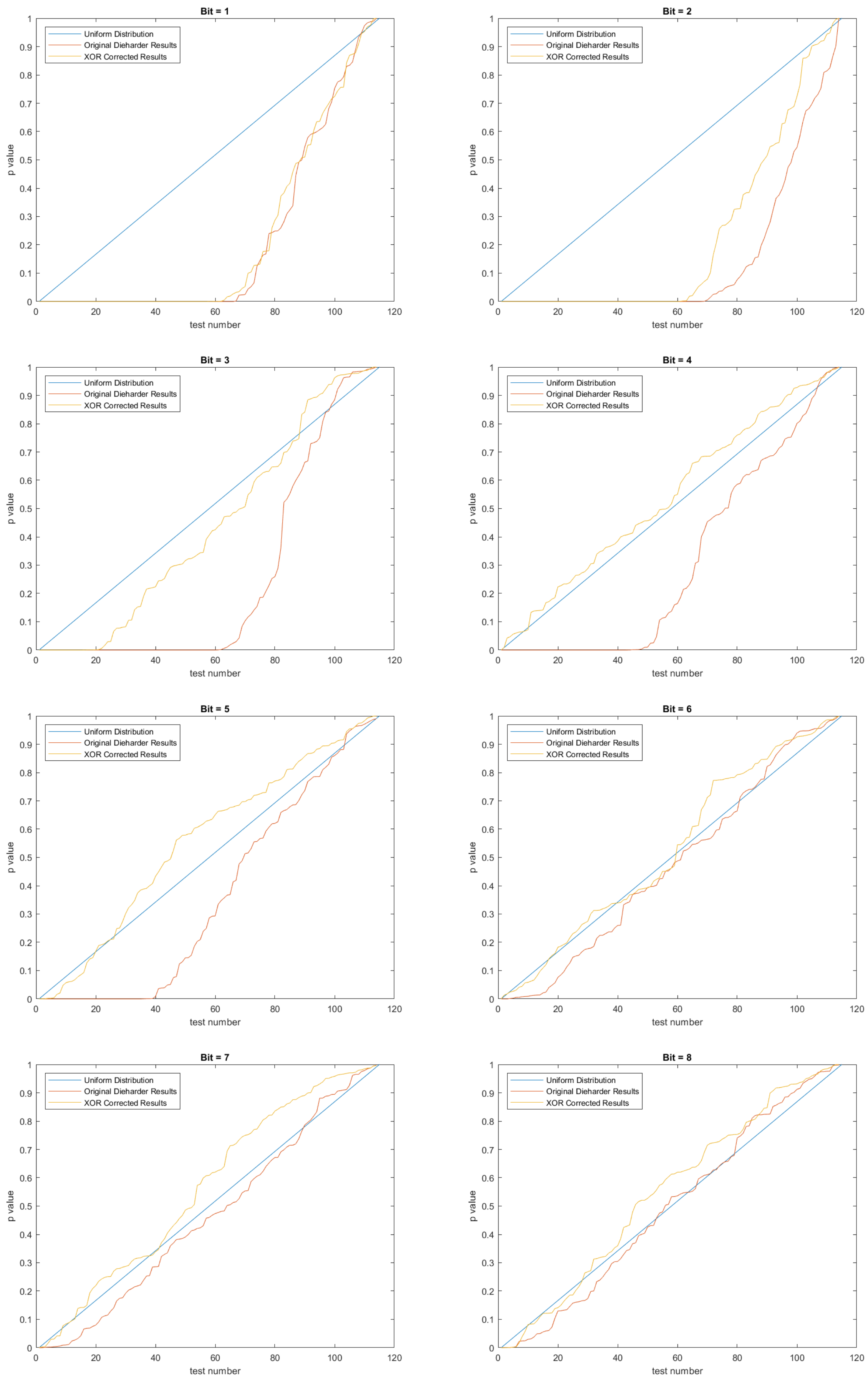

Figure 6.4: Results from simulated data with XOR correction (bits 1-8)

Based on these results, the XOR operation is very powerful in reducing bias at more significant bits (i.e. bits 1-6). After this point, the simulated data are unbiased enough for the XOR operation between the two data sets to not provide much improvement.

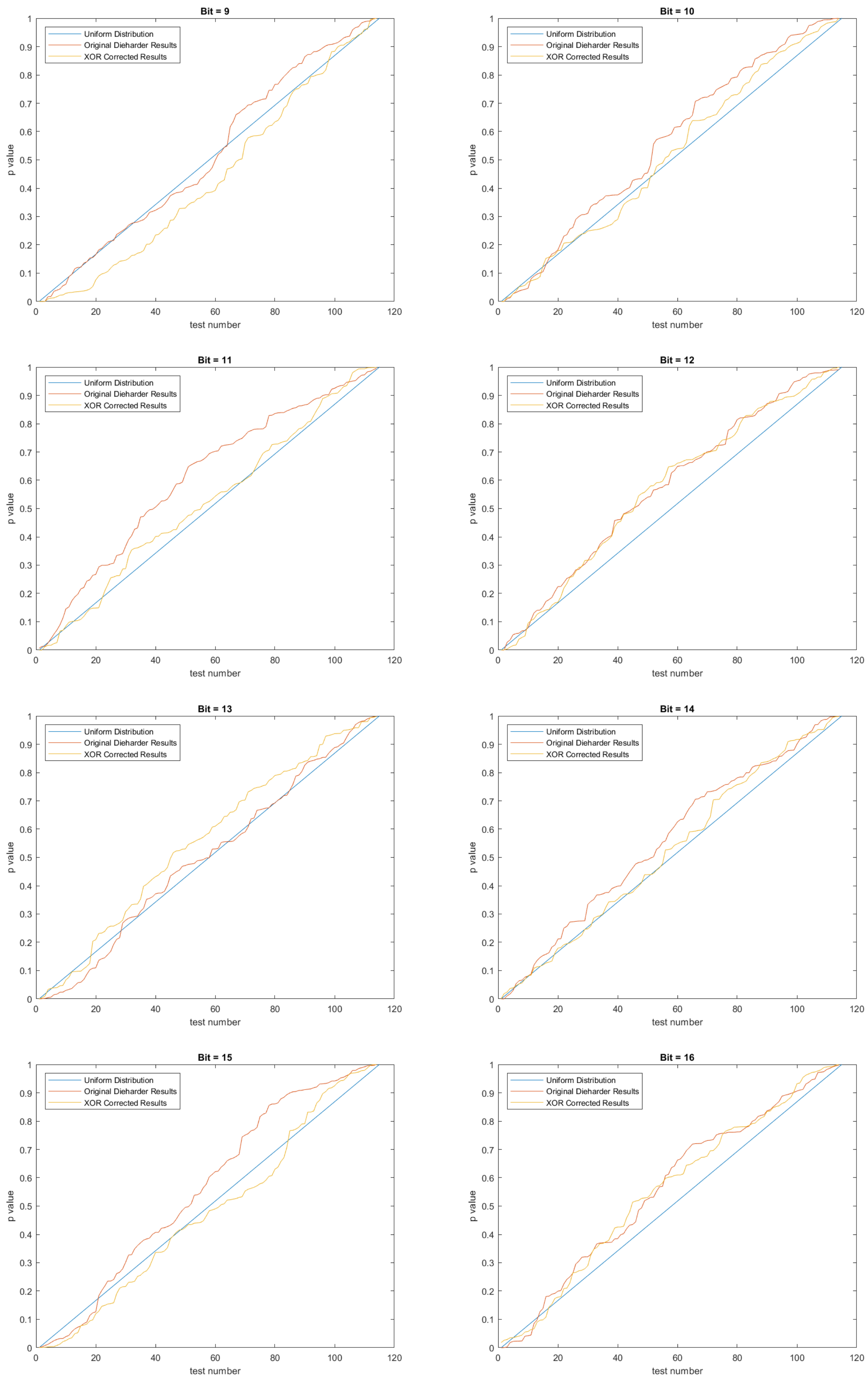

Figure 6.5: Results from simulated data with XOR correction (bits 9-16)

Chapter 7

Conclusion

In this work, a discussion of information and randomness in relation to chaos and random number generators has been presented. From examining certain properties of a chaotic system in simulation, a scheme for extracting true random numbers directly from this chaotic jerk system has been shown. This system has been implemented in high speed electronics on a small printed circuit board and sampled in accordance with the necessary parameters found from the simulation results. The bit sequences generated from both the simulation and the physical system pass the Dieharder RNG test suite, which enables this system to function as a fast random number generator for many different applications. The hardware circuit has a small footprint of 1.5 cm x 1.5 cm and operates at a fundamental frequency of 4 MHz. The 16th bit of the sampled chaotic circuit provides statistically random bits without any post processing of the resulting sequence. Also, a framework for evaluating other random number generation schemes based on chaotic systems has been given and is applicable to a wide variety of potential RNG solutions. Overall, this system shows high dynamic complexity in a compact form, which is desirable for a true random number generator.

Chapter 8

Future Work

The focus of this work has been to research and examine one chaotic system of interest for its viability as a random number generator using one method of bit extraction. Some selected other methods have also been briefly explored. Future work could potentially incorporate other bit extraction methods or other chaotic systems as candidates for true random number generation. Also, overall system form factor could be reduced by incorporating sampling and bit extraction with the chaotic system on a single printed circuit board.

Other bit extraction techniques include sampling other nodes of the circuit (e.g. the $\dot{x}$ node or the signum node), or by using multiple oscillators in tandem. After confirming that the multiple oscillators are not synchronized (which would nullify any advantage of using more than one), they could then be fed through an exclusive OR gate whose output can then be sampled at a potentially higher rate since the overall capacity for RNG has increased with more oscillators. These oscillators could also be inputs into another analog operation such as multiplication, which could be sampled with an ADC with higher speed or lower resolution.

There are also bit manipulation techniques involving bitwise operations between successive samples, or by taking more than one bit per sample and then operating on the group of bits as a whole. These techniques would take more processing between samples, which might limit implementations in embedded or high speed applications. However, they have the potential to allow for multiple bit streams into separate random number sinks from the improved bit rate provided from these techniques.

Multiple other chaotic systems exist that are based on mathematical equations that also have been implemented in high speed electronics. By utilizing the framework for maximum

Lyapunov exponent determination, the maximum bit rate of these chaotic systems can be found in relation to their natural frequency, which can then be scaled to higher frequencies as shown with the jerk chaos system. Also, the ADC resolution needed for sampling these systems can be found in simulation and then confirmed in hardware at the higher frequency.

System form factor can be reduced further by using a dedicated FPGA and ADC on the same circuit board as the chaotic system. The ADC would perform sampling of the chaotic system at the specified resolution and pass the samples to the FPGA. The FPGA would then do any post processing (Von Neumann correction, bitwise operations, etc.) on the samples and then pass the bit sequence continually to a PC. This would allow for Dieharder testing without rewinding a file, which is the preferred method for testing random number generators in the test suite.

The ultimate reduction in system form factor would result in the entire system (chaotic equation, sampling, bit manipulation, and data transfer) all being implemented into an application specific integrated circuit (ASIC). The chaotic circuit itself has been layed out on a printed circuit board in the same manner that it would topologically be implemented into the area in an ASIC. This would result in a “chip” that could be dropped into any security focused system in order to provide true random number generation for encryption. This would probably require a special fabrication process that could combine the high bandwidth analog requirements of the chaotic oscillator with the digital requirements for communication, but this research is left for future study.

References

[1] K. Sayood, *Lossless compression handbook.* Elsevier, 2002.

[2] P. Diaconis, S. Holmes, and R. Montgomery, "Dynamical bias in the coin toss," *SIAM review*, vol. 49, no. 2, pp. 211–235, 2007.

[3] V. Z. Vulović and R. E. Prange, "Randomness of a true coin toss," *Physical Review A*, vol. 33, no. 1, p. 576, 1986.

[4] J. Strzałko, J. Grabski, A. Stefański, P. Perlikowski, and T. Kapitaniak, "Dynamics of coin tossing is predictable," *Physics reports*, vol. 469, no. 2, pp. 59–92, 2008.

[5] C. E. Shannon and W. Weaver, *The mathematical theory of communication.* University of Illinois press, 1998.

[6] C. E. Shannon, "Prediction and entropy of printed english," *Bell system technical journal*, vol. 30, no. 1, pp. 50–64, 1951.

[7] B. Poulain, "Linux certif." [Online]. Available: http://www.linuxcertif.com/man/1/ent/

[8] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of the Atmospheric Sciences*, vol. 20, no. 2, pp. 130–141, 1963.

[9] R. M. May, "Simple mathematical models with very complicated dynamics," *Nature*, vol. 261, no. 5560, p. 459, 1976.

[10] P. L'Ecuyer and F. Panneton, "A new class of linear feedback shift register generators," in *Proceedings of the 32nd conference on Winter simulation.* Society for Computer Simulation International, 2000, pp. 690–696.

[11] X. Wang, X. Wang, J. Zhao, and Z. Zhang, "Chaotic encryption algorithm based on alternant of stream cipher and block cipher," *Nonlinear Dynamics*, vol. 63, no. 4, pp. 587–597, 2011.

[12] "NIST SP 800-22: Documentation and Software - Random Bit Generation — CSRC," https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software, accessed: 2018-12-4.

[13] "Dieharder: A Random Number Test Suite," https://webhome.phy.duke.edu/~rgb/General/dieharder.php, accessed: 2018-12-4.

[14] A. J. Pinkerton and L. Li, "Direct additive laser manufacturing using gas-and water-atomised h13 tool steel powders," *The International Journal of Advanced Manufacturing Technology*, vol. 25, no. 5-6, pp. 471–479, 2005.

[15] M. Skorczakowski, J. Swiderski, W. Pichola, P. Nyga, A. Zajac, M. Maciejewska, L. Galecki, J. Kasprzak, S. Gross, A. Heinrich *et al.*, "Mid-infrared q-switched er: Yag laser for medical applications," *Laser Physics Letters*, vol. 7, no. 7, p. 498, 2010.

[16] R. Hashimoto, A. Ito, and T. Goto, "Effect of deposition atmosphere on the phase composition and microstructure of silicon carbide films prepared by laser chemical vapour deposition," *Ceramics International*, vol. 41, no. 5, pp. 6898–6904, 2015.

[17] D. Kade, K. Akşit, H. Ürey, and O. Özcan, "Head-mounted mixed reality projection display for games production and entertainment," *Personal and Ubiquitous Computing*, vol. 19, no. 3-4, pp. 509–521, 2015.

[18] P. G. Halverson, A. Kuhnert, J. Logan, M. Regehr, S. Shaklan, R. Spero, F. Zhao, T. Chang, E. Schmidtlin, R. Gutierrez *et al.*, "Progress towards picometer accuracy laser metrology for the space interferometry mission," in *International Conference on Space OpticsICSO 2000*, vol. 10569. International Society for Optics and Photonics, 2017, p. 1056919.

[19] H. Asghar, W. Wei, P. Kumar, D. Marah, E. Sooudi, and J. G. McInerney, "Sub-picosecond pulse stability of passively mode-locked two-section quantum dash laser at 1550 nm subject to single and dual optical feedback," in *Lasers and Electro-Optics (CLEO), 2016 Conference on*. IEEE, 2016, pp. 1–2.

[20] T. Taimre, M. Nikolić, K. Bertling, Y. L. Lim, T. Bosch, and A. D. Rakić, "Laser feedback interferometry: a tutorial on the self-mixing effect for coherent sensing," *Advances in Optics and Photonics*, vol. 7, no. 3, pp. 570–631, 2015.

[21] J. Li, Y. Niu, and H. Niu, "Measurement of phase retardation of optical multilayer films based on laser feedback system," *Optics express*, vol. 24, no. 1, pp. 409–416, 2016.

[22] C. O. Weiss and J. Brock, "Evidence for lorenz-type chaos in a laser," *Phys. Rev. Lett.*, vol. 57, pp. 2804–2806, Dec 1986. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.57.2804

[23] T. Midavaine, D. Dangoisse, and P. Glorieux, "Observation of chaos in a frequency-modulated co_2 laser," *Phys. Rev. Lett.*, vol. 55, pp. 1989–1992, Nov 1985. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.55.1989

[24] C. Weiss, W. Klische, P. Ering, and M. Cooper, "Instabilities and chaos of a single mode nh3 ring laser," *Optics communications*, vol. 52, no. 6, pp. 405–408, 1985.

[25] F. Arecchi, G. Lippi, G. Puccioni, and J. Tredicce, "Deterministic chaos in laser with injected signal," *Optics communications*, vol. 51, no. 5, pp. 308–314, 1984.

[26] J. Ohtsubo, *Semiconductor lasers: stability, instability and chaos*. Springer, 2012, vol. 111.

[27] G. E. James, E. M. Harrell, C. Bracikowski, K. Wiesenfeld, and R. Roy, "Elimination of chaos in an intracavity-doubled nd: Yag laser," *Optics letters*, vol. 15, no. 20, pp. 1141–1143, 1990.

[28] M. Tsunekane, N. Taguchi, and H. Inaba, "Elimination of chaos in a multilongitudinal-mode, diode-pumped, 6-w continuous-wave, intracavity-doubled nd: Yag laser," *Optics letters*, vol. 22, no. 13, pp. 1000–1002, 1997.

[29] G. P. Agrawal, "Effect of gain nonlinearities on period doubling and chaos in directly modulated semiconductor lasers," *Applied physics letters*, vol. 49, no. 16, pp. 1013–1015, 1986.

[30] J. Ye, H. Li, and J. G. McInerney, "Period-doubling route to chaos in a semiconductor laser with weak optical feedback," *Physical Review A*, vol. 47, no. 3, p. 2249, 1993.

[31] T. B. Simpson, J. M. Liu, A. Gavrielides, V. Kovanis, and P. M. Alsing, "Period-doubling cascades and chaos in a semiconductor laser with optical injection," *Phys. Rev. A*, vol. 51, pp. 4181–4185, May 1995. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.51.4181

[32] F.-Y. Lin and J.-M. Liu, "Chaotic radar using nonlinear laser dynamics," *IEEE Journal of Quantum Electronics*, vol. 40, no. 6, pp. 815–820, June 2004.

[33] T. Sugawara, M. Tachikawa, T. Tsukamoto, and T. Shimizu, "Observation of synchronization in laser chaos," *Phys. Rev. Lett.*, vol. 72, pp. 3502–3505, May 1994. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.72.3502

[34] J.-P. Goedgebuer, L. Larger, and H. Porte, "Optical cryptosystem based on synchronization of hyperchaos generated by a delayed feedback tunable laser diode," *Phys. Rev. Lett.*, vol. 80, pp. 2249–2252, Mar 1998. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.80.2249

[35] T. Yamazaki and A. Uchida, "Performance of random number generators using noise-based superluminescent diode and chaos-based semiconductor lasers," *IEEE Journal of Selected Topics in Quantum Electronics*, vol. 19, no. 4, pp. 0 600 309–0 600 309, 2013.

[36] K. Hirano, T. Yamazaki, S. Morikatsu, H. Okumura, H. Aida, A. Uchida, S. Yoshimori, K. Yoshimura, T. Harayama, and P. Davis, "Fast random bit generation with bandwidth-enhanced chaos in semiconductor lasers," *Optics express*, vol. 18, no. 6, pp. 5512–5524, 2010.

[37] I. Reidler, Y. Aviad, M. Rosenbluh, and I. Kanter, "Ultrahigh-speed random number generation based on a chaotic semiconductor laser," *Phys. Rev. Lett.*, vol. 103, p. 024102, Jul 2009. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.103.024102

[38] R. M. Nguimdo, G. Verschaffelt, J. Danckaert, X. Leijtens, J. Bolk, and G. Van der Sande, "Fast random bits generation based on a single chaotic semiconductor ring laser," *Optics express*, vol. 20, no. 27, pp. 28 603–28 613, 2012.

[39] N. J. Corron, M. T. Stahl, R. C. Harrison, and J. N. Blakely, "Acoustic detection and ranging using solvable chaos," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 23, no. 2, p. 023119, 2013.

[40] V. Venkatasubramanian and H. Leung, "A robust chaos radar for collision detection and vehicular ranging in intelligent transportation systems," in *Intelligent Transportation Systems, 2004. Proceedings. The 7th International IEEE Conference on*. IEEE, 2004, pp. 548–552.

[41] A. Beal, J. Blakely, N. Corron, and R. Dean, "High frequency oscillators for chaotic radar," in *SPIE Defense+ Security*. International Society for Optics and Photonics, 2016, pp. 98 290H–98 290H.

[42] A. N. Beal and R. N. Dean, "A random stimulation source for evaluating mems devices using an exact solvable chaotic oscillator," *Additional Papers and Presentations*, vol. 2015, no. DPC, pp. 001 594–001 625, 2015.

[43] A. Volkovskii, L. S. Tsimring, N. Rulkov, and I. Langmore, "Spread spectrum communication system with chaotic frequency modulation," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 15, no. 3, p. 033101, 2005.

[44] T. Stojanovski and L. Kocarev, “Chaos-based random number generators-part i: analysis [cryptography],” *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 48, no. 3, pp. 281–288, Mar 2001.

[45] T. Stojanovski, J. Pihl, and L. Kocarev, “Chaos-based random number generators. part ii: practical realization,” *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 48, no. 3, pp. 382–385, Mar 2001.

[46] S. Callegari, R. Rovatti, and G. Setti, “Spectral properties of chaos-based fm signals: theory and simulation results,” *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 50, no. 1, pp. 3–15, Jan 2003.

[47] S. Ergn, “Security analysis of a chaos-based random number generator for applications in cryptography,” in *2015 15th International Symposium on Communications and Information Technologies (ISCIT)*, Oct 2015, pp. 319–322.

[48] F. Pareschi, G. Scotti, L. Giancane, R. Rovatti, G. Setti, and A. Trifiletti, “Power analysis of a chaos-based random number generator for cryptographic security,” in *2009 IEEE International Symposium on Circuits and Systems*, May 2009, pp. 2858–2861.

[49] M. Blaszczyk and R. A. Guinee, “A true random binary sequence generator based on chaotic circuit,” in *Signals and Systems Conference, 208. (ISSC 2008). IET Irish*, June 2008, pp. 294–299.

[50] A. N. Beal, J. P. Bailey, S. Hale, R. N. Dean, M. Hamilton, J. K. Tugnait, D. W. Hahs, and N. J. Corron, “Design and simulation of a high frequency exact solvable chaotic oscillator,” in *MILCOM 2012-2012 IEEE Military Communications Conference*. IEEE, 2012, pp. 1–6.

[51] R. A. Guinee and M. Blaszczyk, “A novel true random binary sequence generator based on a chaotic double scroll oscillator combination with a pseudo random generator for cryptographic applications,” in *Internet Technology and Secured Transactions, 2009. ICITST 2009. International Conference for*, Nov 2009, pp. 1–6.

[52] S. Ergun and S. Ozoguz, "A chaos-modulated dual oscillator-based truly random number generator," in *2007 IEEE International Symposium on Circuits and Systems*, May 2007, pp. 2482–2485.

[53] S. Ozoguz and N. S. Sengor, "On the realization of npn-only log-domain chaotic oscillators," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 50, no. 2, pp. 291–294, Feb 2003.

[54] T. Saito and H. Fujita, "Chaos in a manifold piecewise linear system," *Electronics and Communications in Japan (Part I: Communications)*, vol. 64, no. 10, pp. 9–17, 1981.

[55] M. E. Yalcin, J. A. K. Suykens, and J. Vandewalle, "True random bit generation from a double-scroll attractor," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 7, pp. 1395–1404, July 2004.

[56] A. Wang, L. Wang, and Y. Wang, "Post-processing-free 400 gb/s true random number generation using optical heterodyne chaos," in *2016 25th Wireless and Optical Communication Conference (WOCC)*, May 2016, pp. 1–4.

[57] J. Sprott, *Elegant Chaos: Algebraically Simple Chaotic Flows*. World Scientific, 2010. [Online]. Available: https://books.google.com/books?id=buILBDre9S4C

[58] J. C. Sprott, "Simple chaotic systems and circuits," *American Journal of Physics*, vol. 68, no. 8, pp. 758–763, 2000.

[59] S. Ergün, "On the security of a double-scroll based" true" random bit generator," in *Signal Processing Conference (EUSIPCO), 2015 23rd European*. IEEE, 2015, pp. 2058–2061.

[60] A. G. Radwan, A. M. Soliman, and A. L. El-Sedeek, "Mos realization of the double-scroll-like chaotic equation," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 50, no. 2, pp. 285–288, Feb 2003.

[61] J. C. Sprott, "Simple chaotic systems and circuits," *American Journal of Physics*, vol. 68, no. 8, pp. 758–763, 2000.

[62] R. C. Harrison, B. K. Rhea, F. T. Werner, and R. N. Dean, “A 4 mhz chaotic oscillator based on a jerk system,” in *2016 International Conference on Applications in Nonlinear Dynamics*, 2016.

Appendices

Appendix A

Compiling and Using Dieharder on Windows

This section is a combination of an installation guide for Dieharder in a Windows environment, as well as a low-level usage guide for Dieharder from the command line. The best resource for general questions about the test suite encapsulated within Dieharder is the main Dieharder page [13]. Installing Dieharder in a Linux environment is much easier and is actively developed for Linux. The command `install dieharder` should be all that is required for installing and updating Dieharder within Linux. Usage between Windows and Linux should mostly be the same, save for a few commands dealing with copying output files from Dieharder to other processes.

Cygwin will be used to compile Dieharder from the source code available on the Dieharder website. Download the `setup-x86-64.exe` file from the home page and run it. Select the default options for installation until you get to the "Select Packages" screen, shown in Fig. A.1. The packages listed in the figure are needed in order for Dieharder to compile. By searching for the packages under `View->Full` and double-clicking on `Skip` in order to select the most recent version of the packages, they can be placed in the queue for installation (viewed under `View->Pending`.) Once this is done, click `Next` to confirm these changes and start the package downloads.

Once the Cygwin installation is finished, the Dieharder installation can begin. Launching Cygwin will provide a command line interface as shown in Fig. A.2. A few commands are useful going forward for those unfamiliar with general Linux commands: `cd` changes current directory, and `ls` lists the current files in the current path. Pressing `Tab` in the middle of typing a command will attempt to autofill the rest of the command (useful for finishing filenames).

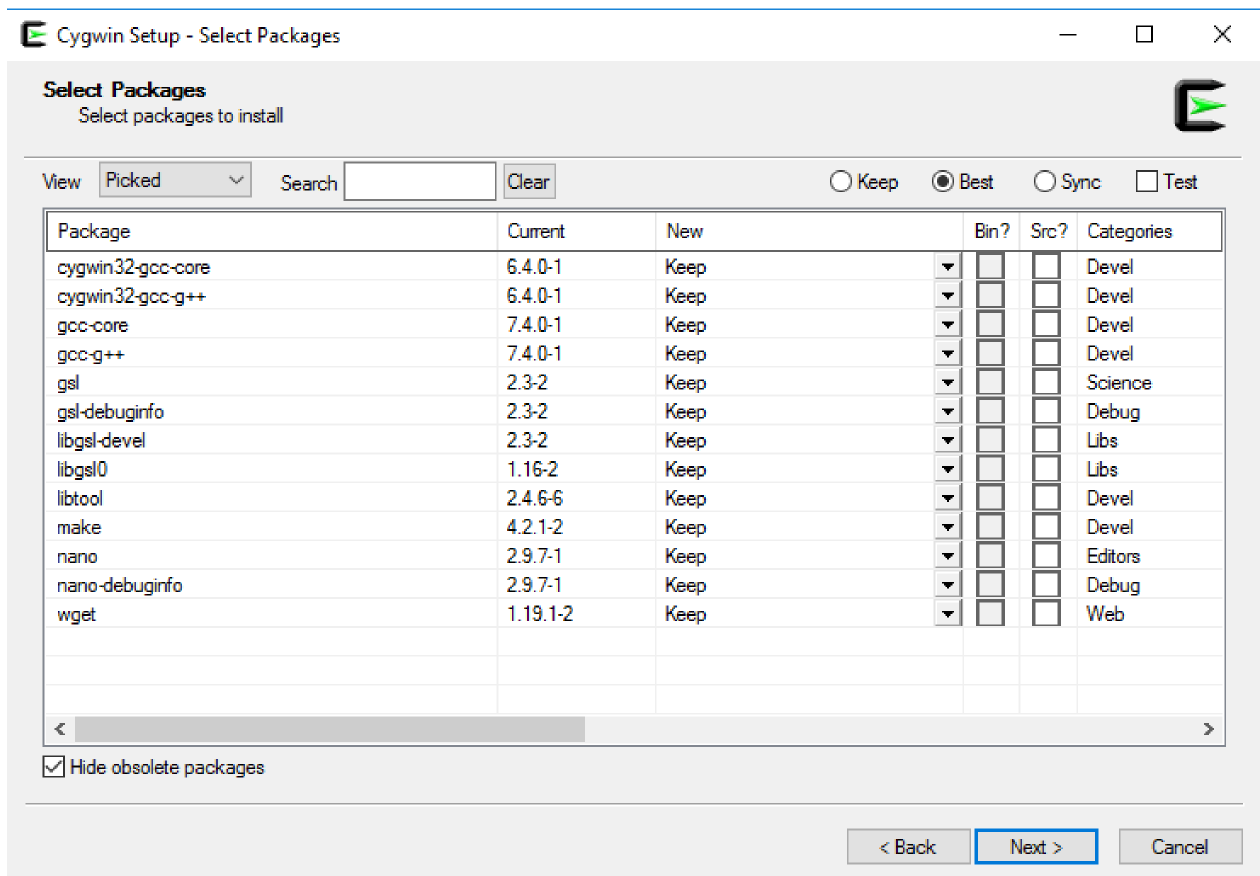

Figure A.1: The select packages screen within Cygwin Setup. Make sure that at least these packages are installed.

First, download the latest `.tgz` file of the Dieharder source using `wget`. At the time of writing, this command would be

`wget https://webhome.phy.duke.edu/~rgb/General/dieharder/dieharder-3.31.1.tgz`.

Use `ls` to confirm that the file is downloaded. Then, unpack this file with

`tar -zxf dieharder-3.31.1.tgz`. Using `ls` again will show the original downloaded file as well as a new `dieharder` folder. `cd` into this directory.

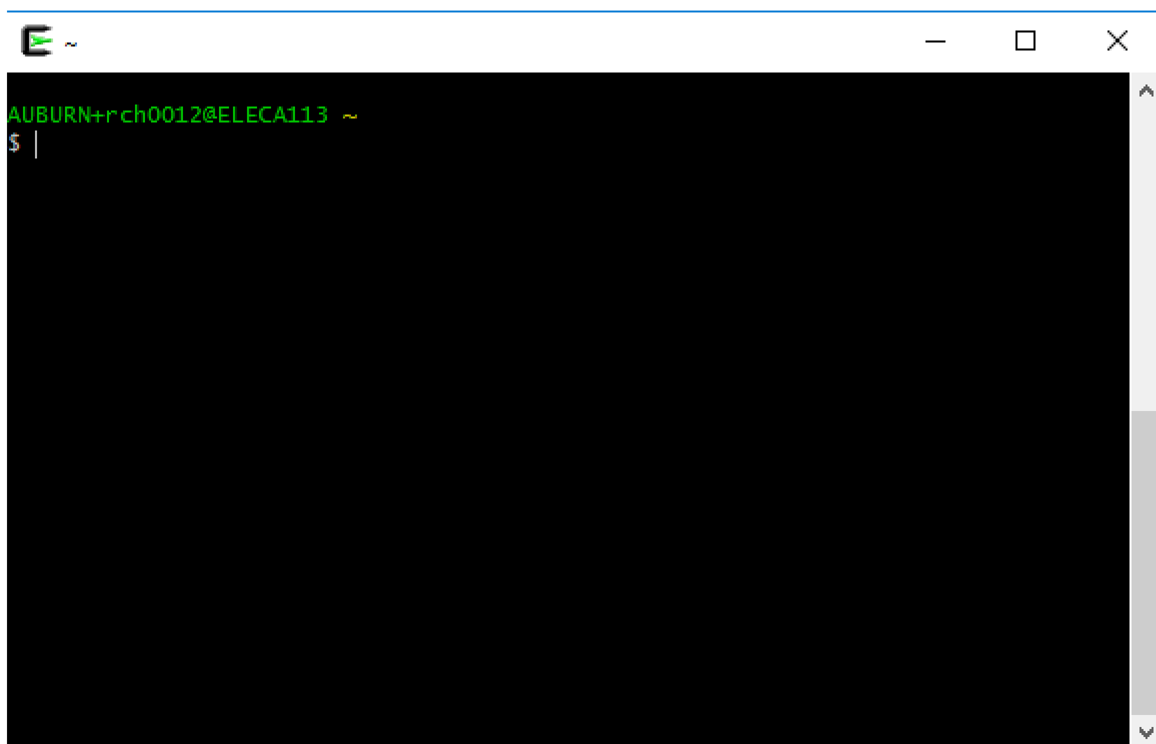

Figure A.2: Cygwin's terminal

Once in the `dieharder` folder, run `./configure` to start the first half of the compilation process. This command will check to make sure all of the necessary package files for compilation are in place. If an error occurs here that lists a specific package still needed, simply run the Cygwin setup file again and select the needed package as before. Once the configure command is finished successfully, type `make`.

`make` will begin the actual compilation process. It is very likely that it will fail halfway through. Currently, at least two changes are needed in the source files to fix the compilation errors with the given source code. The first error thrown is due to an undefined `M_PI` (meaning π). The next error is `uint` (meaning `unsigned int`) being undefined in multiple places.

cd into /include/dieharder and type nano libdieharder.h. This will open this header file in a command line text editor. Navigate using the arrow keys and mouse to the section labelled "Useful Defines" and add the following:

#define M_PI 3.14159265358979323846264 3 (nothing after the final digit). Then, navigate to the end of the file and at the end add typedef unsigned int uint; (remember the semicolon). Press Ctrl-X and then Y->Enter to save the changes and overwrite the original file. After these two fixes, running make in the original directory (cd .. moves up a directory) should finish the compilation. Using the file explorer in Windows, navigate to the Cygwin folder (most likely

C:\cygwin64\home\username\dieharder-3.31.1\dieharder if using the default Cygwin setup). A dieharder.exe should now be in this folder. Shift + Right-clicking on whitespace within this folder and choosing Open command window here will quickly open up a Windows terminal. Entering dieharder within the terminal should confirm that the program compiled and runs successfully, as shown in Fig. A.3.

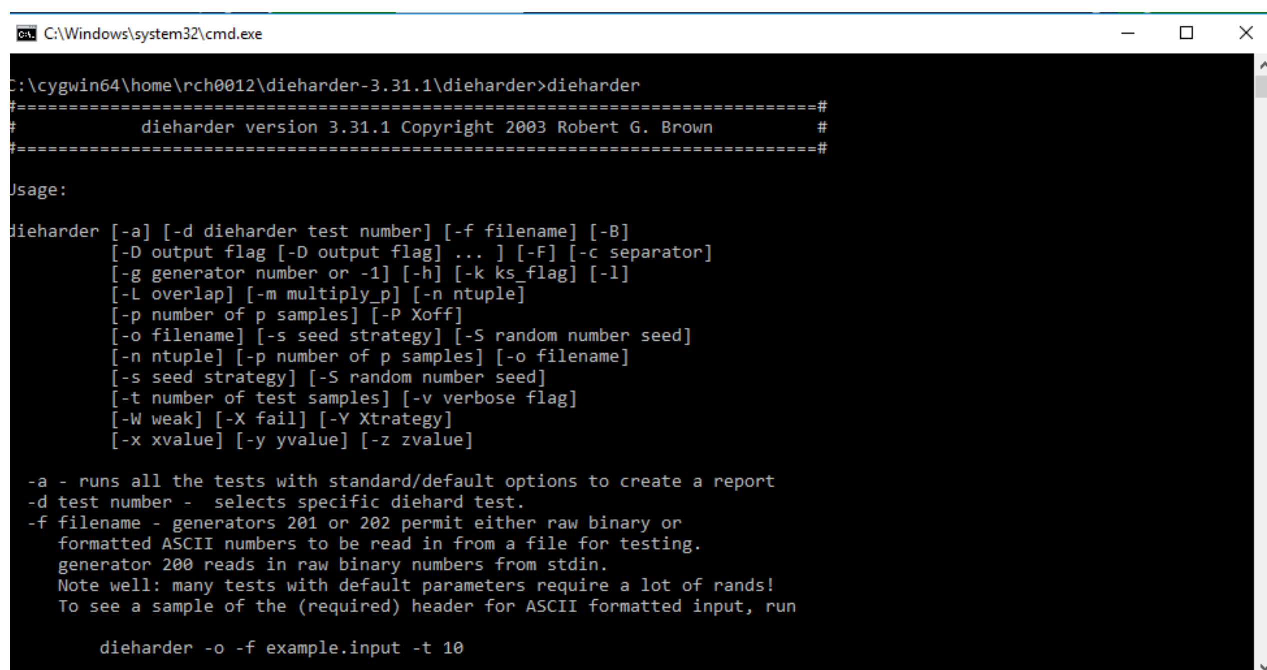

Figure A.3: Dieharder output with no commands given

Appendix B

Creating a Printed Circuit Board using KiCad

In this section, the KiCad PCB suite is presented and a tutorial is given for going from a schematic to a full PCB layout. This tutorial is not intended to go over all of the proper design techniques when laying out a PCB, rather it is intended to show how to navigate the KiCad menus as well as practical steps to take in order to achieve a desired PCB design.

We will begin by creating a blank project in KiCad. It is recommended that a fresh folder is used for the PCB design so that all of the created files are kept in one place. Once this is done, you should arrive at the home screen of KiCad as shown in Fig. B.1.

We will then launch the schematic editing software known as "EESchema" from within KiCad. This is done by simply clicking on the leftmost icon on the main KiCad homescreen toolbar. This should present you with a blank EEschema layout as shown in Fig. B.2.

It is now time to enter our schematic that we want to make into a PCB. For this tutorial, we will make a simple inverting op amp board. This will consist of an op amp chip as well as a few passive components. An example of picking and placing a component is shown in Fig. B.3. An example of a basic schematic that has been created is shown in Fig. B.4. Keep in mind that eventually the PCB will need to be populated with real components; throughout the design process, make sure that you are able to obtain the correct components with the right footprint.

Since we are making a PCB, we need to add a few extra items that aren't needed in a simulation. These are namely connectors for inputting power and input/output of signals, as well as some filter capacitors at our power input as well as for each integrated circuit. An updated schematic is shown in Fig. B.5. This schematic includes a power connector for providing $\pm 9V$ and ground to the op amp, as well as a bank of capacitors to be placed near this connector as

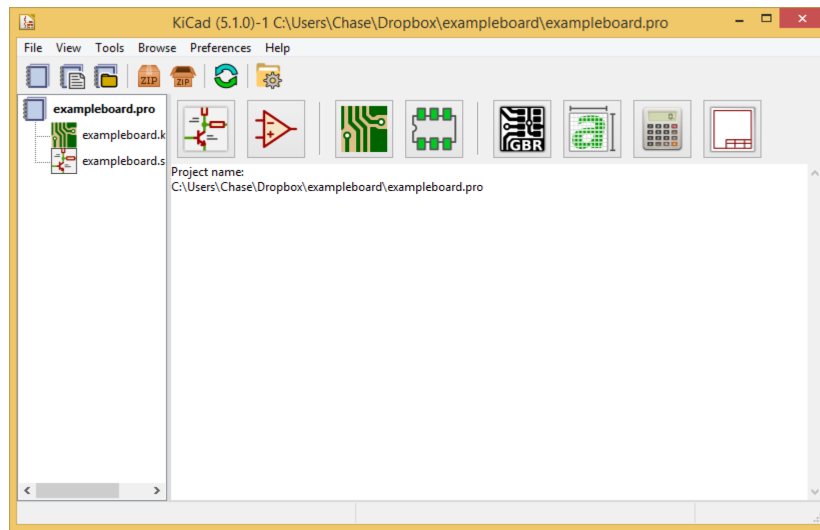

Figure B.1: The homescreen of KiCad after a new blank project has been created

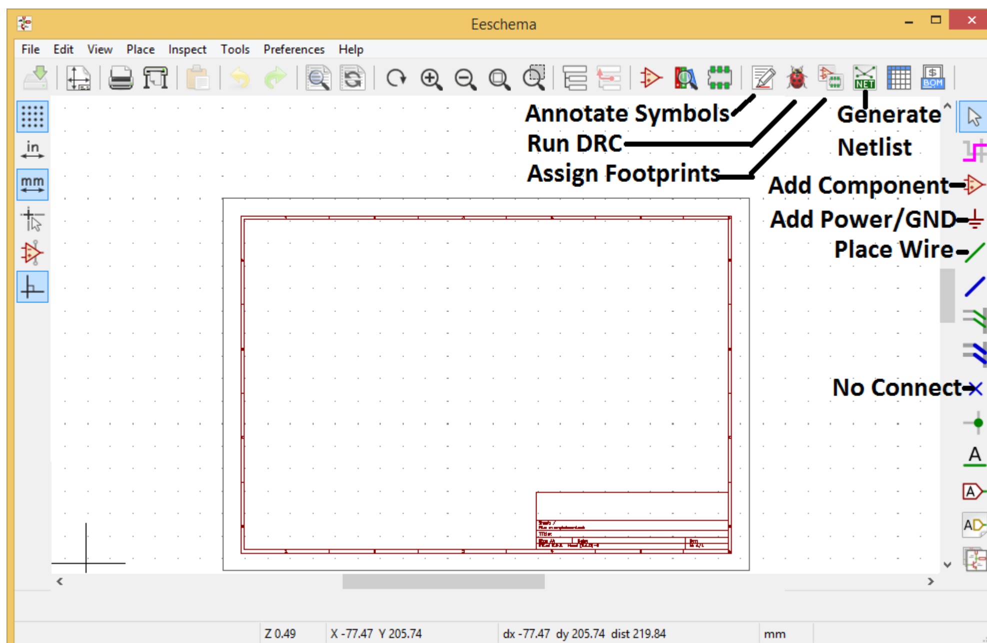

Figure B.2: A blank EESchema schematic, with labelled items

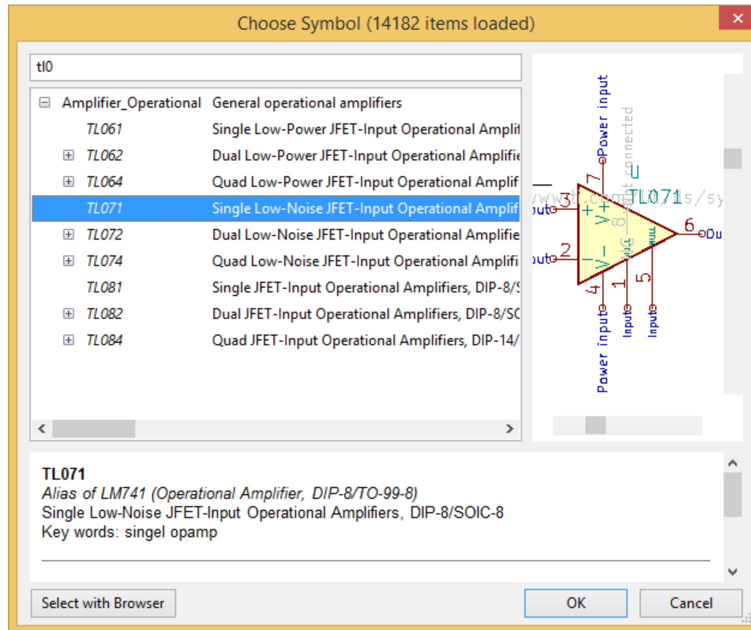

Figure B.3: Picking and placing a part withing EESchema

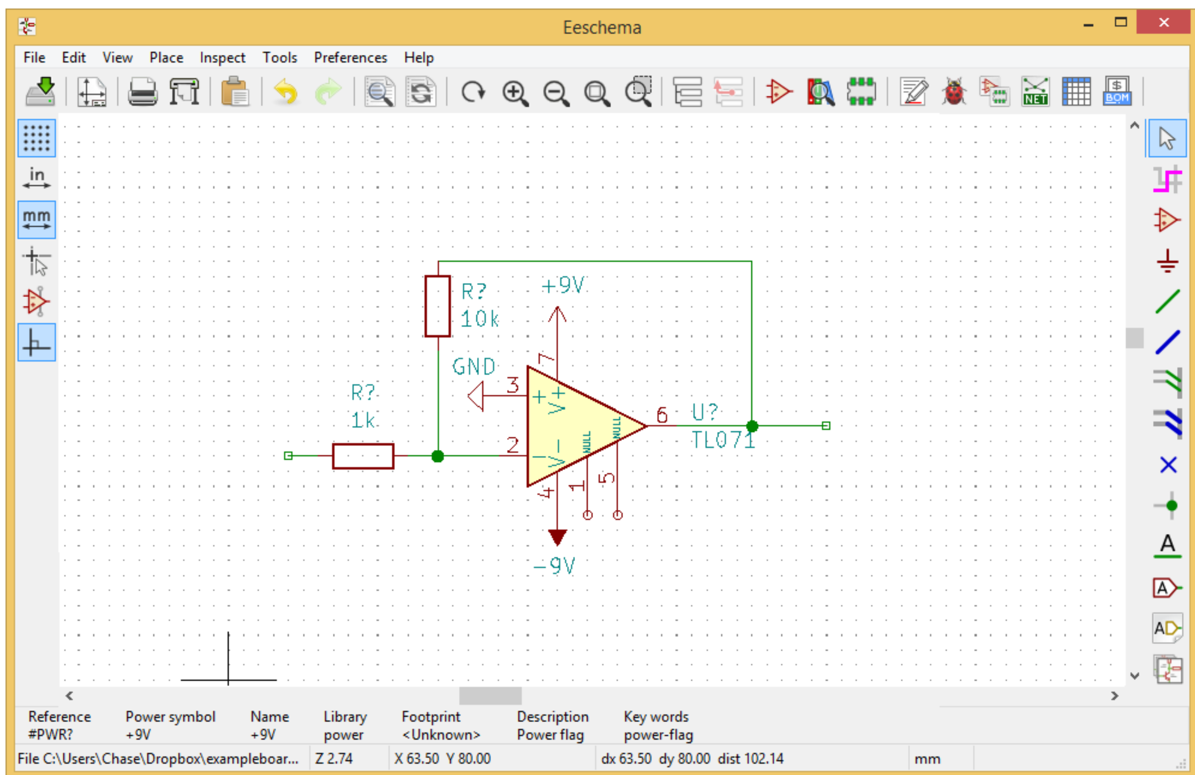

Figure B.4: A basic EESchema schematic of the op amp circuit

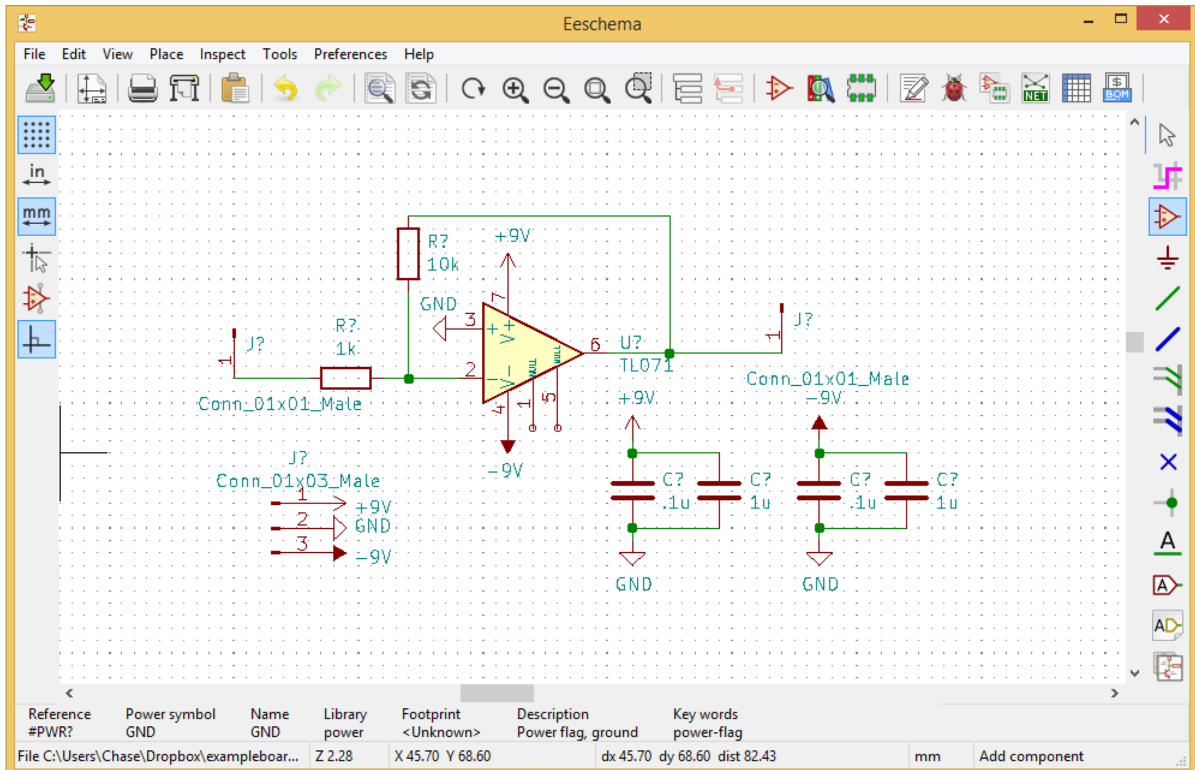

Figure B.5: A better EESchema schematic of the op amp circuit

well as near the chip. Additionally, pins are placed at the input and output of the op amp for getting the signals in and out.

This schematic will be sufficient for laying out a PCB. Before moving on, we need to annotate our schematic so that all of the parts are labelled. This can be accomplished by pressing “Annotate schematic symbols” on the upper EESchema toolbar. You should be presented with a window similar to Fig. B.6. There are a couple of options within this window if you are picky, but hitting the annotate button should be fine.

Before going further, we need to make sure that our schematic doesn’t violate any of our design rules. We haven’t listed any specific design rules for this project, so EESchema will use the default ones. Click on the ladybug icon within EESchema to begin. Hitting “Run” within the DRC window will begin the check. A screenshot of an example output is shown in Fig. B.7. In general, it is fine to move on with some warnings, but any errors need to be addressed.

Once the schematic is annotated and a Design Rules Check performed, we can assign footprints to the circuit. The footprints are the actual PCB layout of each individual component; it is important to assign the right footprints to each part. This process can be started by clicking

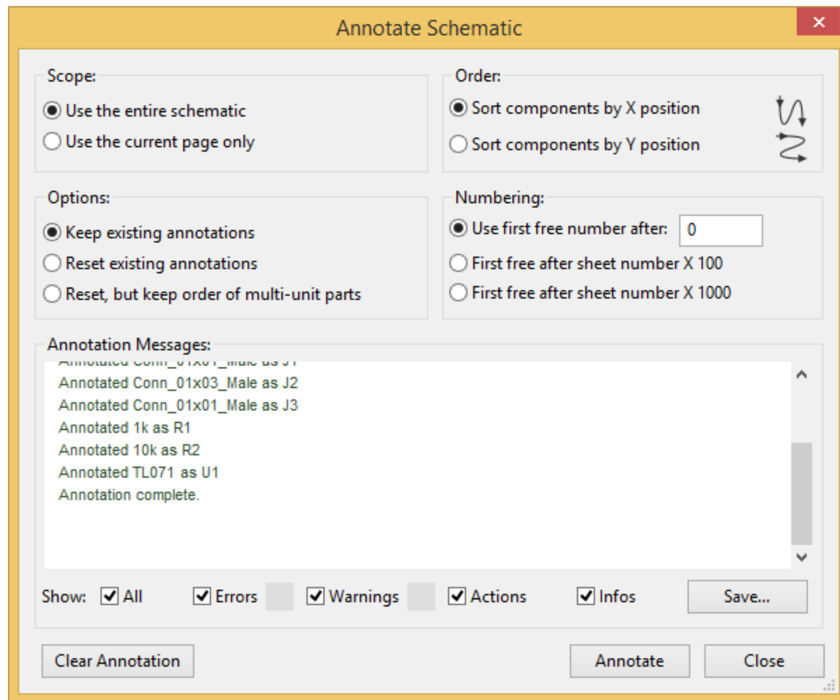

Figure B.6: Annotating the schematic

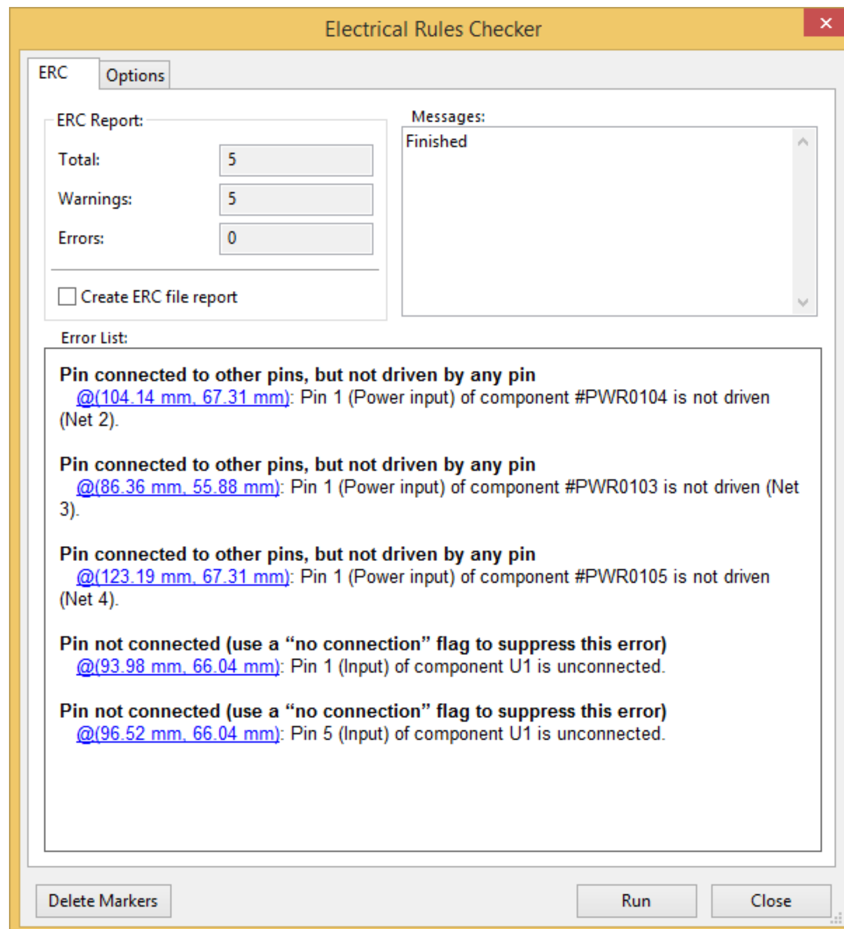

Figure B.7: A screenshot of the Design Rules Check output with some warnings

the “Assign PCB footprint to schematic symbols” button in the main toolbar. A window similar to Fig. B.8 should appear. On the left is a list of all of the categories of footprints available for assignment, on the right is a list of footprints within the selected category. The middle column shows a list of all of the components that are in the schematic. Above this middle column are four integrated circuit icons with labels. The “#” icon filters by pin count (i.e. only show footprints with the same number of terminals as the schematic symbol). The “L” icon filters the footprints by the selected library. The other two icons (text and “>”) are a keyword filter and a search string filter, respectively.

By going through the different libraries, the desired footprints can be found. To view the selected footprint, click the integrated circuit icon with the magnifying glass above the left column. Within the footprint pop up the 3D model can be displayed to get a better idea of size and shape of the actual component on the PCB. A picture of a completed footprint assignment is shown in Fig. B.9. Keep in mind that the footprint that is chosen must reflect the actual component that is ordered and meant to be populated. It is generally difficult to fit a component onto an incorrect footprint, especially integrated circuit chips.

Once the footprint assignment is saved, the netlist in EESchema can be generated. This will form a file that has all of the components in the schematic, as well as the footprint assignments, that can be read into the PCB editor. A screenshot of the generate netlist screen is shown in Fig. B.10. Save this netlist in the same folder as the other project files.

It is now time to begin actually laying out the printed circuit board. Within the main KiCad screen click on the “PCB Layout Editor” icon to launch the PCBnew program. A window showing a blank layout should appear similar to Fig. B.11. Click on the “Read Netlist” icon next to the ladybug icon on the top toolbar. This will present a screen similar to Fig. B.12. Once the netlist is read, the PCBnew screen should reappear on top with the parts that were read from the netlist directly under the mouse (shown in Fig. B.13). Place these components near the center of the blank layout.

The next thing to do is setup the layers of the PCB. Go to “Setup-Layers Setup” at the top menu to access a window similar to Fig. B.14. For this example PCB, a four layer design will be used. A two layer design would be very adequate for something of this magnitude, and

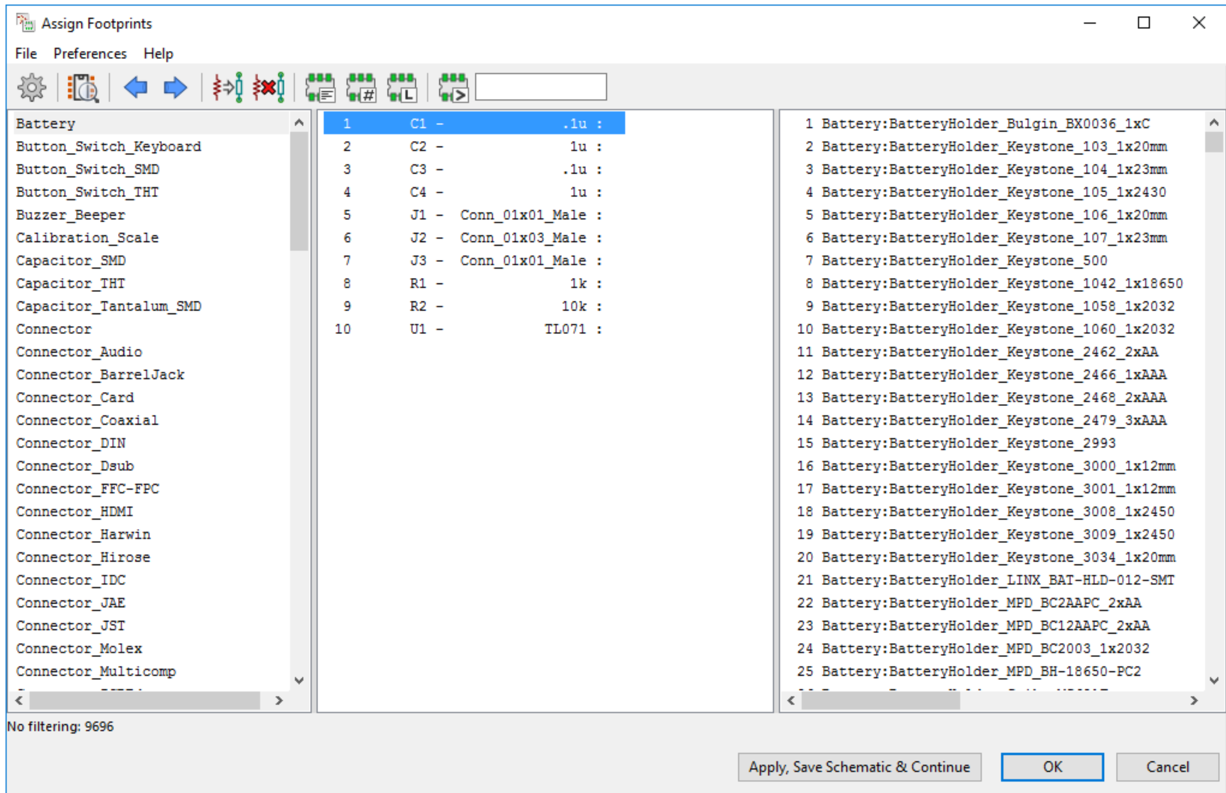

Figure B.8: A screenshot of the blank footprint assignment screen

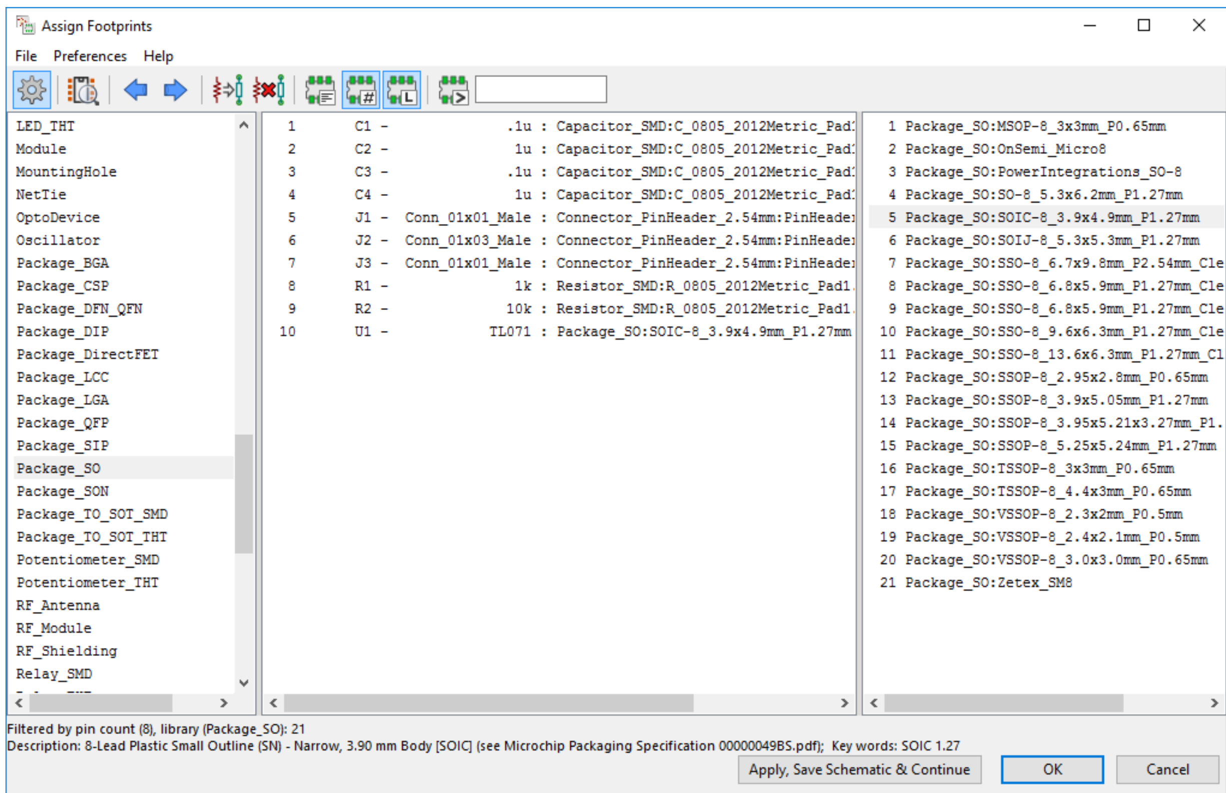

Figure B.9: A screenshot of the filled footprint assignment screen

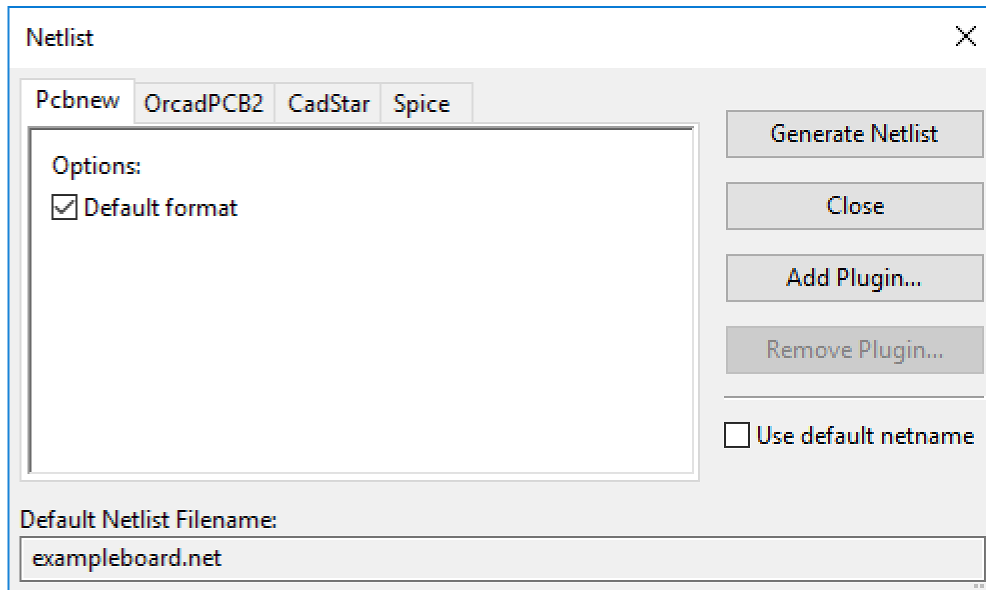

Figure B.10: A screenshot of the “Generate Netlist” screen in EESchema

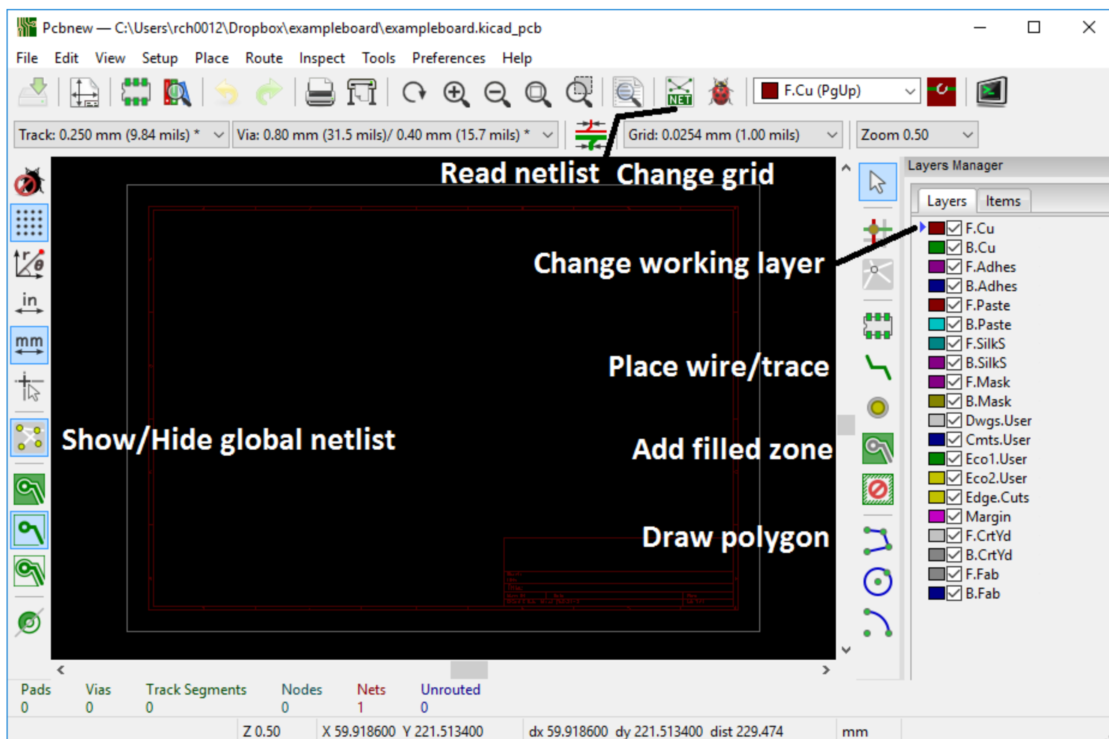

Figure B.11: A screenshot of the blank PCB layout in PCBnew

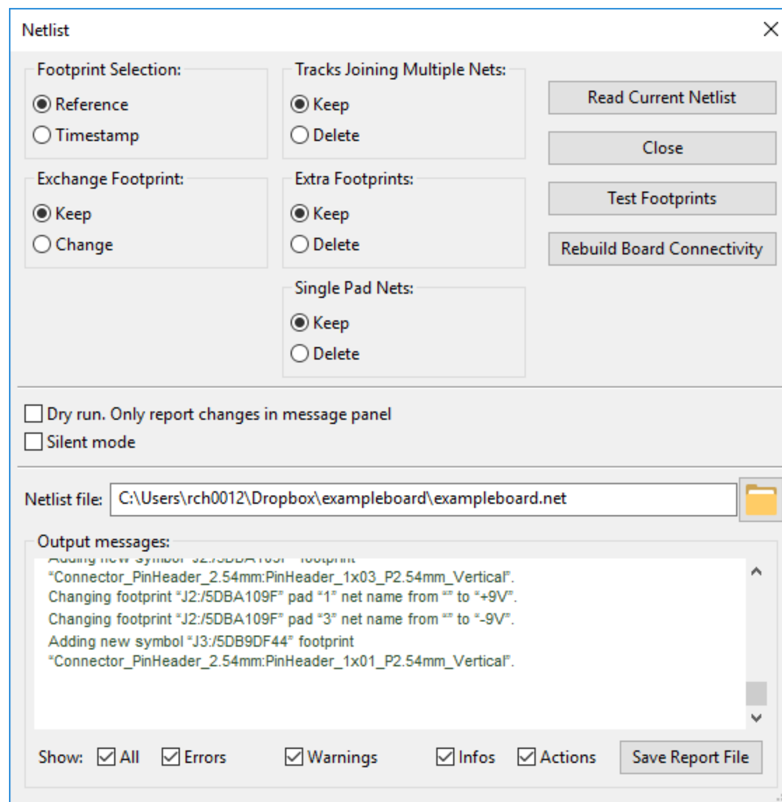

Figure B.12: A screenshot of the read netlist window in PCBnew

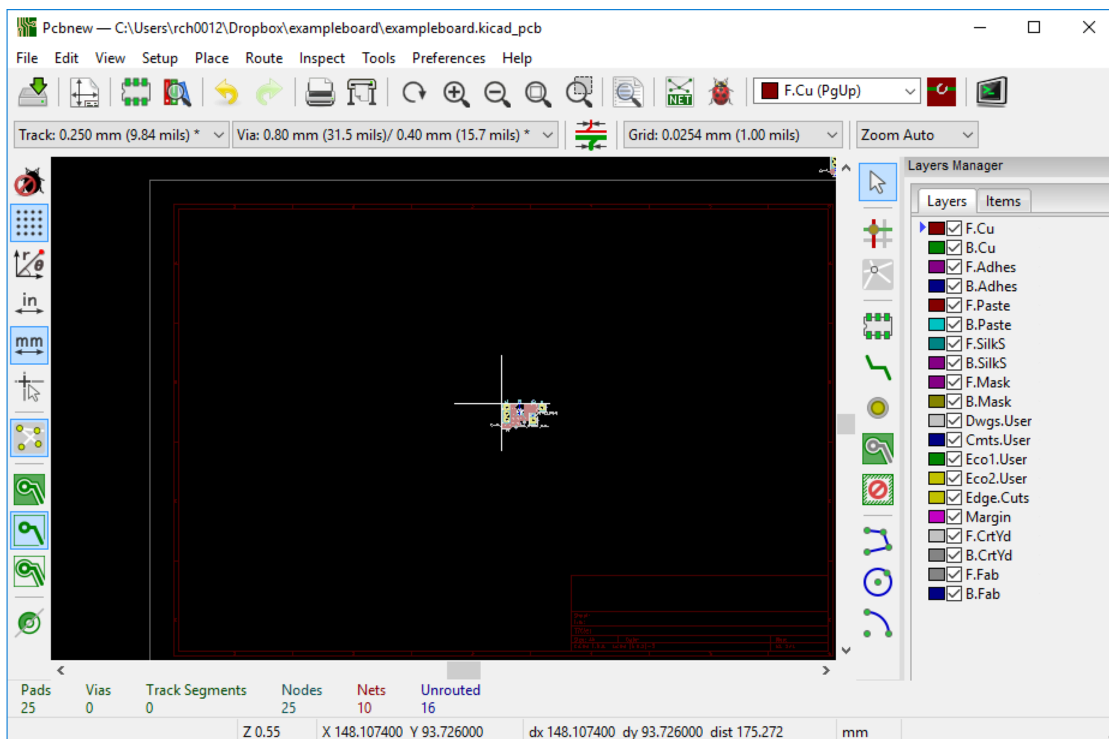

Figure B.13: A screenshot of the components ready to be placed in PCBnew

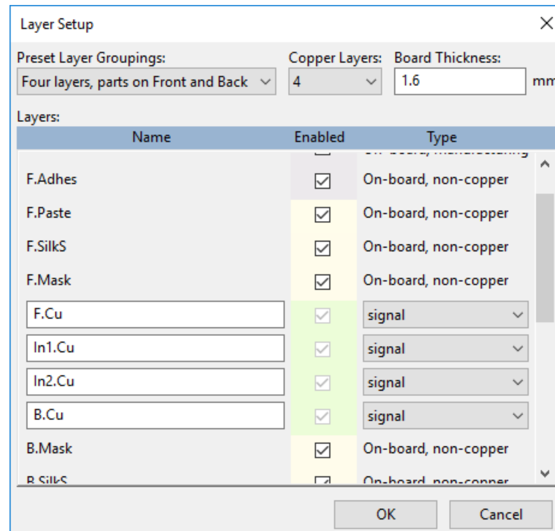

Figure B.14: A screenshot of the layer setup in PCBnew

two layer PCBs are cheaper to manufacture. Generally, if a design is low frequency and has only a few components, a two layer design is preferred. Higher frequency designs and complex designs with many components are much easier to lay out with four layers.

Next, the edge of the board needs to be defined. This technically can be done at any step, but it is preferred to do it early, especially if there are physical size or shape restraints for the PCB implementation. The edges of the board will be defined using a polygon (in this case a rectangle). Change the working layer to the "Edge Cuts" layer and select the "Draw polygon" icon (shown in Fig. B.15. Note that the grid has been changed to be large; this will increase ease in drawing the polygon as well as beginning and ending at the same corner. Draw the desired PCB shape around the placed components (or anywhere if desired), clicking the mouse once for each desired corner. Once the first corner has been reached again, click to place the corner and then right click to select "Cancel" to release the drawing. Ultimately a border similar to the one shown in Fig. B.16 should now be within PCBnew.

Now it is time to move the components to their desired locations. Note that each component has a few separate parts: the actual footprint, a reference text, and a value text. If PCBnew is unsure about which one is being selected, a context menu will appear. Simply select the desired part to move. Also, the grid has been decreased to give more granularity in component and text position. A few handy shortcuts (all of these can be viewed and edited from the "Preferences-Hotkey Options - Edit Hotkeys" top menu option):

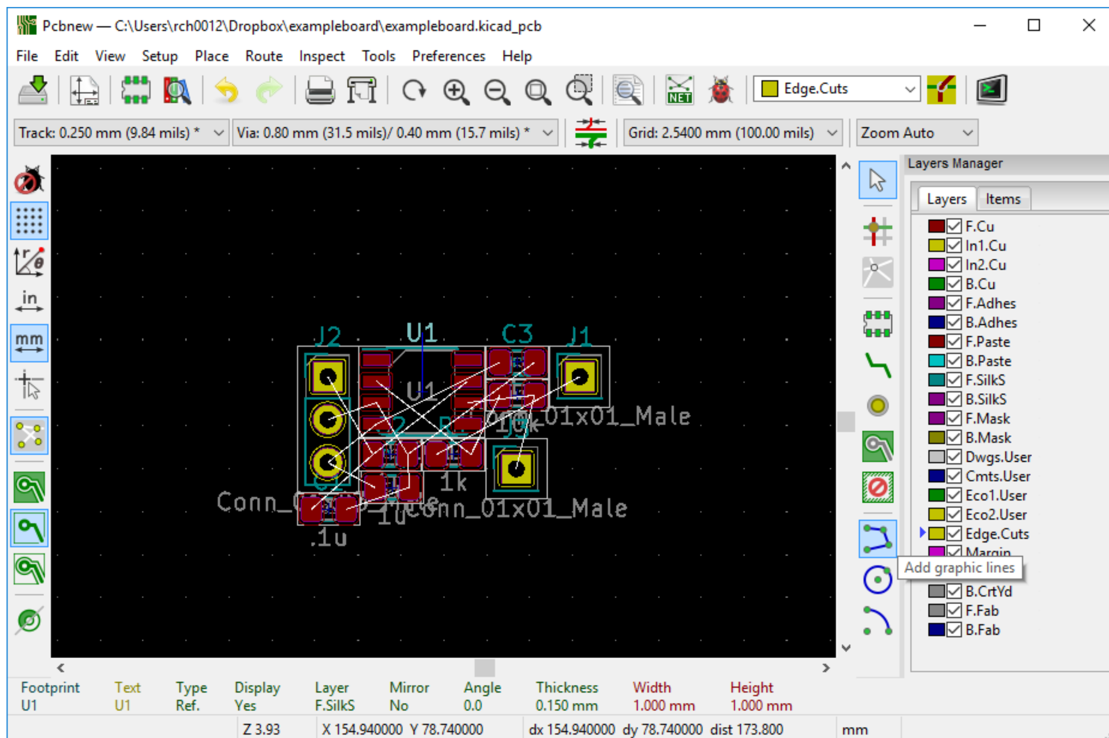

Figure B.15: Beginning step of creating the edges of the PCB

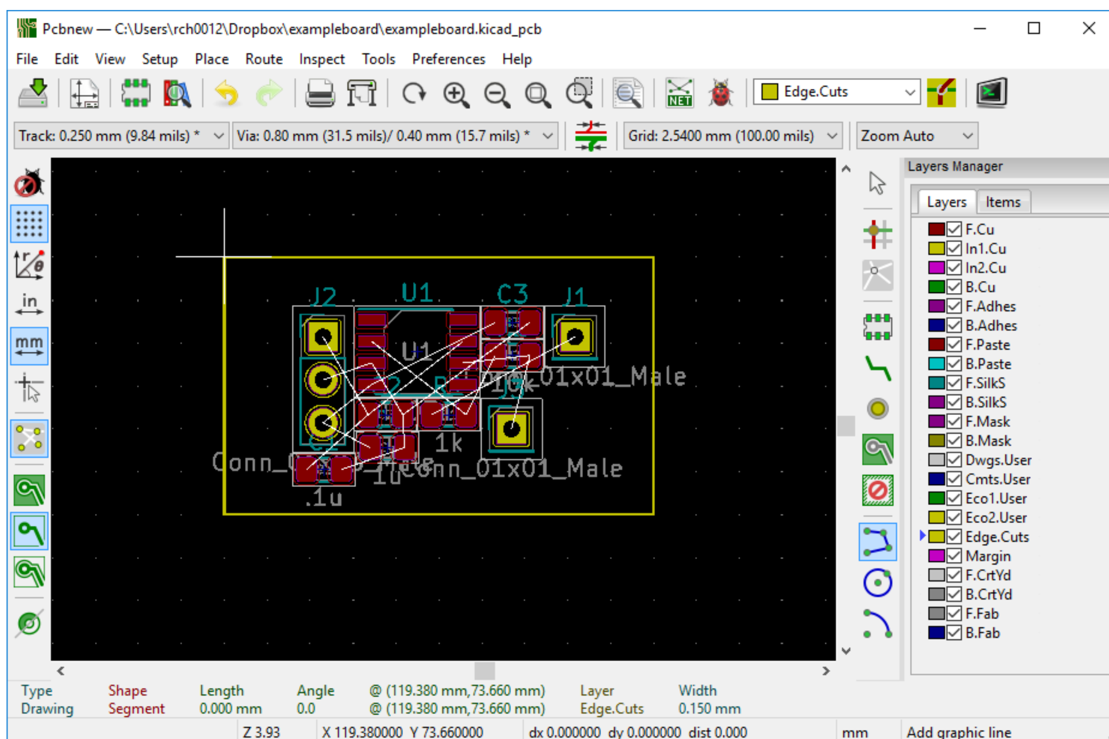

Figure B.16: A completed border that defines the edges of the PCB

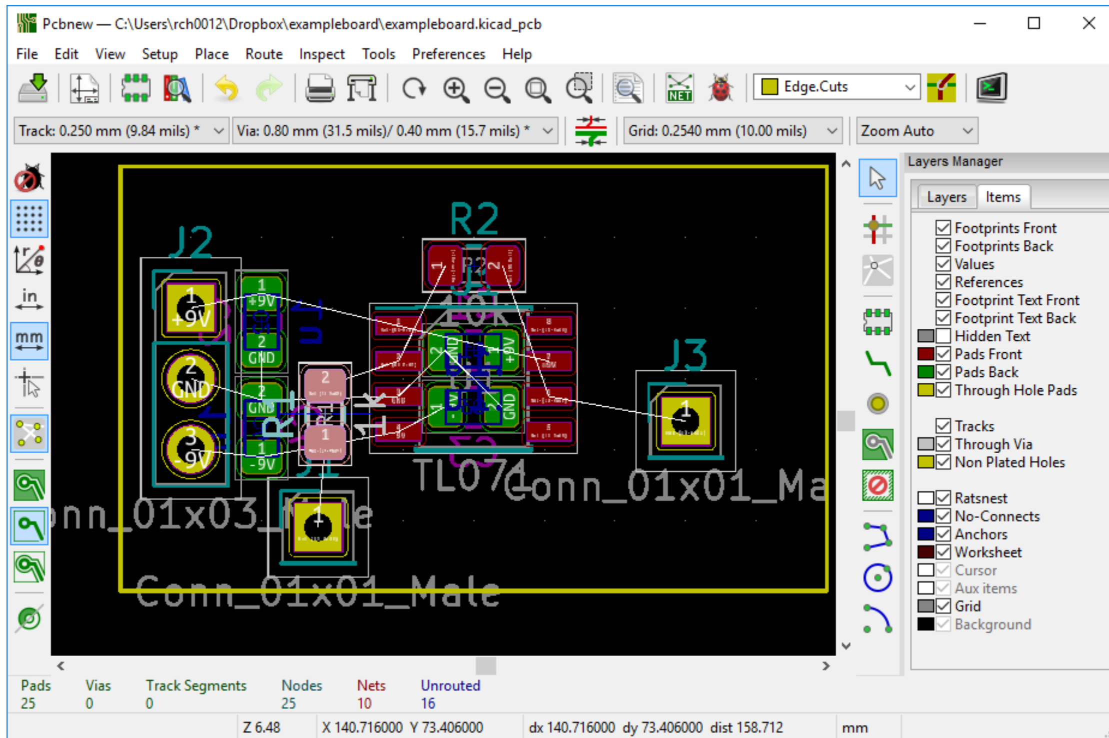

Figure B.17: The components moved to their desired locations, but unrouted

M picks up a part to be moved

R rotates a part 90 degrees at a time

F flips the part from the front of the board to the back or vice versa

E edits a component (change footprint, edit text, change position, etc.)

Middle mouse button and scroll wheel zooms and pans around

An example of a finished component placement is shown in Fig. B.17. At the bottom of this screen it lists the number of unrouted nets (16 in this example). In order to complete the electrical layout of the board, this number needs to reach 0. We will achieve this through a combination of placing traces (copper ”wires” on a single layer), vias (holes drilled into the PCB and then plated to provide electrical connection between layers), and filled zones (areas of copper that provide a continuous connection to a net).

To begin forming electrical connections between components, we will place filled zones on different layers. These zones will be connected to each of our power nets (9V, -9V, and GND) to provide a distributed connection throughout the board. Within PCBnew, click on the

”Add filled zone” botton on the right toolbar, and then click within the PCB. A window similar to Fig. B.18 should appear. For this PCB, we will first add a copper zone to the first inner layer that is a ground zone. Then, we will do the same with the bottom layer. Finally, we will add our +9V and -9V zones to the second inner layer. Select the appropriate net and layer from this screen and the menu options shown for each zone that is added. By drawing in a similar manner to the PCB border creation, the zone will be created. Once the first corner has been reached again, right click the corner and select “Close Zone Outline” to finish the drawing. A zone should appear as shown in Fig. B.19. Repeat this process for the bottom layer ground zone (shown in Fig. B.20). This will provide mostly automatic connection to ground for any parts on the bottom of the board without having traces for each one.

For the power planes in the second layer, start the drawing for the 9V zone close to where the 9V connection comes into the board. The zone will surround this connection as well as extend to wherever else we need 9V power (in this case, the upper half of the op amp). The drawing for the -9V zone will surround the bottom connection as well as the lower half of the op amp. The resulting zones should look like Fig. B.21. Note that in general there will be multiple chips with either single or dual power requirements. A ”interlocking finger” design for interleaving the power planes on the second layer works especially well if the chips are able to be places in a regular grid (e.g. chip 1 directly above chip 2 vertically, chip 2 above chip 3, chips 4-6 horizontally next to chips 1-3, etc.).

Now we will begin forming the electrical connections between the connectors, chips, passives, and power planes using traces and vias. Throughout this process, you can change the working layer using the small blue arrow on the right side “Layers” tab. This arrow will automatically move from top layer to bottom layer when placing vias. In general, connections should be short, direct, and at 45 degree angles. A 90 turn should be made using two 45 degree turns with some length between them if possible. Of course, the design and placement of the components will necessitate a few longer traces or sharp turns to get through a tight space. Also, there are design rules within PCBnew for setting up trace widths, clearances, etc. For simple boards the default rules are sufficient. For more complex designs (e.g. high frequency,

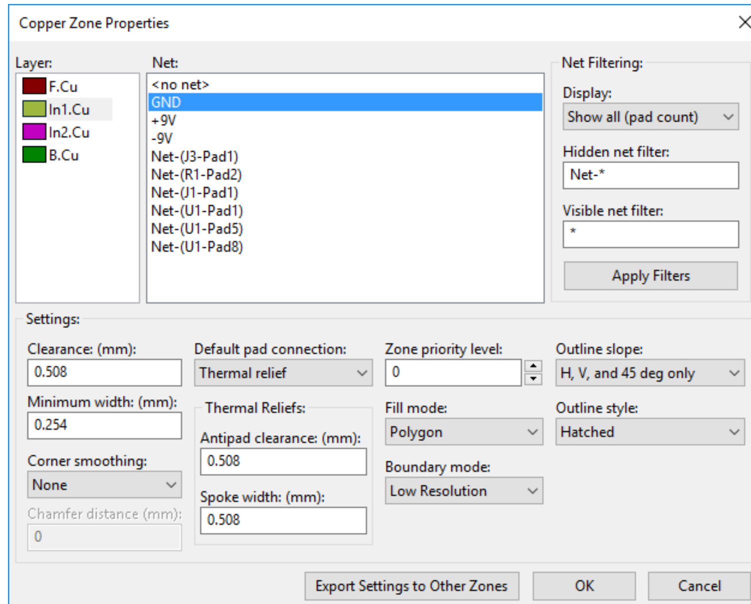

Figure B.18: Selecting a copper zone to begin drawing

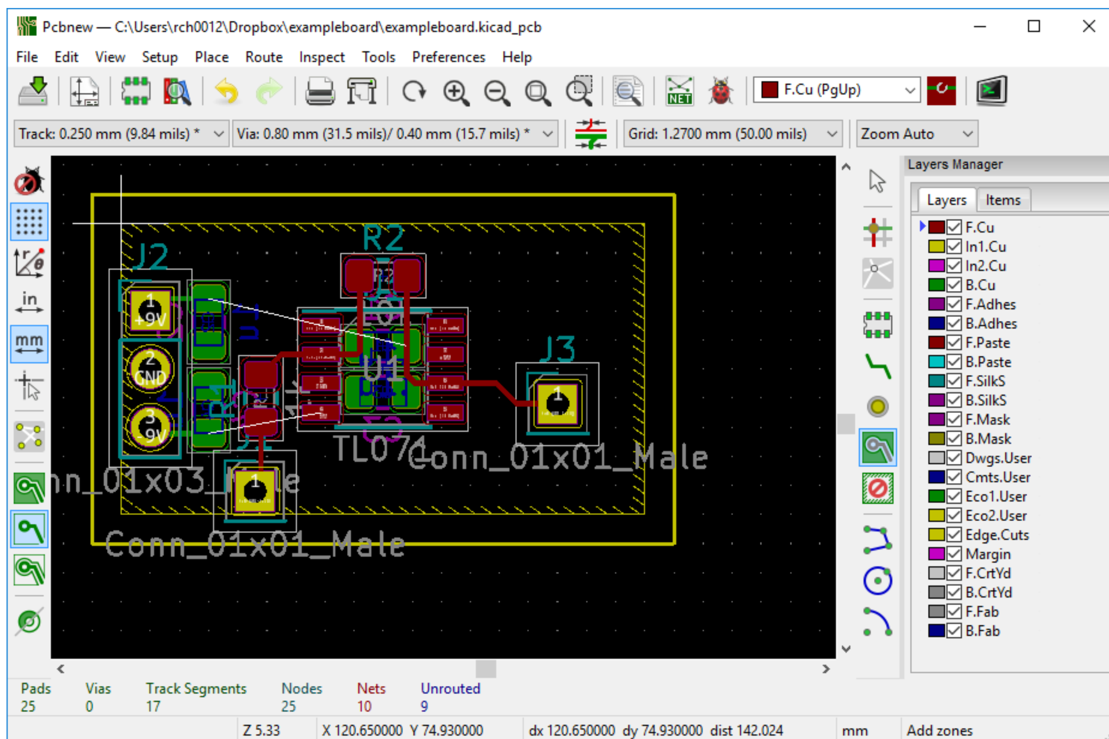

Figure B.19: A copper zone drawn on inner layer 1

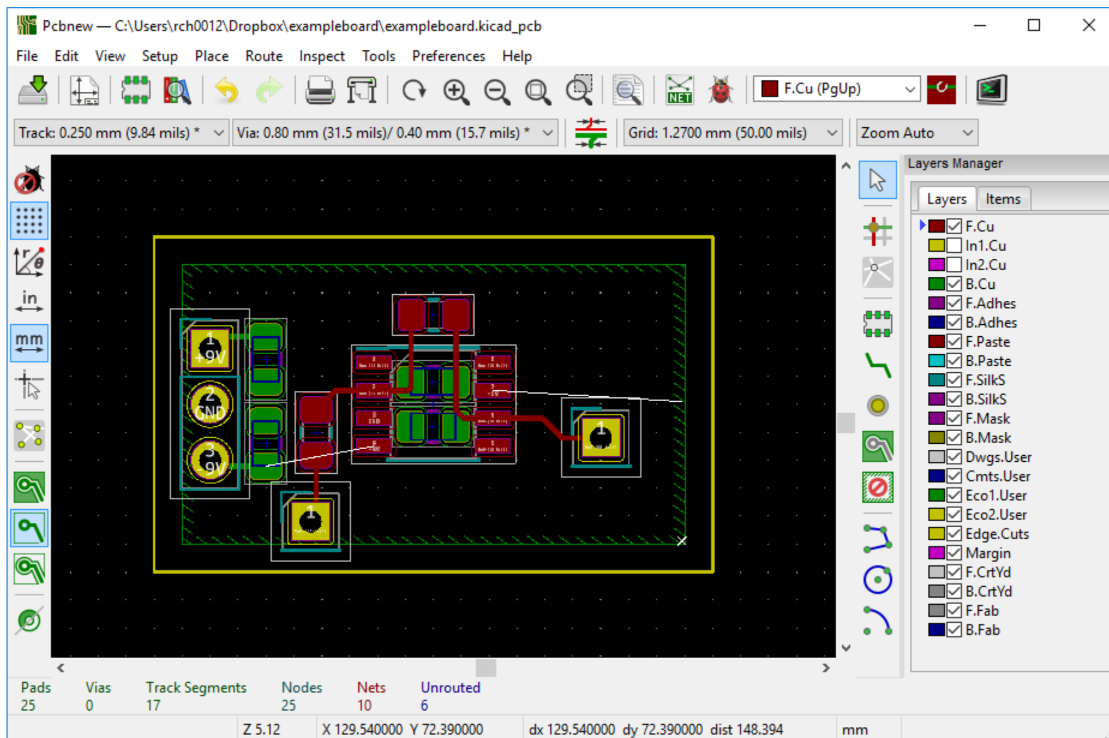

Figure B.20: A copper zone drawn on the bottom layer

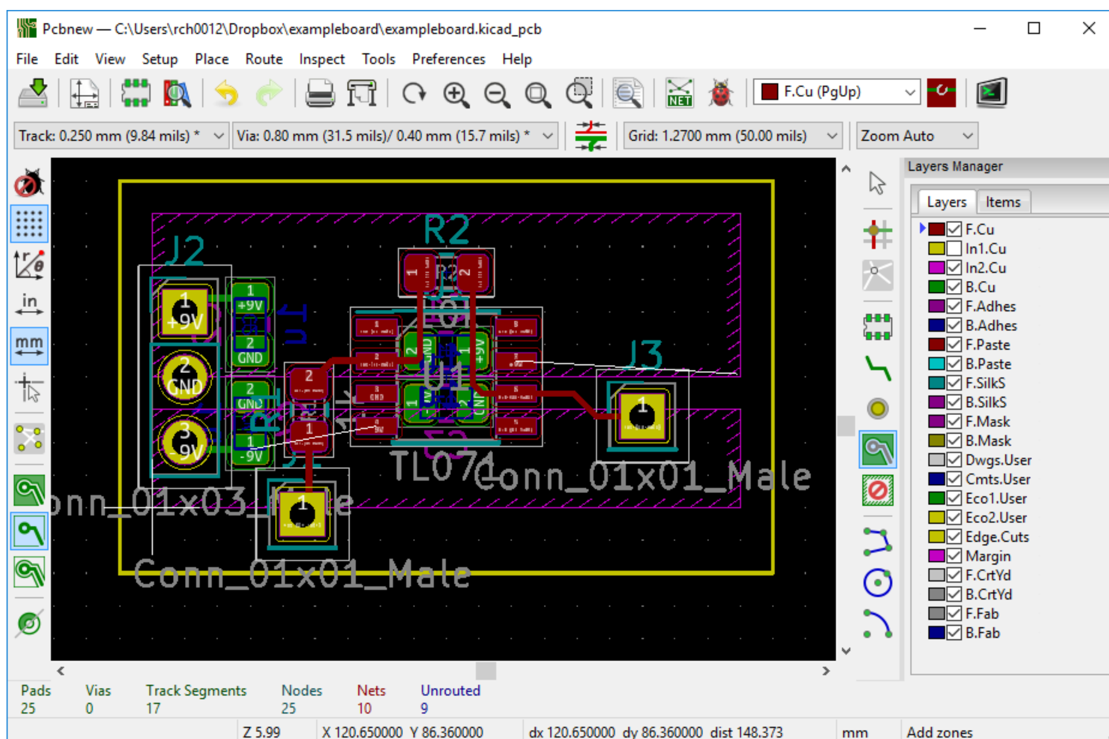

Figure B.21: A copper zone drawn on the second inner layer for the power planes

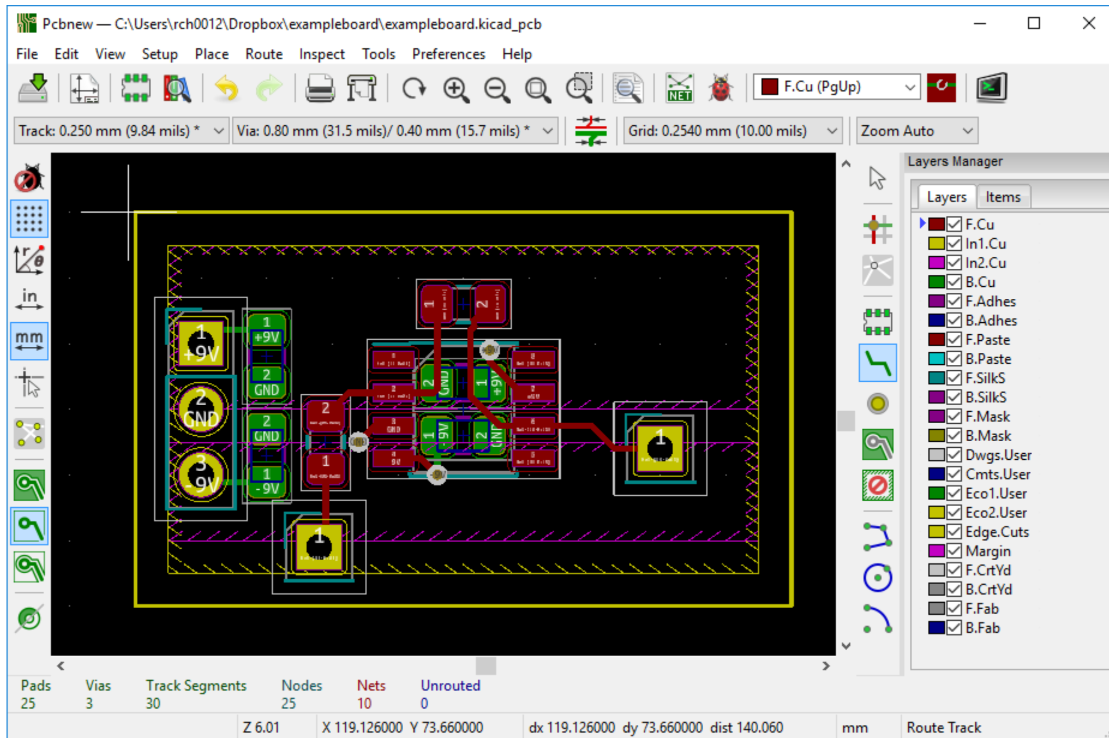

Figure B.22: A completed layout where all components are connected with no unrouted nets

high power) it is worth setting specific rules up to ensure correct PCB operation. A completed electrical layout for this board is shown in Fig. B.22

At this point, the electrical functionality of the PCB is completed. A DRC check can be performed much like in EEschema to make sure everything is connected and nothing is accidentally overlapping. However, there is still some functionality that we can add in the form of silk screen text to aid in populating and using the PCB. Going back to the EEschema schematic and editing the name of the connectors allows us to add these designators on the PCB silk screen for the connector's use. Regenerating the netlist and reading it again in PCBnew (shown in Fig. B.23) will update these names in the PCB. In general, any changes made in EEschema are not updated in PCBnew until the netlist is regenerated and reread. We can then update the layer that the text appears on by right clicking the text and changing the layer from "F. Fab" to "F. SilkS". For each of the parts on the PCB, the aim for placing silk screen designators is to ensure that the footprint and designator are unambiguously related. Usually this merely involves placing the designators directly next to the matching footprint, but in tight layouts this might not be possible. It is helpful also at this point to hide the layers that are

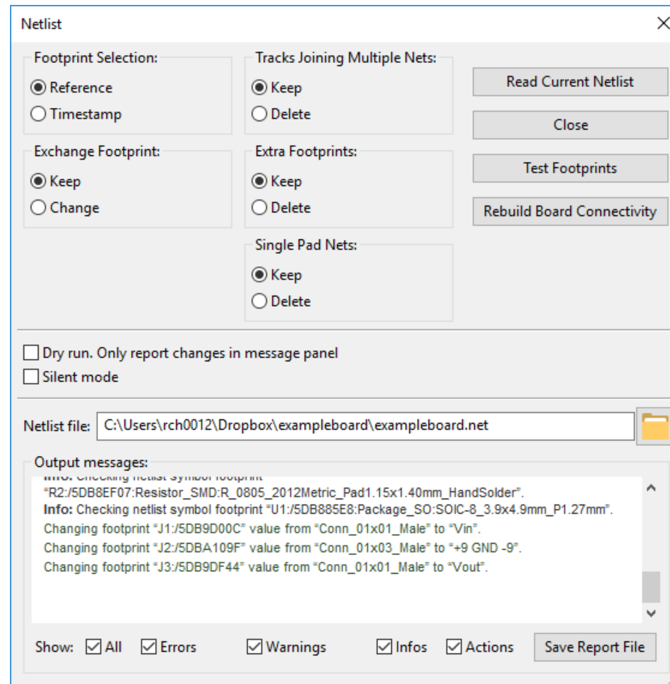

Figure B.23: Changing the names of the connectors on the PCB

not actively being worked on in both the “Layers” and ”Items” tabs. An example of the layed out silk screen is shown in Fig. B.24. By viewing the 3D model of the entire board (View-3D Model in the top menu) the layout of everything can be confirmed to be in the intended location, ash shown in Figs. B.25 and B.26. We can also add text to the board giving the board an identifier or some short notes by clicking on the “T” icon on the right toolbar. The final 3D board is shown in Fig. B.27.

In order to send this design to be manufactured, we will create a set of files called gerbers that give the layout of each layer in a specified format. To do this, “File-Plot” in the top menu will launch a window similar to Fig. B.28. It is recommended that a new folder specifically for these gerbers be created within your project folder. Make sure all of the appropriate layers are checked in the left column and then hit “Plot”. Then, click “Generate Drill Files” to be presented with another screen similar to Fig B.29. The default options on this screen should be sufficient; hit “Generate Drill File” to save this information to the chosen folder.

Once the gerbers have been created, converting the gerber folder into a .zip file will allow easy upload to a PCB manufacturer’s website for checking. These website will often have a preview of the board generated to make sure that everything was read correctly. Once the

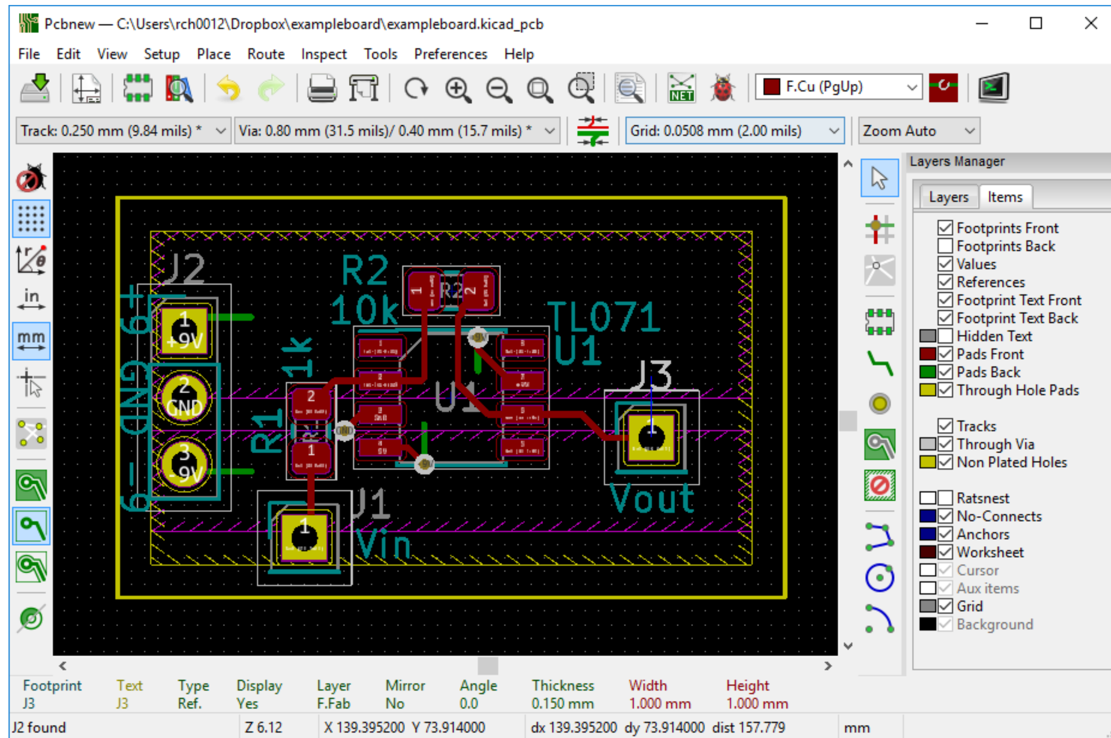

Figure B.24: The front silk screen with designators, references, and values

design is confirmed, order the board and wait 1-2 weeks for your manufactured printed circuit board to arrive. Upon arrival of the boards, visually inspect the board to make sure all traces are complete and all vias are correctly drilled. Now the board can be populated with the parts that were obtained while waiting on the board, and finally the PCB can be used!

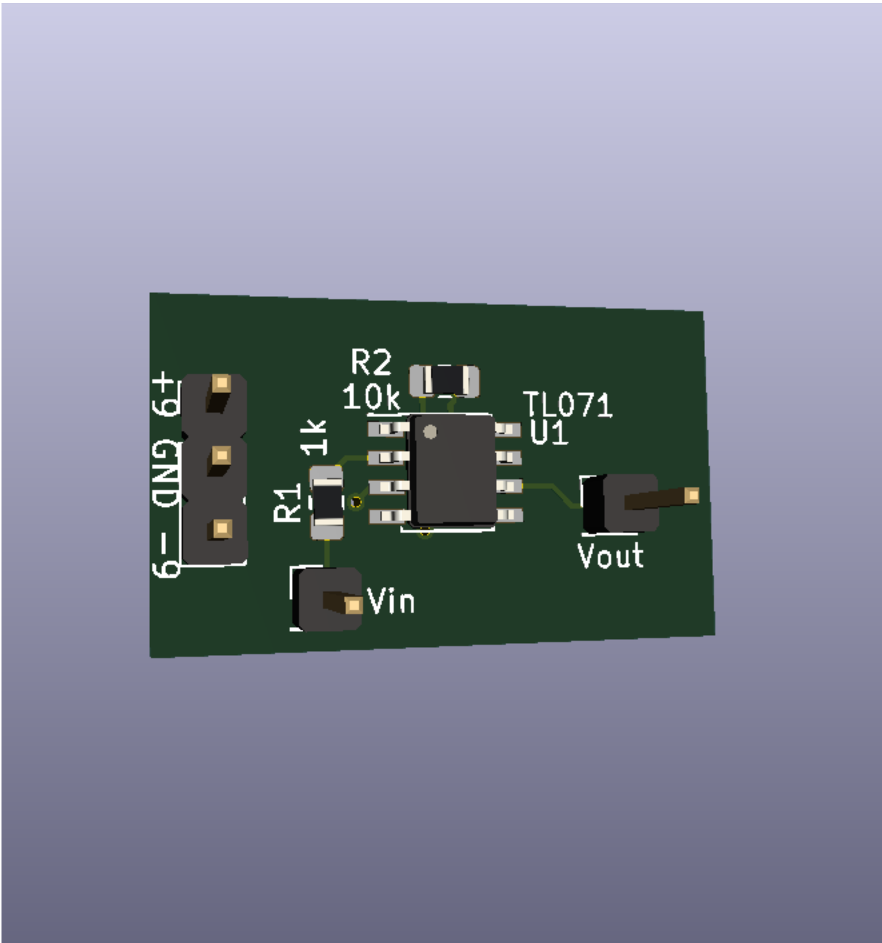

Figure B.25: The front silk screen in 3D

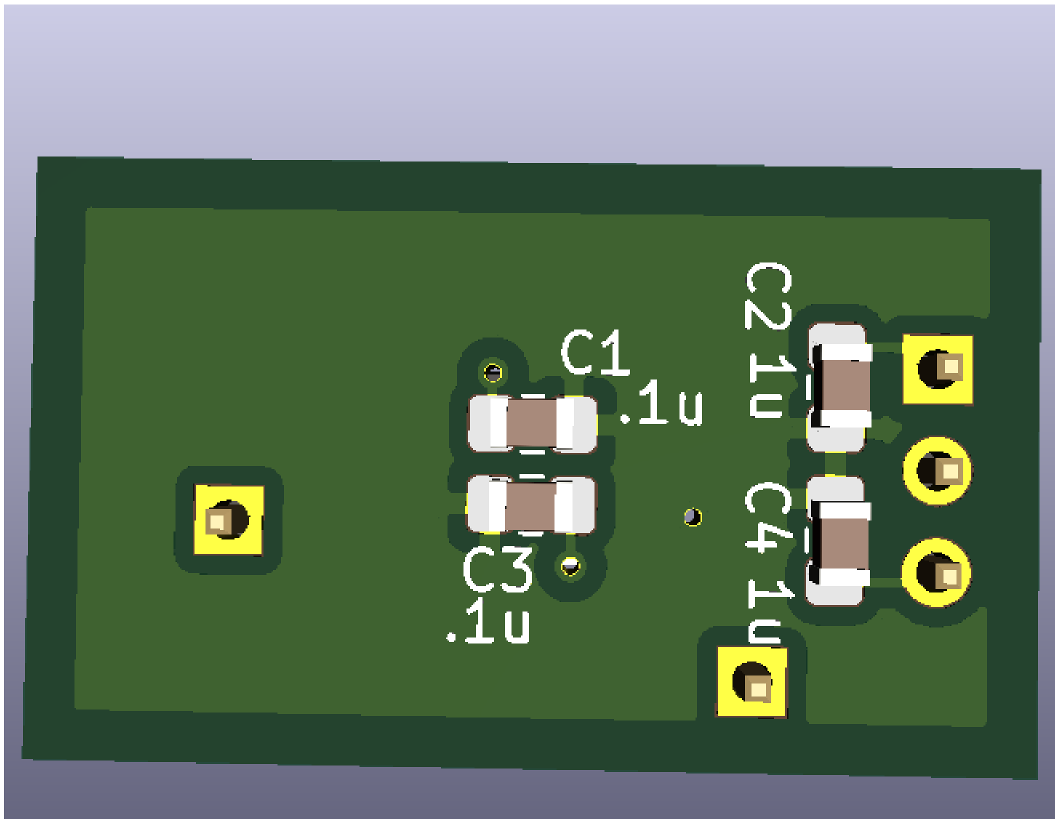

Figure B.26: The back silk screen in 3D

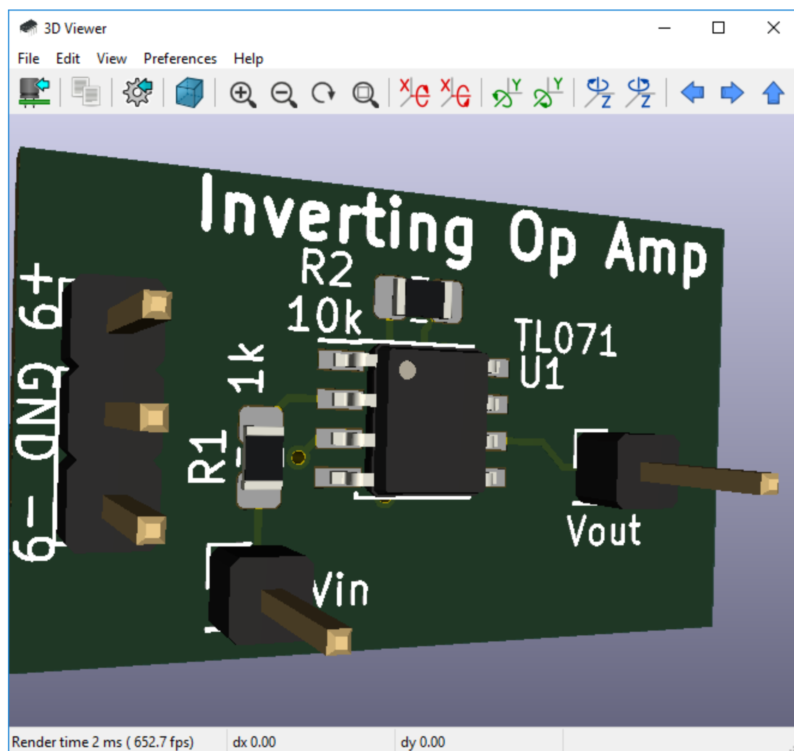

Figure B.27: The completed board in 3D

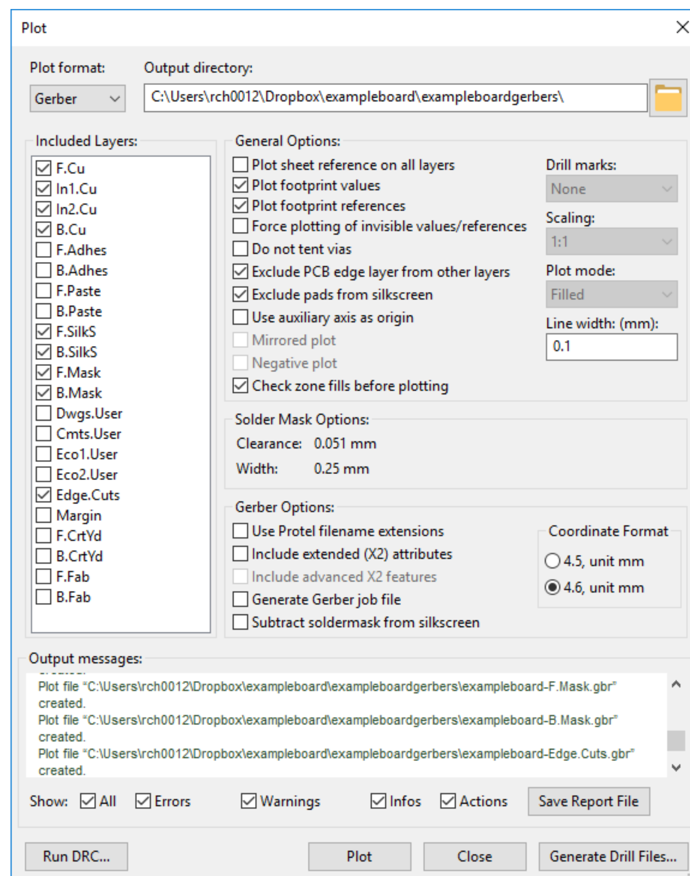

Figure B.28: Plotting each of the layers as a gerber file

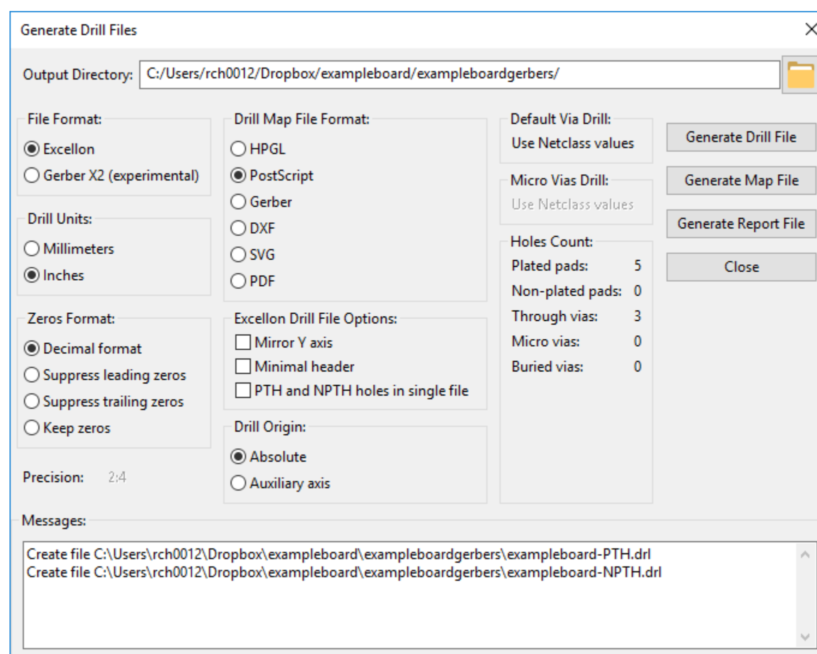

Figure B.29: Plotting the drill locations for the PCB

Appendix C

Code Sections for Various Functions

The code listed here was written to automate various processes, most notably collection and processing of all of the data needed for Dieharder testing. A short description and relevant notes are given for each section of code to aid future use. These were all written in MATLAB unless noted otherwise.

Coverts an array of 0s and 1s into the equivalent 8-bit byte array

```
    function [bytes] = bitstobytes(bits)
%input = (sgnx+1)./2;
%input = (bits+1)./2; % -1 to 1
input = bits; % 0 to 1

bytes = zeros(1,floor(length(input)/8),'uint8');
n = 1;
k=1;

for n = 1:length(bytes)
    bytes(n) = input(k)*128+input(k+1)*64+input(k+2)*32+...
    input(k+3)*16+input(k+4)*8+input(k+5)*4+input(k+6)*2+input(k+7);
    k=k+8;
end
```

Converts an array of bytes into an array of 32-bit integers

```
    function [bytes232] = bytes232(bytes)

%input = (sgnx+1)./2;
```

```
%input = (bits+1)./2; % -1 to 1
input = bytes;

bytes232 = zeros(1,floor(length(input)/4),'uint32');
n = 1;
k=1;
%powers = uint32([2^24 2^16 2^8 2^0])';

for n = 1:length(bytes232)
    %bytes232(n) = typecast([bytes(k+3) bytes(k+2) bytes(k+1)
    %bytes(k)],'uint32'); slower
    bytes232(n) = bytes(k+3)*16777216 + bytes(k+2)*65536 +...
    bytes(k+1)*256 + bytes(k);
    k=k+4;
end
```

Apply the Von Neumann corrector to a byte array

```
    function [vn] = vonneumann(bytes)

% outbits = zeros(1,length(bytes)*2.5,'uint8');
outbits = zeros(1,length(bytes),'uint8');
i=1;
j=0;
for n=1:length(bytes)
    for k=0:2:6
        a=bitand(bitshift(bytes(n),-k),3);
        if(mod(a,3)~=0)
            outbits(i)= outbits(i) + mod(a,2);
            if(j>8)
                j=0;
            i=i+1;
            else
               outbits(i) = bitshift(outbits(i),1);
            j=j+1;
```

```
            end
        end
    end
end
vn = outbits(1:(i-1));
%vn = bitstobytes(outbits);
```

Simulate the jerk equation, sample it, discard MSBs, concatenate the lower bits into a large file for Dieharder

```
    format compact
%tmax = 200;
dt = 1e-1;
%t = 0:dt:tmax;
t = 0;
T = 5; % 1/f, T=5 works
scale = 2.5; %2.5 works
ibit=16;
tstart = cputime;

for i = 1:8

n=0;
k=1;
maxbytes = 268435456;
xbytes = uint8(zeros([1 maxbytes]));
%x = zeros([1 length(t)]);
%x(1) = -1+2*rand(1);

x = -1+2*rand(1);
xd = -1+2*rand(1);
xdd = -1+2*rand(1);

xddd = -0.5*xdd-xd-x+sign(x);

while(k<=(maxbytes))
```

```
xdd = xdd+xddd*dt;
xd = xd + xdd*dt;
x = x + xd*dt;
xddd = -0.5*xdd-xd-x+sign(x);

t = t + dt;
if (t>=T)
    t = 0;
    %abyte = uint8(x*128/scale+128);%8 bit sampling
    abyte = uint16(x*(2^(ibit-1))/scale+(2^(ibit-1)));
    abit = uint8(bitand(abyte,1));
    xbytes(k) = bitshift(xbytes(k),1)+abit;
    n=n+1;
    if(n>=8)
        n=0;
        k=k+1;
    end
end
end

%outfile = ['C:\Users\rch0012\Desktop\matlabdata\data' num2str(i) '.bin'];
outfile = ['data' num2str(i) '.bin'];
wid = fopen(outfile,'w+');
fwrite(wid,xbytes);
fclose(wid)
i
tend = cputime-tstart
end



% bigfile = 'C:\Users\rch0012\Desktop\matlabdata\bigdata_T10.bin';
bigfile = 'bigdata_T5_S2.5_B16.bin';
bid = fopen(bigfile,'w+');
fclose(bid);
```

```
% for i=1:8 %windows
% outfile = ['C:\Users\rch0012\Desktop\matlabdata\data' num2str(i) '.bin'];
%
% s = sprintf('copy /b %s+%s %s',bigfile,outfile,bigfile);
% system(s);
%
% end
s = sprintf('cat data*.bin -> %s',bigfile); %linux
system(s);
```

Perform preliminary RNG analysis within MATLAB to determine if the dataset is obviously biased. Fig. 1 should be a 50/50 even split between 0s and 1s in the data set. If visually the histogram is unequal, the bit stream is probably too biased. Fig. 2 should be a uniform distribution across all 8-bit byte values (0-255). Fig. 3 should look like a guassian distribution (bell curve). Fig. 4 is an FFT of the bit stream (chosen at a shorter length for speed). No prominent peaks should be here. Fig. 5 is a quick image made from the bits in the array. No patterns should be here, but even very biased arrays do not show patterns in this image. Fig. 6 is the autocorrelation of the bitstream, zoomed in on the center. There should be no peaks other than the one at the center.

```
    for n=6:-1:1
%for n=4
    ts = cputime;
figure(n);

subplot(2,3,1);
bitjerk=uint8(bitget(testarray,n)); %pick a bit to take, 1-8
histogram(bitjerk,'BinMethod','integers');
xlabel('bit')
ylabel('frequency')
s=sprintf('0s vs 1s (bit=%d)',n);
title(s)
```

```
subplot(2,3,3);
bytejerk = bitstobytes(bitjerk);
histogram(bytejerk,'BinMethod','integers');
%axis([-.5 255.5 0 inf])
axis tight
xlabel('byte')
ylabel('frequency')
s=sprintf('8-bit concatenation (bit=%d)',n);
title(s)

subplot(2,3,2);
bitsum = zeros(length(bytejerk),1);
for n = 1:length(bytejerk)
bitsum(n) = uint8(sum(bitget(bytejerk(n),1:8)));
end
histogram(bitsum,9);
xlabel('bits set')
ylabel('frequency')
title('Number of bits set per byte')

subplot(2,3,4)
maxfft = min(length(bitjerk),1E6);
myfft = abs(fft(bitjerk(1:maxfft)));
plot(myfft(2:floor(end/2)))
title('FFT')

subplot(2,3,5)
pictest = bitjerk(1:1000000);
pic = reshape(pictest,[1000,1000]);
imshow(pic,[0,1])
title('1000x1000 plot');

subplot(2,3,6)
myauto = xcorr(bitjerk(1:1e6)-0.5);
plot(myauto((1e6-1000):(1e6+1000)))
```

```
axis tight
title('Autocorrelation')

te = cputime;
time = te-ts
end
```

Process a lot of .wav files generated from hardware sampling. This script will extract one bit from each sample and concatenate them together, for as many files or separate bits as desired. It takes 32 1 GB files of 16 bit samples to create one 2 GB .bin file for Dieharder.

```
    %%
%no plots, just file
clear all; close all;
for n=1:3  %bits to choose, 1 = LSB
for i = 1:32 %num of input files
 ts = cputime;

infile = ['C:\filepath\data (' num2str(i) ').wav'];
[testarray,Fs]=audioread(infile,'native');

bitjerk=uint8(bitget(testarray,n)); %pick a bit to take, 1-16
bytejerk = bitstobytes(bitjerk);

outfile = ['C:\filepath\data' num2str(i) '.bin'];
wid = fopen(outfile,'w+');
%fprintf(wid,'type: d\ncount: %d\nnumbit: 8\n',length(bytejerk));
%fwrite(wid,bytejerk);
fwrite(wid,bytejerk);
fclose(wid);

te = cputime;
time = te-ts
end

if(1)
```

```
    fclose('all');
    outfile = 'C:\filepath\data*.bin';
bigfile = ['C:\filepath\bigdataB' num2str(15-n) '.bin'];
% bid = fopen(bigfile,'w+');
% fclose(bid);
s = sprintf('copy /b /y %s %s',outfile,bigfile);

system(s);
end
end
```

Process output text files with results from Dieharder testing. This performs the plotting vs. a uniform distribution previously discussed.

```
    close all;
format compact
figure('DefaultAxesFontSize',8)
for bit = 9:16
filename = "hard_3p125MHz_B" + num2str(bit) + "_results.txt";
fileID = fopen(filename);
A = textscan(fileID, '%s %n %n %n %f %s','Delimiter',...
{'|','\t'},'MultipleDelimsAsOne',1,'CommentStyle',{'#'},...
'HeaderLines',8,'CollectOutput',false);

nans = isnan(A{2});
A{1}(nans)=[];
A{2}(nans)=[];
A{3}(nans)=[];
A{4}(nans)=[];
A{5}(nans)=[];
A{6}(nans)=[];
%{'test_name','ntup','tsamples','psamples','p_value','Assessment'};
uniform = 0:1/114:1;
index = 1:114;

pvalues = sort(A{5});
```

```
s = ['Bit = ',num2str(bit)];
subplot(4,2,bit-8)
plot(uniform)
hold;
plot(pvalues)
%annotation('textbox',[.5 .5 0 .5], 'String', 'mytext');
title(s);
xlabel('test number');
ylabel('p value');
legend('Uniform Distribution','Dieharder Results','Location','northwest');

bit
passes = sum(count(A{6},"PASS"))
weaks = sum(count(A{6},"WEAK"))
fails = sum(count(A{6},"FAIL"))
end
```

C Program to obtain bits from a simulated chaotic jerk system

```
 // jerkbits.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h> // For exit() function
#include <string.h>

#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
using namespace std;

int main(int argc, const char **argv)
{
```

```
double timeInterval = 5;
double analogScale = 2.5;
double initialConditions[] = { 1,0,1 };
string outputfilename = "";
string usage = "jerkbits -o outputfile -b bit -s analogScale -T timeInterval -b bit -l m
int bit = 16;
long long maxlength = 100;
if (argc == 1)
{
cout << usage;
return 0;
}

else {
int n = 1;
while (n < argc) {

if (argv[n][0] == '-')
switch (argv[n][1])
{
case 'T':
timeInterval = stod(argv[n + 1]);
//cout << "timeInterval " << timeInterval << '\n';
n += 2;
break;
case 'o':
outputfilename = argv[n + 1];
//cout << "outputfilename" << outputfilename << '\n';
n += 2;
break;
case 's':
analogScale = stod(argv[n + 1]);
n += 2;
break;
case 'b':
```

```
bit = stoi(argv[n + 1]);
n += 2;
break;
case 'l':
maxlength = stoi(argv[n + 1]);
n += 2;
break;

case 'i':
initialConditions[0] = stod(argv[n + 1]);
initialConditions[1] = stod(argv[n + 2]);
initialConditions[2] = stod(argv[n + 3]);
n += 4;
break;
case 'h':
cout << usage;
return 0;
break;
default:
break;
}
else
{
n += 1;
}
}
}

cout << "timeInterval " << timeInterval << '\n';
cout << "outputfilename " << outputfilename << '\n';
cout << "analogScale " << analogScale << '\n';
cout << "bit " << bit << endl;
cout << "maxlength " << maxlength << endl;
cout << "initialConditions x, xd, xdd = " << initialConditions[0] << ", "...
<< initialConditions[1] << ", " << initialConditions[2] << '\n';
```

```
ofstream(fptr2);
fptr2.open(outputfilename, ios::binary);

char *outbyteptr;

double x = initialConditions[0], xd = initialConditions[1],...
xdd = initialConditions[2], xddd;
double dt = 0.1;
double t = 0;
xddd = (x > 0) - (x < 0) - 0.5*xdd - xd - x;
unsigned int abyte = 0;
char abit = 0;
char samplebyte = 0;
outbyteptr = &samplebyte;
int n = 0;
int k = 0;
cout << '\n';
while (n < maxlength)
{

xdd = xdd + xddd*dt;
xd = xd + xdd*dt;
x = x + xd*dt;
xddd = -0.5*xdd - xd - x + (x > 0) - (x < 0);
if (abs(x) > 5)
{
cout << "\n|X|> 5, system blowing up, breaking";
return 0;
}
t += dt;
if (t >= timeInterval) {
abyte = (x*(pow(2,bit-1)/analogScale) + pow(2,(bit - 1)));
//cout << abyte<<endl;
abit = abyte & 1;
```

```
//cout << to_string(abit)<<endl;
samplebyte = (samplebyte << 1) + abit;
k += 1;
t = 0;
if (k >= 8) {
k = 0;
//cout << samplebyte << endl;
fptr2.write(outbyteptr, 1);
//outputbuffer.write(outbyteptr,1);
samplebyte = 0;
n = n + 1;
}

}
}

//outputbuffer.flush();
//outputbuffer.close();

    return 0;
}
```

C Program to perform the Von Neumann corrector on a given binary file.

```
#include <stdio.h>
#include <stdlib.h> // For exit() function
#include <string.h>

#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
using namespace std;

int main(int argc, const char **argv)
{
```

```
    FILE *fptr;

FILE *fptr2;

string filename;
if (argc == 1) {
cout << "Enter filename: ";
 //filename = "Record1.raw";

cin >> filename;

}
else {
  filename = argv[1];
}

char originalfile[100];
strcpy(originalfile,filename.c_str());

if ((fptr = fopen(originalfile, "rb")) == NULL)
    {
        printf("Error opening file!");
getchar();
        // Program exits if file pointer returns NULL.
}

else {

string outfile = filename.append(".vnbin");


//fptr2 = fopen(outfile,"w");

ofstream(fptr2);
fptr2.open(outfile, ios::binary);
```

```
int outcount = 0;
unsigned char inbyte =0;
 char outbyte=0;
int incount = 0;
unsigned char bits = 0;
unsigned char abyte = fgetc(fptr);
 char *outbyteptr;
outbyteptr = &outbyte;
//printf("\n%d\n",abyte);
long long n=0;
char bit = 0;
cout << "M bits: "<<setw(4)<<"    ";
while(!feof(fptr)){

bits = (abyte & 3);
if (!((bits % 3 ) == 0) ) {
outbyte <<= 1;
bit = (bits & 1);
outbyte = outbyte + bit;
outcount++;
//printf("%d",bit);
if(outcount == 8) {

fptr2.write(outbyteptr, 1);
outcount = 0;
outbyte = 0;
}
}
else{
// printf("_");
}
abyte >>= 2;
incount++;
if(incount == 4){
```

```
abyte = fgetc(fptr);
//printf("\n%d\n",abyte);
incount = 0;
}
n++;
if ((n%10000000) == 0){
cout << "\b\b\b\b";
cout << std::setw(4) << n / 1000000 ;
}

}

fclose(fptr);
//fclose(fptr2);
fptr2.close();
printf("\nDone!");
//getchar();
    return 0;
}
}
```

C Program to perform the XOR operation between two files.

```
// xor2files.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h> // For exit() function
#include <string.h>
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
using namespace std;
```

```
int main(int argc, const char **argv)
{
string filename1;
string filename2;
string outfile;
if (argc != 3) {
cout << "xor2files file1 file2 ";
return 0;
}

else {
filename1 = argv[1];
filename2 = argv[2];
outfile = filename1 + ".xorbin";
}
ifstream(fptr1);
ifstream(fptr2);
ofstream(ofptr);
fptr1.open(filename1, ios::binary);
fptr2.open(filename2, ios::binary);
ofptr.open(outfile, ios::binary);
int outcount = 0;
unsigned char inbyte = 0;
char outbyte = 0;
int incount = 0;
char byte1 = 0;
char byte2 = 0;
char *pbyte1;
char *pbyte2;
char *outbyteptr;
outbyteptr = &outbyte;
pbyte1 = &byte1;
pbyte2 = &byte2;
//printf("\n%d\n",abyte);
```

```
long long n = 0;

/*
fptr1.seekg(ios::end);
fptr2.seekg(ios::end);
cout << "file1 length: " << fptr1.tellg()<<"\n";
cout << "file2 length: " << fptr2.tellg()<<"\n";
fptr1.seekg(ios::beg);
fptr2.seekg(ios::beg);
*/

cout << "M bits: " << setw(4) << "    ";
while (!fptr1.eof() && !fptr2.eof()) {

fptr2.read(pbyte2, 1);
fptr1.read(pbyte1, 1);
outbyte = byte1 ^ byte2;
ofptr.write(outbyteptr, 1);
n++;

if ((n % 10000000) == 0) {
cout << "\b\b\b\b";
cout << std::setw(4) << n / 1000000;
}
}
//fclose(fptr2);
fptr2.close();
fptr1.close();
ofptr.close();
printf("\nDone!");
//getchar();
return 0;
}
```

Appendix D

Dieharder Output

Below is a full Dieharder Output for a successful test run of hardware sampling of the jerk circuit. These results were achieved at 3.125 MS/s at 16 bits of resolution.

```
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |           filename             |rands/second|
 file_input_raw|           hard_3p125MHz_B16.bin|  1.92e+07  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.97980759|  PASSED
      diehard_operm5|   0|   1000000|     100|0.65329807|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.84539595|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.16086332|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.52860159|  PASSED
        diehard_opso|   0|   2097152|     100|0.53328270|  PASSED
        diehard_oqso|   0|   2097152|     100|0.96756200|  PASSED
         diehard_dna|   0|   2097152|     100|0.13417533|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.96201967|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.28801364|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.75190252|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.88077548|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.71120484|  PASSED
     diehard_squeeze|   0|    100000|     100|0.16043836|  PASSED
```

```
        diehard_sums|   0|       100|     100|0.04247707|  PASSED
        diehard_runs|   0|    100000|     100|0.10396408|  PASSED
        diehard_runs|   0|    100000|     100|0.40512514|  PASSED
       diehard_craps|   0|    200000|     100|0.39642118|  PASSED
       diehard_craps|   0|    200000|     100|0.99525386|   WEAK
 marsaglia_tsang_gcd|   0|  10000000|     100|0.08673830|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.07824175|  PASSED
         sts_monobit|   1|    100000|     100|0.63321358|  PASSED
            sts_runs|   2|    100000|     100|0.53583272|  PASSED
          sts_serial|   1|    100000|     100|0.08891374|  PASSED
          sts_serial|   2|    100000|     100|0.37654939|  PASSED
          sts_serial|   3|    100000|     100|0.90855291|  PASSED
          sts_serial|   3|    100000|     100|0.91452190|  PASSED
          sts_serial|   4|    100000|     100|0.85668685|  PASSED
          sts_serial|   4|    100000|     100|0.95973988|  PASSED
          sts_serial|   5|    100000|     100|0.87837796|  PASSED
          sts_serial|   5|    100000|     100|0.73277233|  PASSED
          sts_serial|   6|    100000|     100|0.98439096|  PASSED
          sts_serial|   6|    100000|     100|0.44613983|  PASSED
          sts_serial|   7|    100000|     100|0.86798753|  PASSED
          sts_serial|   7|    100000|     100|0.93718911|  PASSED
          sts_serial|   8|    100000|     100|0.09208397|  PASSED
          sts_serial|   8|    100000|     100|0.02299659|  PASSED
          sts_serial|   9|    100000|     100|0.24029532|  PASSED
          sts_serial|   9|    100000|     100|0.87496595|  PASSED
          sts_serial|  10|    100000|     100|0.36604654|  PASSED
          sts_serial|  10|    100000|     100|0.33169004|  PASSED
          sts_serial|  11|    100000|     100|0.51167487|  PASSED
          sts_serial|  11|    100000|     100|0.17995236|  PASSED
          sts_serial|  12|    100000|     100|0.81032808|  PASSED
          sts_serial|  12|    100000|     100|0.77918720|  PASSED
          sts_serial|  13|    100000|     100|0.89269175|  PASSED
          sts_serial|  13|    100000|     100|0.99726511|   WEAK
          sts_serial|  14|    100000|     100|0.92784070|  PASSED
          sts_serial|  14|    100000|     100|0.44744675|  PASSED
```

```
         sts_serial|  15|    100000|     100|0.17764366|  PASSED
         sts_serial|  15|    100000|     100|0.66036433|  PASSED
         sts_serial|  16|    100000|     100|0.10486936|  PASSED
         sts_serial|  16|    100000|     100|0.07998189|  PASSED
        rgb_bitdist|   1|    100000|     100|0.60347623|  PASSED
        rgb_bitdist|   2|    100000|     100|0.78935558|  PASSED
        rgb_bitdist|   3|    100000|     100|0.36950922|  PASSED
        rgb_bitdist|   4|    100000|     100|0.80189917|  PASSED
        rgb_bitdist|   5|    100000|     100|0.10428693|  PASSED
        rgb_bitdist|   6|    100000|     100|0.81796095|  PASSED
        rgb_bitdist|   7|    100000|     100|0.16461538|  PASSED
        rgb_bitdist|   8|    100000|     100|0.25920552|  PASSED
        rgb_bitdist|   9|    100000|     100|0.11588167|  PASSED
        rgb_bitdist|  10|    100000|     100|0.65821942|  PASSED
        rgb_bitdist|  11|    100000|     100|0.65260112|  PASSED
        rgb_bitdist|  12|    100000|     100|0.87896867|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.54505545|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.46994061|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.89764185|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.61806373|  PASSED
    rgb_permutations|   2|    100000|     100|0.55691835|  PASSED
    rgb_permutations|   3|    100000|     100|0.28555688|  PASSED
    rgb_permutations|   4|    100000|     100|0.93668558|  PASSED
    rgb_permutations|   5|    100000|     100|0.19530908|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.36191429|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.85500343|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.86512479|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.99816597|   WEAK
      rgb_lagged_sum|   4|   1000000|     100|0.97052378|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.44287832|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.63599326|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.19474510|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.04743264|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.35582642|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.16939980|  PASSED
```

```
    rgb_lagged_sum|  11|   1000000|     100|0.75887694|  PASSED
    rgb_lagged_sum|  12|   1000000|     100|0.28553662|  PASSED
    rgb_lagged_sum|  13|   1000000|     100|0.43049563|  PASSED
    rgb_lagged_sum|  14|   1000000|     100|0.37601254|  PASSED
    rgb_lagged_sum|  15|   1000000|     100|0.09485090|  PASSED
    rgb_lagged_sum|  16|   1000000|     100|0.07780751|  PASSED
    rgb_lagged_sum|  17|   1000000|     100|0.33815089|  PASSED
    rgb_lagged_sum|  18|   1000000|     100|0.99894682|   WEAK
    rgb_lagged_sum|  19|   1000000|     100|0.70818871|  PASSED
    rgb_lagged_sum|  20|   1000000|     100|0.20643923|  PASSED
    rgb_lagged_sum|  21|   1000000|     100|0.04115607|  PASSED
    rgb_lagged_sum|  22|   1000000|     100|0.15519526|  PASSED
    rgb_lagged_sum|  23|   1000000|     100|0.09292389|  PASSED
    rgb_lagged_sum|  24|   1000000|     100|0.00336061|   WEAK
    rgb_lagged_sum|  25|   1000000|     100|0.83260080|  PASSED
    rgb_lagged_sum|  26|   1000000|     100|0.01675291|  PASSED
    rgb_lagged_sum|  27|   1000000|     100|0.33367606|  PASSED
    rgb_lagged_sum|  28|   1000000|     100|0.69096408|  PASSED
    rgb_lagged_sum|  29|   1000000|     100|0.05660090|  PASSED
    rgb_lagged_sum|  30|   1000000|     100|0.27288913|  PASSED
    rgb_lagged_sum|  31|   1000000|     100|0.56074936|  PASSED
    rgb_lagged_sum|  32|   1000000|     100|0.00793888|  PASSED
   rgb_kstest_test|   0|     10000|    1000|0.39983289|  PASSED
   dab_bytedistrib|   0|  51200000|       1|0.19440268|  PASSED
           dab_dct| 256|     50000|       1|0.62968020|  PASSED
Preparing to run test 207.  ntuple = 0
      dab_filltree|  32|  15000000|       1|0.50680058|  PASSED
      dab_filltree|  32|  15000000|       1|0.39493822|  PASSED
Preparing to run test 208.  ntuple = 0
     dab_filltree2|   0|   5000000|       1|0.12619958|  PASSED
     dab_filltree2|   1|   5000000|       1|0.19962421|  PASSED
Preparing to run test 209.  ntuple = 0
      dab_monobit2|  12|  65000000|       1|0.66036824|  PASSED
```