The Beauty In Chaos

And its applications in Music Generation

Parshv Joshi, Et al.
202301039

Under the guidance of

Prof. Manish Gupta Prof. Sudip Bera

Dhirubhai Ambani Institute of Information Technology, Gandhinagar

As a part of first semester Calculus Project

Acknowledgements

We, as a group, would like to acknowledge the group members individually, along with their contributions, that made this project possible.

Dhruv Jain - 202301272 - For working on the project webpage, integrating the RNG and L-systems with midi-writer and generating music.
Guru Vyas - 202301196 - For his research in Music Theory, and L-systems.
Jas Mehta - 202301432 - For his work in understanding fractals, L-systems and PUFs.
Parshv Joshi - 202301039 - For understanding Lorenz Attractor, drafting the LaTeX document.
Siddharth Rambhia - 202301072 - For implementing Lorenz Attractor and PUFs and creating an RNG model, implementing NIST tests and other applications in python
Yug Savalia - 202301263 - For his work in understanding statistics and validating randomly generated numbers.
Yug Tejani - 202301487 - For his work in understanding Music Theory, and XOR-Arbiter PUFs, as well as the Project Video.

We would also like to thank Prof. Manish Gupta, for guiding us throughout our project journey and Prof. Mukesh Tiwari for introducing us to Chaotic Systems.

Abstract

In this project report, we first explain chaotic systems and chaotic attractors, and explain how chaotic systems can be used as entropy seeds to Physical Unclonable Functions (PUFs) to act as random number generators for providing musical notes. We then implement such a random number generator in Python, using Lorenz Attractor and XOR-Arbiter PUFs and use it to generate random numbers. We then use the randomly generated numbers to choose notes, and then use L systems for abstract rewriting of those notes. Finally, we use a MIDI writer to translate the notes into a midi file and play it on our website.

Contents

Chapter 1

Motivation

Prof. Manish Gupta served as the initial motivator for this project. After the project was announced, he gave us several options of topics that we could choose from. He also advised us to meet Prof. Mukesh Tiwari, for guidance about non-linear systems.

After meeting Prof. Mukesh Tiwari, we had more clarity of the course on which the project was going. We explored chaotic attractors, and their applications in real life. One such application we found, was random number generation.

Turgay Kaya's paper on random number generation using a Chaotic Attractor as the entropy seed for a PUF, resulting in Truly Random Number motivated us to explore this idea further to add more randomness to our model.[12] More specifically, we wanted to experiment with an attractor different from Chua's, as well as use XOR Arbiter PUFs, rather than Ring Oscillator PUFs, and wanted to see the effect it would have. Eventually, we settled on using Lorenz Attractor and generating random numbers this way.

Chapter 2

Historical Development

2.1 Maxwell and the Origin of Chaos

It is widely believed that the initial idea of chaos was developed intuitively by James Clerk Maxwell, who is also known for his work in electromagnetism. In the 19th century, the notion of a clockwork universe was widespread. At that time, Maxwell was one of the first scholars to recognize the importance of "sensitive dependence of initial conditions."

When Maxwell was deriving his famous distribution law for the velocities of gas molecules, he intuitively developed the idea of chaos, by considering the velocities of two particles.

> *Two spheres, moving in [specified] opposite directions with velocities inversely as their masses strike one another. [F]ind the probability of the direction of the velocity after impact, lying between given limits.*
> (James Clerk Maxwell)

Maxwell assumes a system of two spheres, with non-zero velocity, such that their total momentum vanishes. Now, he states that for the spheres to collide, one of the spheres must be in a certain cylinder, which can be visualized taking into account the other sphere. He assumes that because the sphere could initially be positioned anywhere in the cylinder, with uniform probability, thus, any direction of rebound is equally likely.

In fact, there would be certain initial positions where even an infinitesimal change could drastically change the rebound direction of the sphere. This remarkable conclusion was made by Maxwell, even though his further work didn't depend on this. The conclusion managed to serve as a foundation to chaos theory.

Although Maxwell didn't develop this idea further, in his publications, it is clear to see his views on chaos in his non-scientific writings of that time.

> *When the state of things is such that an infinitely small variation of the present state will alter only by an infinitely small quantity the state at some future time, the condition of the system, whether at rest or in motion, is*

said to be stable; but when an infinitely small variation in the present state may bring about a finite difference in the state of the system in finite time, the condition of the system is said to be unstable. It is manifest that the existence of unstable conditions renders impossible the prediction of future events, if our knowledge of the present state is only approximate, and not accurate.

(James Clerk Maxwell, 1873)

His ideas from the speech would later go on to serve as the foundation of the Duffing Equation, which is a dynamical system that exhibits chaotic behavior.[8]

2.2 Henri Poincaré and the Three Body Problem

In 1887, King Oscar II, of Sweden, established a prize for the n-body problem, which was an intriguing problem at the end of 19th century. The prize would be awarded to anyone, who, given a system of mass points, could obtain a formula for the coordinates of the points, as a function of time, and prove that for all values, it converges uniformly.

Poincaré, while trying to solve the three body problem, realized the sensitive dependence of initial conditions of a system with more than two bodies, and published a paper, in which, although didn't directly achieve the objective of the proposal, but still massively contributed to celestial mechanics, by introducing the idea of chaotic behavior. Poincaré also concluded that obtaining a power-series solution to the three body problem would be impossible. The conclusion was wrong, as Sundman obtained a solution for three bodies, in 1912, in the form of an infinite series, and was able to prove that it would converge uniformly for all values. This was later generalized Qiudong Wang, in 1990s, who solved the n-body problem for $n > 3$.[6]

2.3 Jaques Hadamard and Hadamard's Billiard

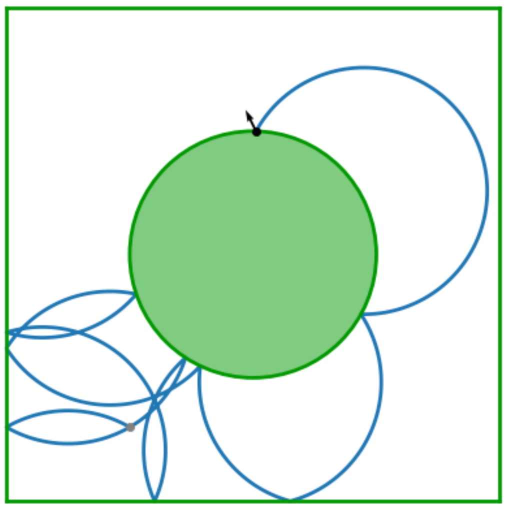
Jaques Hadamard, introduced a dynamical billiard system in 1898.[7] A dynamical billiard system is a system in which a particle alternates between free motion (which could either be a straight line, or accelerated motion, depending on the conditions) and specular reflections. The system became the first system to be proven chaotic.

The paper studied the motion of a particle on the Bolza Surface, a 2-D surface with genus two and negative curvature, and Hadamard was able to prove that all resulting trajectories had a positive Lyapunov exponent. We'll explore Lyapunov exponents later in the report, but a positive lyapunov exponent means that the system is chaotic.

The system is governed by the following Hamiltonian. [4]

$$H(p,q) = \frac{1}{2m} p_i p_j g^{ij}(q) \qquad (2.1)$$

Figure 2.1: Example of a dynamical billiard system, charged particle, with perpendicular magnetic field



Original work of George Datseris Used under CC BY-SA 4.0

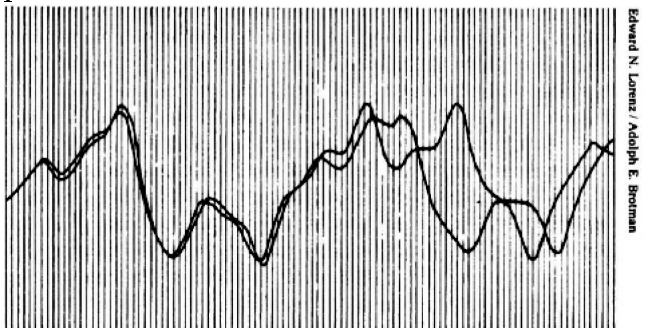
Chapter 3

Edward Lorenz and Modern Chaos

3.1 Sensitive Dependence on Initial Conditions

Edward Lorenz was serving as a meteorologist in MIT, trying to simulate weather using Royal Mcbee. In 1961, he tried to see a sequence for a greater duration. He gave the initial conditions to the machine from a printout, expecting it to mimic the original run. However, the conditions diverged so rapidly from the original run, that it was very hard to see the resemblance after just months of simulation.

Figure 3.1: Two weather patterns diverging from nearly the same starting point



From Lorenz' original printouts

Lorenz realized that small errors proved catastrophic. These systems were later said to have "Sensitive Dependence on Initial Conditions", and were said to be "Chaotic Systems". The realization that Lorenz made realization was surprising as well as frightening. This was the time when meteorology was just starting to develop, with the advent of powerful computers, that could perform series of tiring calculations quickly, like the one that Lorenz was using. The realization was frightening because this meant that weather couldn't be predicted far into the future, something that was still considered a real possibility at that time, with the hope spearheaded by successful predictions of eclipses and tides, even when the dynamics of the Earth, Sun and Moon were fairly complicated.

The average person, seeing that we can predict tides pretty well a few months ahead would say, why can't we do the same thing with the atmosphere, it's just a different fluid system, the laws are about as complicated. But I realized that any physical system that behaved nonperiodi-

cally would be unpredictable.
(Edward Lorenz)

3.2 Origin of the "Butterfly Effect"

It was soon concluded that in the computer's memory, six decimal places were stored. But, in the printout, only three decimal places were printed to save space. When the three decimal places were provided as initial conditions for the simulation, although the difference of one in a thousand was inconsequential, it provided much different results. Lorenz observed that errors and uncertainties multiplied significantly, such that if the starting conditions weren't exact, even with small divergences, the results varied drastically.

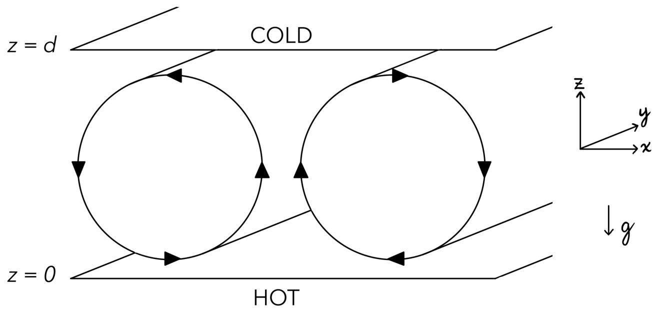
However, as frightening as it was for Lorenz, it was also as interesting for him. Because, this meant that the system wouldn't repeat itself. A non chaotic system, on the other hand, was more likely to repeat itself, because if it came arbitrarily close to a point it had visited before, then, it's path would stay arbitrarily close to the previous path. This would subtract from the unpredictability of weather prediction, and make it boring, Lorenz thought.

During his lectures, he changed "Sensitive Dependence of Initial Conditions", to a term that could be understood more widely. He initially used a seagull flapping wings causing a storm, as an example. Later, he switched to referring this as the "Butterfly Effect", implying how a butterfly flapping its wings could lead to drastic changes in the weather patterns.

3.3 The Lorenz Attractor

After Lorenz' contributions into identifying chaotic systems, he started modelling chaotic systems, to help simulate a lot of things in real life. His work, along with Ellen Fetter and Margaret Hamilton, led to the discovery of the Lorenz Model, for atmospheric convection.

Figure 3.2: Saltzman's equations predicted convection rolls, shown below.



Original Work of Sizmer and Lewis Used under CC BY 4.0 Deed

The model is a set of three ordinary differential equations:

$$\frac{dx}{dt} = \sigma(y - x) \tag{3.1}$$

$$\frac{dy}{dt} = x(\rho - z) - y \tag{3.2}$$

$$\frac{dz}{dt} = xy - \beta z \tag{3.3}$$

Here, σ, ρ and β are proportional to Prandtl number, Rayleigh number and certain physical dimension of the layer, respectively.

These equations form a system of "nonlinear" equations. This is because they cannot be expressed as a system of linear equations because of the terms like xy and $x(\rho - z)$. Nonlinear equations are particularly difficult to solve because, firstly, they exhibit complex and unpredictable behavior, which makes approximating solutions harder. And secondly, the superposition principle cannot be applied to them.

Lorenz stumbled on this system of equations when he was studying a two-dimensional fluid layer, simultaneously warmed from below and cooled from above. Lorenz took a set of differential equations for conventions, and removed all the complexity, leaving just the nonlinearity behind.

The convention equations used were derived by Saltzman in 1962, and were as follows:

$$\frac{\partial}{\partial t}\nabla^2\psi = -\frac{\partial(\psi, \nabla^2\psi)}{\partial(x,z)} + \nu\nabla^4\psi + g\alpha\frac{\partial\theta}{\partial x}, \tag{3.4}$$

$$\frac{\partial}{\partial t}\theta = -\frac{\partial(\psi, \theta)}{\partial(x,z)} + \frac{\Delta T}{H}\frac{\partial\psi}{\partial x} + \kappa\nabla^2\theta. \tag{3.5}$$

Upon expanding ψ and θ in double Fourier Series in x and z, and then simplifying, we obtain Lorenz' Equations.

An attractor, in dynamical systems, is a set of states, towards which a system evolves, for a variety of starting conditions. Typically, the word "attractor" is used to describe asymptomatic behavior of orbits. Putting it simply, if rules that govern a given dynamical system are provided, if a wide variety of initial points appear to move/transform towards a set of points in space, then that set of points would be called an attractor. Lorenz' system also formed an attractor, commonly referred to as the Lorenz Attractor.

Chapter 4

Generating Random Numbers through Chaos

Music can be generated by mapping random notes from the Lorenz Attractor to Piano Notes. However, from our experimentation, we found that because the Lorenz Attractor goes in orbits, it proved difficult to randomly generate notes. We came up with a solution, inspired by Turgay Kaya's Paper on True Random Number Generation, to use Physically Unclonable Functions.

4.1 Introduction to PUFs

With the increase in the use of electronic devices there began a need for verification of the devices as someone could very well clone the device and use it as their own to access the information and technology hidden behind the various things being protected by electronics. Not even small mistakes could be made in this task as a lot of technology can be exploited through the mismanagement of the physical hardware that is attached to it, like credit card cloning, device manipulation etc. Now the first thought that comes to mind is why don't we attach keys to the electronics that are unique to them. But the problem again arises that that key could be copied, so a new solution had to be thought of about this problem. The main thought being, being able to authenticate the IC without relying on a secret key hidden in the IC . As this would remove the attacks which focus on getting the key.

Looking at this a new method was introduced to protect and authenticate ICs, this new method was called as PUFs or physically unclonable function. These functions are built on the factor that every electronic component has some variations while being built such as differences in gate delays, transistor threshold voltages, and other physical characteristics. This factor is used to create a function for each component such that for the same input two different components would give different output even if they have made a clone of the component.

There are two types of PUFs which are most commonly used : the arbiter PUF and ring oscillator PUF. We wanted to use a PUF that is known as xor arbiter, to create a random number generator. And use the numbers which have been generated to create music, hence creating a music generator.

4.2 Basics of PUFs

PUFs were thought of for security but they are not limited to this they can be used in different applications and we are going to be using PUFs for one such application,that is converting the xor arbiter PUF into a random number generator. While this is the main application we will be understanding PUFs as a whole.

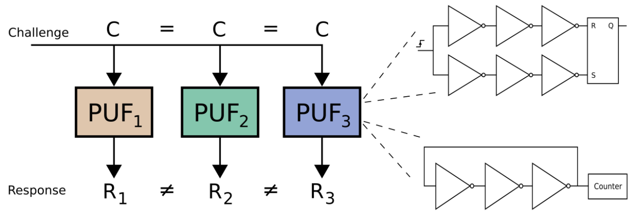
Starting with the basics of PUFs, a small definition that can be used to describe the original use of PUFs is, they are used to authenticate electronic components and are used in cybersecurity as they are unique for each and every particular component.

Let's start with the working of a PUF, the working of the PUF is based on the nanoscale variations which are present during the manufacturing of the device. The PUF takes an input that is know as a challenge and provides an output which is known as the response, but as this is going to be used for authentication it has to be such that a clone device which is created and an random individual using different cryptographic methods can not create the same response for the same challenge. Hence the name it is a function which is physically unclonable or rather it is a function which upon receiving input gives output which can not be cloned using external tools.

This is a slight note that this function differs highly from normal functions such as cos, tan, sin, etc. that would be typically implemented in different softwares or hardware. This difference is what is utilised to create a hard to crack authentication process.

Now to explain in brief what is expected of a PUF we will take an example, so if we assume there to be 3 different PUF devices which are being utilised in different components of the same type, and they are given the same challenge(input), the expectation of a good quality PUF is that each of the devices will provide 3 different responses. Over the years there have been proposals for many different types of PUFs, but not all of them are as good as the other. Hence to differentiate between good and bad PUFs there have been a lot of classification and rankings.

Figure 4.1: Illusttration of Challenge and Response in a PUF



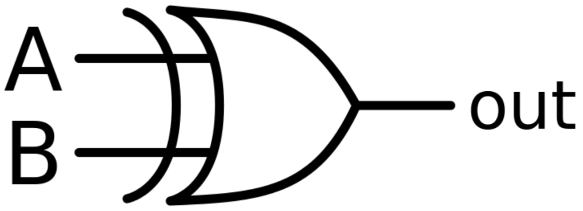
From Dr. Maurits Ortmann's article on PUFs

4.3 XOR Arbiter PUFs

A XOR Arbiter PUF is made of k parallelly connected Aribiter PUFs, but the response of its output is that like of a XOR gate. To understand the xor arbiter PUF let us break it into two parts:

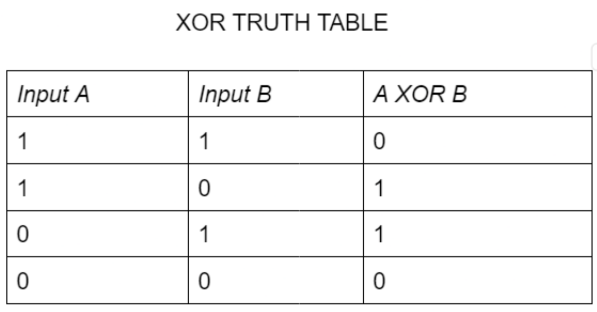
- XOR Gate: XOR GATE IS a digital logic gate which gives positive response/true value (1) when the number of positive inputs is odd. It can be better understood with the help of the truth table below.

Figure 4.2: Truth Table of XOR



Original Work of Heron Used under CC BY-SA 3.0 DEED

Figure 4.3: Truth Table of XOR



XOR TRUTH TABLE

Input A	Input B	A XOR B
1	1	0
1	0	1
0	1	1
0	0	0

Original Work

From the truth table we could see that if the number of positive responses to the xor gate is odd in number it would give us a positive feedback according to the equation:

$$\bar{A}B + A\bar{B} \tag{4.1}$$

- Arbiter PUF: Arbiter PUF is a special implementation of PUF using arbiter circuits. What are arbiter circuits?. Arbiter circuits are electronic circuits most commonly used for tasks related to timing and synchronisation. Arbiter circuits are specifically used in arbiter PUFs to generate unique responses and exploit the race conditions which greatly benefit us. A race condition refers to a scenario where two or more signals within arbiter PUFs reach their destination at the same or similar time. This condition is intentionally induced in the arbiter PUFs so that we can take advantage of process variations in integrated circuits intentionally created during time of manufacture.

4.4 Generating Random Numbers with PUFs and Lorenz Attractors

We take random inputs from the user for the initial conditions of the Lorenz System. We solve the differential equations approximately in python, using Runge-Kutta method. Then, we take alternatively take the x, y, and z coordinates 10000 times each, distributed equally in time, in the first second.[5]

We store them, and take their remainder with 2 to convert them into binary. We then join every 8 elements of the array to form an integer between 0 and 255, and input it into XOR Arbiter Simulator. We then obtain an array of around 800 elements, which we then convert into integers between 0 and 255 to condense it into an array of 100 elements of random numbers between 0 and 255.

Chapter 5

Testing the randomness

Randomness is a probabilistic property; that is, the properties of a random sequence can be characterized and described in terms of probability because the likely outcome of statistical tests, when applied to a truly random sequence can be described in probabilistic terms.

Random number generator tests are designed to assess the quality and randomness of sequences produced by random number generators (RNGs). They are used to test whether the number generated can be described as truly random or not. We have taken some tests to check whether the numbers generated via our software are truly random.

Because there are so many tests for judging whether a sequence is random or not, no specific finite set of tests is deemed "complete." Thus each test or set of tests will have some possibility of error.

For a test, α is the probability that the test will indicate that the sequence is not random when it really is random. This error is often referred to as 'level of significance' of a test and β is the probability that the test will indicate that the sequence is random when it is not.

The cutoff point for the acceptability of the tests is chosen such that the probability of falsely accepting a sequence as random has the smallest possible value. For these tests, P-value is the probability that a perfect random number generator would have produced a sequence less random than the sequence that was tested, given the kind of non-randomness assessed by the test.

If a P-value for a test is determined to be equal to 1, then the sequence appears to have perfect randomness. A P-value of zero indicates that the sequence appears to be completely non-random. If P-value $\geq \alpha$, the sequence appears to be random.If P-value $< \alpha$, then the sequence appears to be non-random. α is the same error mentioned above.[1]

5.1 MonoBit Test

5.1.1 Purpose

To count the number of zeros and ones in a sequence are approximately equal as expected from a truly random sequence.

All subsequent tests depend on the passing of this test.

5.1.2 Parameters Involved

- n - The length of the sequence
- ϵ - The sequence being tested; $\epsilon_1, \epsilon_2, ..., \epsilon_n$.
- S_{obs} - The absolute value of the sum of the x_i (where, $x_i = 2\epsilon - 1 = \pm 1$) in the sequence divided by the square root of the length of the sequence.

5.1.3 Test Description

The reference distribution for the test statistic is half normal.

Steps:-

- The zeros and the ones of the input sequence (ϵ) are converted to values to values of –1 and +1 respectively and are added together to produce $S_n = x_1 + x_2 + x_3 + ... + x_n$, where $x_i = 2\epsilon_i - 1$
- Compute the test statistic $s_{obs} = \frac{|S_n|}{\sqrt{n}}$ WSA.
- Compute P-value = $\text{erfc}(S_{obs}/\sqrt{2})$, where efrc is the complementary error function: $\text{efrc}(z) = \frac{2}{\sqrt{\pi}} \int e^{-u^2} dt$

5.1.4 Test output and conditions

If the computed P-value is < 0.01, then conclude that the sequence is non-random. If P-value is > 0.01, then conclude that the sequence is random.

5.2 Runs Test

5.2.1 Test Purpose

This test is about counting the total number of runs in a sequence of bits. A "run" is a stretch of consecutive identical bits, and it's considered a run of length "k" if it consists of exactly "k" identical bits, with a different bit on either side. The goal of the Runs test is to figure out if the number of runs of ones and zeros of different lengths is what we would expect in a random sequence. Essentially, the test helps us see if the switching between zeros and ones is happening too quickly or too slowly.

5.2.2 Parameters Involved

- n -The length of the bit string.
- ϵ -The sequence being tested; $\epsilon_1, \epsilon_2, ..., \epsilon_n$.
- V_n(obs) -The total number of runs (i.e.,the total number of zero runs in addition to the total number of one-runs) across all n bits.

5.2.3 Test Description

The reference distribution for the test statistic is a x_2 distribution.

Steps:

- Compute the pre-test proportion of ones in the input sequence. $\Pi = \frac{j\epsilon_j}{n}$
- If it can be shown that $\Pi - \frac{1}{2} \geq \tau$, then the runs test need not be performed. $\tau = \frac{2}{\sqrt{n}}$

- Compute the test statistic $V_n(\text{obs}) = \Sigma_{k=1}^{n-1} r(k) + 1$, where $r(k) = 0$ if $\epsilon_k = \epsilon_k + 1$, and $r(k) = 1$, otherwise.
- Compute P-value $= \text{efrc}\left(\dfrac{|V_n(\text{obs}) - 2n\pi(1-\pi)}{2\sqrt{2n}\pi(1-\pi)|}\right)$

5.2.4 Test Output and Conclusions

If the computed P-value is < 0.01, then conclude that the sequence is non-random. If P-value is > 0.01, then, conclude that the sequence is random.

5.3 Test for the longest run of ones

5.3.1 Test Purpose

This test looks at the position of separate sub-matrices within the entire sequence. It checks whether there is a linear relationship among substrings of a fixed length from the original sequence.

5.3.2 Parameters Involved

- n : The length of bit string
- ϵ : The sequence being tested; $\epsilon = \epsilon_1, \epsilon_2, \epsilon_3, ..., \epsilon_n$
- M : The number of rows in each matrix.
- Q : The number of columns in each matrix.
- $X^2(\text{obs})$: A measure of how well the observed number of ranks of various orders match the expected number of ranks under an assumption of randomness.

5.3.3 Test Description

The reference distribution for the test statistic is a x_2 distribution.
Steps:

- Sequentially divide the sequence into MxQ-bit disjoint blocks; there will exist $N = \left[\dfrac{n}{MQ}\right]$. Put the M • Q bit segments into M by Q matrices. Each row of the matrix is filled with successive Q-bit blocks of the original sequence ϵ.
- Determine the binary rank (R_l), of each matrix, each l = 1, 2, 3, ..., N.
- Let F_M = the number of matrices with $R_l = M$ (full rank). F_{M-1} = number of matrices with $R_l = M-1$ (full rank - 1), $N - F_M - F_{M-1}$ = the number of matrices remaining.
- Compute $X^2(\text{obs}) = \dfrac{(F_M - 0.2888N)^2}{0.288N} + \dfrac{F_{M-1} - 0.5776N)^2}{0.5776N} + \dfrac{(N - F_M - F_{M-1} - 0.1336N)^2}{0.1336N}$
- Compute P-value $= e^{-X^2(\text{obs})/2}$

5.3.4 Test output and conclusion

If the computed P-value is < 0.01, then conclude that the sequence is non-random. If the P-value is > 0.01, then conclude that the sequence is random.

5.4 Implementing the tests and obtaining results

The following python code implements the tests (Original Work). In our testing, we found the randomly generated numbers to pass the tests mentioned above.

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
import struct
from pypuf.simulation import XORArbiterPUF
from pypuf.io import random_inputs
import sys

def lorenz(t, xyz, sigma, rho, beta):
    x, y, z = xyz
    dxdt = sigma * (y - x)
    dydt = x * (rho - z) - y
    dzdt = x * y - beta * z
    return [dxdt, dydt, dzdt]

def get_lorenz_solution(sigma, rho, beta, initial_conditions, t_sp
    t_eval = np.linspace(t_span[0], t_span[1], 10000)

    sol = solve_ivp(lorenz, t_span, initial_conditions, args=(sigm

    return sol.y[:, 1:].flatten()

def plot_lorenz_attractor(solution):
    x, y, z = np.split(solution, 3)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.plot(x, y, z, lw=0.5)
    ax.set_xlabel('X-axis')
    ax.set_ylabel('Y-axis')
    ax.set_zlabel('Z-axis')
    ax.set_title('Lorenz Attractor')

    plt.show()

sigma = 10.0
rho = 28.0
beta = 8.0 / 3.0
```

```
t_span = (0, 25)

initial_conditions = [float(input("Enter initial x: ")), float(inp
lorenz_solution = get_lorenz_solution(sigma, rho, beta, initial_co
plot_lorenz_attractor(lorenz_solution)

result_array = [x*10000 % 32 for x in lorenz_solution]

def sum_binary_digits(fraction):
    binary_representation = ''.join(f'{b:08b}' for b in struct.pac
    digit_sum = sum(int(digit) for digit in binary_representation)
    return digit_sum

result_array = [sum_binary_digits(fraction)%2 for fraction in resu

challenge_length = 64
puf = XORArbiterPUF(n=challenge_length, k=2)

chunks = [result_array[i:i+8] for i in range(0, len(result_array),
responses = []

for chunk in chunks:
    binary_element = ''.join(map(str, chunk))
    seed = int(binary_element, 2)
    puf.seed = seed
    challenges = random_inputs(n=challenge_length, N=1, seed=seed)
    response = puf.eval(challenges)
    responses.append(response[0])

result_array = [0 if x == -1 else x for x in result_array]

def condense_binary_array(binary_array):
    condensed_array = []

    for i in range(0, 800, 8):
        binary_chunk = ''.join(map(str, binary_array[i:i+8]))
        decimal_value = int(binary_chunk, 2)
        condensed_array.append(decimal_value)

    return condensed_array

random_numbers = condense_binary_array(result_array)
```

```
print(random_numbers)
```

Chapter 6

Introduction to L Systems and Fractals

Let us start from the beginning with the creation of fractals, then we will give a briefing about what fractals are and how they are being used.

6.1 Discovery of Fractals

The usage of the term fractal itself did not start until the 1980's when it was used by Benoit B. Mandelbrot in his seminal book "The Fractal Geometry of Nature", but fractals have been discovered long before that,

Starting with mathematicians such as Gottfried Leibniz and Karl Weierstrass, Leibniz in the 17th century was pondering the recursive self similarity, which is the basis of fractals, he did make a few mistakes regarding his pondering, he thought that only the straight line was self-similar in that sense. Even after the mistake he did start thinking about fractals. He wrote about "fractional exponents", but commented that "Geometry " did not know of them yet.

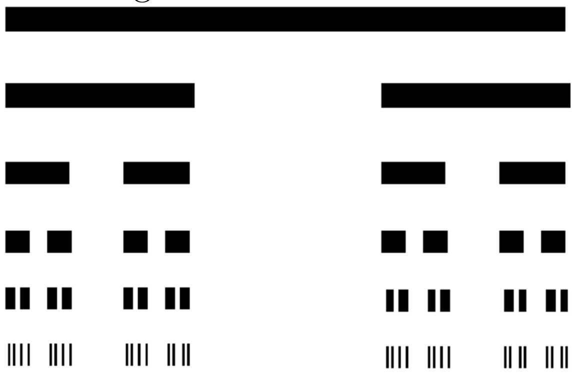
Now due to this being a very unfamiliar term during that time a few mathematicians worked on it but largely remained obscured due to the concept emerging being unknown. This is known as a mathematical "monster" at times. It took a long time but after a couple centuries in 1872 Karl Weierstrass presented the definition of a function without a graph, in today's time it would be considered as a fractal. It can be said to have the non-intuitive property of being continuous everywhere but differentiable nowhere.

After that in 1883, Georg Cantor, published examples of subsets of the real line known as cantor sets, which had unusual properties and are now recognised as fractals.[9]

After this a snowballing started occurring and a lot more development started happening about fractals.

During the end of the century Klein and Henri Poincaré introduced a category of fractals that has come to be called "self-inverse" fractals. Being unsatisfied with Weierstrass's abstract and analytic definition, Helen von

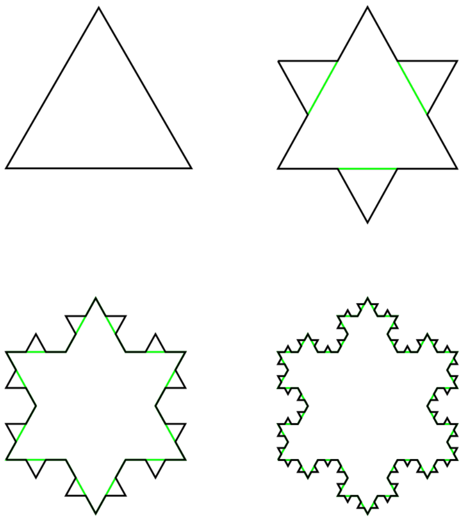
Figure 6.1: Cantor Sets



Original Work of Uta Freiberg

Koch, while building on the ideas of Pointcaré, gave a more geometric definition including a hand drawn image of a similar function, which is now called the Koch snowflake.

Figure 6.2: First four iterations of Koch Snowflake



Original Work of Chas_zzz_brown, Shibboleth

Used under CC BY-SA 3.0

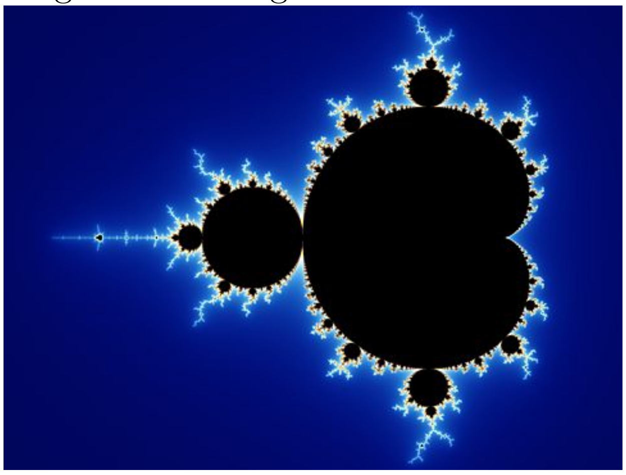
In 1918, two mathematicians, Pierre Fatou and Gaston Julia, while working independently, arrived simultaneously at the results which now can be called as the association of fractal behaviour with mapping complex numbers and iterative functions. Which further led to the idea of attractors and repellors, which are very important in the study of fractals.[11]

In the march of 1918, Felix Hausdorff expanded the definition of "dimension", significantly expanding the evolution of fractals,to allow for sets to have non-integer dimensions. In 1938 Paul Lévy took the idea of self-similar curves further, describing a new fractal curve, the Levy C curve.

With the coming of the age of computers, and the aid of modern computer graphics, the true beauty of fractals could be explored as before this the only way to visualise the fractals were the hand drawn images of the curve which were limited. In 1960, Benoit Mandelbrot started writing about self-similarity in papers such as How long is the coastline of Britain? Statistical Self-Similarity and fractional dimension(a very well known example of fractals in nature).

In 1975, Mandelbrot gave us the term "fractal" and in doing so he solidified hundreds of years of work, thought and mathematical development. And illustrated his mathematical definition with computer constructed visualisation. A lot of the images including his Mandelbrot set were based on recursion hence leading to the popular definition of the word fractal.

Figure 6.3: Image of Mandelbrot Set



Used under CC BY-SA 3.0

6.2 Brief Description of Fractals

A fractal logically is a mathematical geometrical figure that is infinitely complex and it comprises a pattern that repeats forever. Fractals are said to be 'self-similar' as every part of a fractal looks similar to the whole image, in a fractal the smaller parts of the image look like the initial image itself. The more you keep zooming in the more the number of times the shape repeats itself.

Fractals are created by using a feedback loop that feeds a simple process over and over again. Fractals are not a new concept, they are pre-existing in nature to name a few examples, Ferns, Romanesco broccoli etc. fractals are applicable in a lot of fields such as artificial intelligence, machine learning, application for chaotic and nonlinear modelling, etc.[2]

6.3 Introduction to L-Systems

An L system or Lindenmayer system is a string rewriting system, basically it is a set of rules using which, starting from the initial condition/term, we can create a larger or complex term which we desire to use for our work, that is to create fractals.

A small brief on rewriting systems, a rewriting system is basically a set of rules which allow us to manipulate a term to get to the form that is desirable following a set of rules.

Starting the examples with the original L system itself which was developed by a biologist and botanist, Aristid Lindenmayer. This was used for modelling the growth of algae.[10]

The rules:

- A -> A B
- B -> A

Now taking the initial condition as A only we can see that.

- A
- A B
- A B A
- A B A A B
- A B A A B A B A .. and so on

We can observe from this example that the L system is a simple rewriting structure which is recursive in nature and never ending. Using these properties we wanted to create a true random number generator.

A few more examples are:[3]

- The Fractal Tree
- The Cantor Set
- Koch Curve
- Sierpinski Triangle
- Dragon Curve

6.4 Generating Trees using L-Systems

One of the most prominent applications of L-Systems in animations and games is the generation of realistic trees. Since trees exhibit a hierarchical structure, with branches branching into smaller branches and ultimately terminating in leaves. L-Systems provide an elegant solution for simulating this branching pattern. The simple iterative nature of L-Systems allows for the recursive generation of branches, mirroring the organic growth process of real trees. we can also use probabilistic models in these L-Systems to make varying systems that create new and varying trees every iterations

By defining rules that dictate the behavior of each symbol, such as '+' or '-', as rotations or angles, and '[or]' as branching points, L-Systems can accurately represent the branching structure of a tree. The resulting algorithm can be implemented in graphics engines to generate diverse and visually appealing trees that mimic the complexity of their natural counterparts.

6.5 Generating Corals

Similar principles can be applied to the generation of coral structures. Corals, with their intricate shapes and delicate structures, are challenging to model accurately using traditional methods. L-Systems, however, provide a flexible and efficient means of simulating the growth and branching patterns observed in various coral species.

By extending the basic principles of L-Systems, such as incorporating stochasticity in rule application and considering environmental factors, developers can create a diverse range of coral formations. The iterative and recursive nature of L-Systems enables the simulation of coral growth over time, allowing for the generation of visually stunning underwater environments in games and animations.

6.6 Benefits and Challenges

The adoption of L-Systems for tree and coral generation offers several advantages. First and foremost is the ability to create realistic and intricate structures with relatively simple algorithms. L-Systems also provide a high degree of procedural generation, enabling the creation of diverse and dynamic environments without the need for manual design. Also by tweaking the parameter, these systems can be used to create more complicated fractals such as the Sierpinski triangle, the dragon curve and the Koch curve etc.

However, challenges exist in striking the right balance between realism and computational efficiency. Fine-tuning the parameters of L-Systems to ensure that generated structures align with the desired aesthetic can be a complex task. Additionally, optimizing the algorithms for real-time applications, such as games, requires careful consideration of performance constraints.

6.7 Conclusion

In the ever-evolving landscape of computer graphics, the use of L-Systems for generating trees and corals stands out as a testament to the power of mathematical modeling in simulating complex natural phenomena. From lush forests in video games to vibrant coral reefs in animations, L-Systems provide a versatile and effective approach to creating visually stunning and immersive virtual environments. As technology advances, the continued exploration and refinement of L-Systems promise even greater strides in achieving realism and authenticity in the digital representation of nature.

6.8 Python Code Implementation (Original Work)

```
import random
import math
import pygame
import sys

# Define the L-system rules
rules = {
    "X": [
        {"rule": "F[+X][-X]FX", "prob": 0.5},
        {"rule": "F[-X]FX", "prob": 0.05},
        {"rule": "F[+X]FX", "prob": 0.05},
        {"rule": "F[++X][-X]FX", "prob": 0.1},
        {"rule": "F[+X][--X]FX", "prob": 0.1},
        {"rule": "F[+X][-X]FA", "prob": 0.1},
        {"rule": "F[+X][-X]FB", "prob": 0.1},
    ],
    "F": [
        {"rule": "FF", "prob": 0.85},
        {"rule": "FFF", "prob": 0.05},
        {"rule": "F", "prob": 0.1},
    ],
}

# Define drawing rules
draw_rules = {
    "A": {"color": (229, 206, 220), "radius": 2},
    "B": {"color": (252, 161, 125), "radius": 2},
    "F": {"length": 10, "width": 2},
    "+": {"angle": -25},
    "-": {"angle": 25},
    "[": {"save": True},
    "]": {"restore": True},
}

len_val = 4
ang = 25
num_gens = 6

stack = []
pos = [300, 600]
```

```
angle = 0
current_length = 10
current_width = 2
current_color = (255, 255, 255)

def setup():
    global screen
    pygame.init()
    screen = pygame.display.set_mode((800, 800))
    pygame.display.set_caption("L-System")
    screen.fill((28, 28, 28))

def draw():
    global pos, angle, current_length, current_width, current_colo
    for char in generate():
        if char in draw_rules:
            rule = draw_rules[char]
            if "length" in rule:
                draw_line()
            if "angle" in rule:
                rotate(rule["angle"])
            if "save" in rule:
                stack.append((pos.copy(), angle, current_length, c
            if "restore" in rule:
                pos, angle, current_length, current_width, current

    pygame.display.flip()

def generate():
    global word
    return "".join(choose_one(rules.get(char, char)) for char in w

def choose_one(rule_set):
    total_prob = sum(rule["prob"] for rule in rule_set)
    rand_num = random.uniform(0, total_prob)
    cumulative_prob = 0
    for rule in rule_set:
        cumulative_prob += rule["prob"]
        if rand_num < cumulative_prob:
            return rule["rule"]
    return ""

def draw_line():
    global pos
    end_pos = (pos[0] + math.cos(angle) * current_length, pos[1] -
    pygame.draw.line(screen, current_color, pos, end_pos, current_
    pos = end_pos

def rotate(angle_diff):
    global angle
    angle += math.radians(angle_diff)

if _name_ == "_main_":
    word = "X"
    setup()

    clock = pygame.time.Clock()
    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
        draw()
        pygame.display.flip()
        clock.tick(30)  # Adjust the frame rate as needed
    pygame.quit()
    sys.exit()
```

Chapter 7

Relation between L-System and Music Generation

7.1 Introduction

In the realm of music, the union of creativity and algorithms has paved the way for innovative approaches to produce unique and exquisite pieces of music. One of the many methods that has captivate the attention of musicians and researches alike are generation of music through the use of Lindenmayer systems or more commonly know as L-systems.

7.2 Understanding L-System

Though L-systems are mostly related and known in the field of biology where they can be used to simulate the growth of plants they have found an unexpected yet brilliant use in the field music generation. L-systems are parallel generative string originally defined to model a plant development. Starting from the 'axiom' or seed,this seed is used as the starting point for the generation process.The grammar rules are applied in parallel to each element of string for several iterations or generations.With each iteration the string undergoes transformations based on the applied rules, leading to the development of the string into more complex structure.

The grammar rule defines how the symbol in the alphabet are transformed after reach loop is run. The rules are generally expressed in form of production rules, where a symbol or a sequence of symbols are replace by another series/string of alphabets.

Let us discuss about the music generation through L-system in more detail, L system is like a empty canvas for where symbols represent musical data like notes, chords , strings. The rules of our L-system define how these will be converted or evolved over successive iteration. The initial axiom becomes our starting point and then our rules come into play, the axiom or seed is converted according to our rules and then it gets transformed by evolution to a potentially complex and beautiful melody.

At the centre of music generation by L-systems lies our rules, its these rules that govern how our seed can grow and be transformed to a string. For example a rule might define how a seed grows to a harmonious melody or might get converted to a crescendo. It is because of these rules governed by us that allow us to create diverse musical patterns and structures.

Once the L-system has gone through several iterations it has to be transformed to an audible form. That's where the translation/interpretation part comes into play. Each symbol in final sequence is mapped to a corresponding musical note, converting our string or sequence to a glorious piece of musical melody. The interpretation part can involve pitching, auto-tuning, rhythm, dynamics etc.

To introduce variety and more beauty to the piece we have created , we add parameters. These parameters allow fine tuning and help governing elements such as tempo variations ,dynamic shifts or changes in instrumentation. This ensures that the piece we have created aligns with what the creator wanted.

There were plenty of instruments that could be used in the above specified format for generation of harmonic melodies, but the one we liked the most and thought sounded the best was the piano. The piano we have chosen has 14 keys.

We will be using the random generated number by our XOR Arbiter PUF and choose 4-6 notes from the set of 12 notes on a piano. We only choose 4-6 notes from the given set of 12 as studies have shown you get the best melody by choosing a set of 4-6 notes maximum for a piece of music.

Now we will generate a map that uses the selected keys and map them to vectors of strings. The vectors actually consist of random arrangement of the 4-6 notes we have randomly selected, this way we have successfully created our L system. We have a seed and now we can apply rules to it to begin our L-system.

We then start the L system iterations by first selecting any notes as our axiom/seed . We then run iterations 3 times, replacing the notes with their respective rules. The rules will also be random for each iteration. We replace the respective notes with their respective rules and in the end we are left with a vector of notes(like a string at end of a L-system). This vector gets very long but we only need 8 notes for our melody so we copy the first 8 notes to another array and use that in our melody. By repeating these 8 notes again and again we get a music like feel and our melody is created, now how to take it to the next level?

Using chord progressions , We did have some further additions to this code that would include chord progressions to match the melody. We would do that by first determining the key of the melody. The key is the note that the melody is centred around. We would determine this by using occurrences of notes and assign them scores. Once the key has been determined, we would then generate a chord progression using the key of melody. We could stick to some very basic chord progression like the I-V-VI-IV or we could have gone for something much more complicated than that.

Bibliography

[1]

[2] 2013.

[3] 2023.

[4] R Aurich, Martin Sieber, and Frank Steiner. Quantum chaos of the hadamard-gutzwiller model. *Physical Review Letters*, 61(5):483–487, Aug 1988.

[5] Mehdi Ayat, Reza Ebrahimi Atani, and Sattar Mirzakuchaki. On design of puf-based random number generators, Apr 2011.

[6] Florin Diacu. The solution of then-body problem. *The Mathematical Intelligencer*, 18(3):66–70, Jun 1996.

[7] Hadamard. Les surfaces à courbures opposées et leurs lignes géodésiques. *Journal de Mathématiques Pures et Appliquées*, 4:27–74, 1898.

[8] Brian Hunt and James Yorke. *Maxwell on Chaos*.

[9] Mike Johnson. History of fractals, Dec 2022.

[10] Przemysław Prusinkiewicz and Jim Hanan. Other applications of l-systems. *Lecture notes in biomathematics*, page 69–80, Jan 1989.

[11] Greg School. Greg school, Nov 2017.

[12] Seda Arslan Tuncer and Turgay Kaya. True random number generation from bioelectrical and physical signals. *Computational and Mathematical Methods in Medicine*, 2018:1–11, Jul 2018.