Indian Institute of Technology Bombay

Introduction to Digital Systems

March - April 2023

Statistical Analysis of Random Pattern Detection

Kandarp Solanki - 210260026

Agnipratim Nag - 210260005

Guide - Prof. Pradeep Sarin

Abstract

This project is aimed at exploring randomness in nature and verifying how the Central Limit Theorem can be visualised through digital electronics. The idea is to perform a 4-bit pattern match on a randomly generated bit-string of length 100 and track the number of successful matches for each of these bit-strings. After this data has been collected, we plot a graph of "Number of Pattern Matches" v/s "Frequency" (of that number of matches) across our entire dataset, and observe if we obtain a distribution that is approximately Gaussian in the limit of a large sample space.

First, we implement a theoretical version of our idea using the *randint* function and the NumPy library to plot the distribution of pattern occurrences. Here, we are able to observe an interesting structure, and this is what we aim to replicate in the next part of the project.

In the digital implementation, our circuit is broadly divided into three parts. The first part of the circuit generates a pseudorandom stream of bits and passes it on to the next part of the circuit. This bit-stream is what we will check for pattern occurrences. The second part of the circuit consists of a finite state machine that carries out the pattern checking for the required pattern bit-string that we have chosen. The last part is a counter circuit that counts the number of occurrences of each pattern and stores it so that it can be referenced during statistical analysis. Statistical analysis involves the plotting and fitting of the collected data, and comparing the obtained experimental mean with the mathematically predicted mean.

Contents

1 Computer Implementation **3**

1.1 Writing the Python code 3

1.2 Statistical Analysis 4

1.3 Verifying CLT 4

2 Digital Implementation **5**

2.1 Theoretical Foundation 5

2.2 Circuit Design 5

2.2.1 Randomiser 5

2.2.2 Pattern Checker 6

2.2.3 Counter 10

2.3 Statistical Analysis 11

3 Experimental Implementation **13**

4 Connection to Physics and Computer Science **14**

5 Conclusion **17**

6 Acknowledgements **17**

7 Appendix **17**

1 Computer Implementation

1.1 Writing the Python code

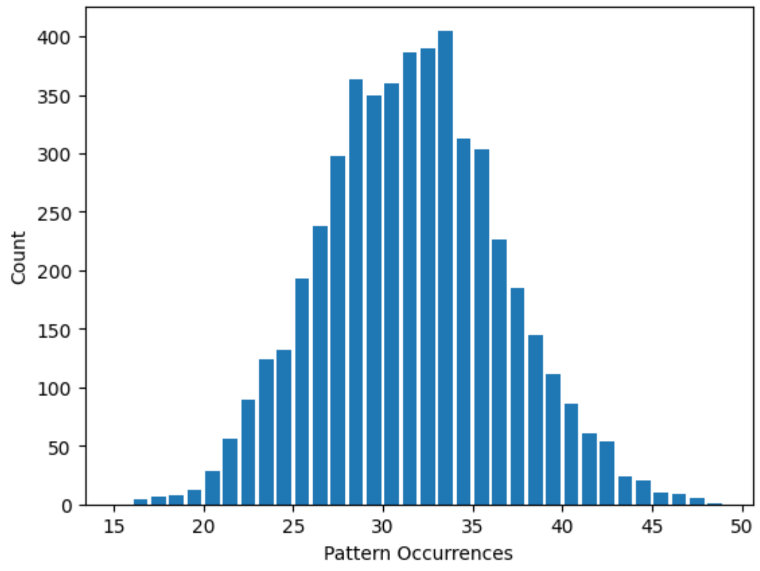
The program for the theoretical implementation is quite trivial. A 1000 bit string is generated using the *randint* function. This is essentially a pseudorandom bit-string as no "purely random" distribution is theoretically possible. After all, we can backtrack the inbuilt circuitry of the Python function as well. After generating multiple such bit-strings, we use a simple for loop to traverse each string and count the number of occurrences of the pattern. An example instance where we pattern match *1101* has been illustrated in the following subsection.

The code has been attached below.

```
import random
import matplotlib.pyplot as plt
def randint():
    return random.randint(0,1)

countlist = []
for i in range(5000):
    num = []
    for i in range(500):
        num.append(randint())

    string = ""
    for i in num:
        string+=str(i)

    counter = 0
    for i in range(2,len(num)):
        if num[i] == 1:
            if num[i-1] == 0:
                if num[i-2] == 1:
                    if num[i-3] == 1:
                        counter += 1
    countlist.append(counter)

plt.hist(countlist, bins=70)
plt.show()
```

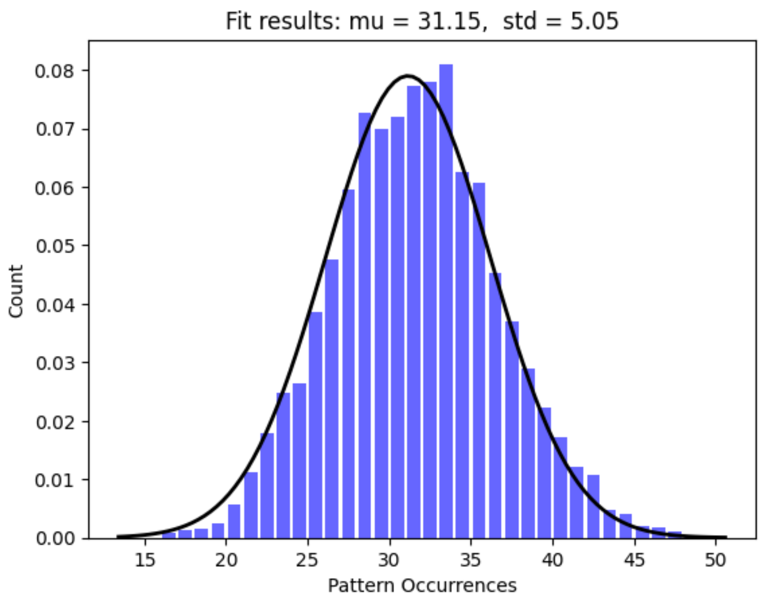
1.2 Statistical Analysis

Once the program has generated the output, we use the *NumPy* library to visualise the obtained data and look for interesting conclusions, if any. What grabs our eye is that on trying out various different patterns, we are able to observe an approximately Gaussian distribution of detection, other than some outliers in the form of skewed peaks or anomaly entries.



1.3 Verifying CLT

This is the most interesting outcome of this exercise. Our randomised exercises is a valid reflection of the well known **Central Limit Theorem** of statistics, which states that in many situations, for identically distributed independent samples, the standardized sample mean tends towards the standard normal distribution even if the original variables themselves are not normally distributed. Now, we move on to replicating the same, on a smaller scale using digital electronics.



2 Digital Implementation

2.1 Theoretical Foundation

For our digital implementation, we first generate a randomised bit-string. This is done using D-registers that are cleared initially. Data inputs are sent into them which update every clock cycle. Using a feedback circuit which consists of XOR gates, we send a logical combination of our bits back into the D register, and this process continues. This generates a pseudo-random pattern which we will use for our analysis.

Note: The pattern generated here is not purely random, because it is after all, the product of a Boolean formula. However, since it capable of generating a bit-string that has 2^8 different possible combinations, we induce a degree of randomness upon it, by arbitrarily choosing one such bit-string to work upon. Additionally, we must also be careful that the pattern we are matching is not biased towards the circuit, which might lead to a skewed result.

After generation of this pattern, we do not attempt to store it as that would be inefficient in terms of space resources. Instead, it is directly sent bit-by-bit into the FSM for pattern checking. Once we accomplish a successful pattern match, the counter circuit increments its value by one for that instance. We repeat this exercises for multiple bit-strings, matching them with the same preset pattern.

Finally, once we have gathered all the data, we plot a graph to examine how often we observe a successful pattern match in each case. On observing the graph, we are able to make further conclusions such as whether our system approximately obeyed CLT, and if it did not, we investigate the reason behind it.

2.2 Circuit Design

The circuit used here can be broadly divided into three parts:

- Randomiser
- Pattern Checker
- Counter

2.2.1 Randomiser

This part of the circuit is a **Linear Feedback Shift Register**. The building blocks here are D-registers and logic gates. Four D-registers are connected sequentially, controlled by a common CLK signal. These are cleared initially, and a feedback circuit is set up using XOR gates as shown in the circuit diagram. This causes a random sequence to be generated with time.

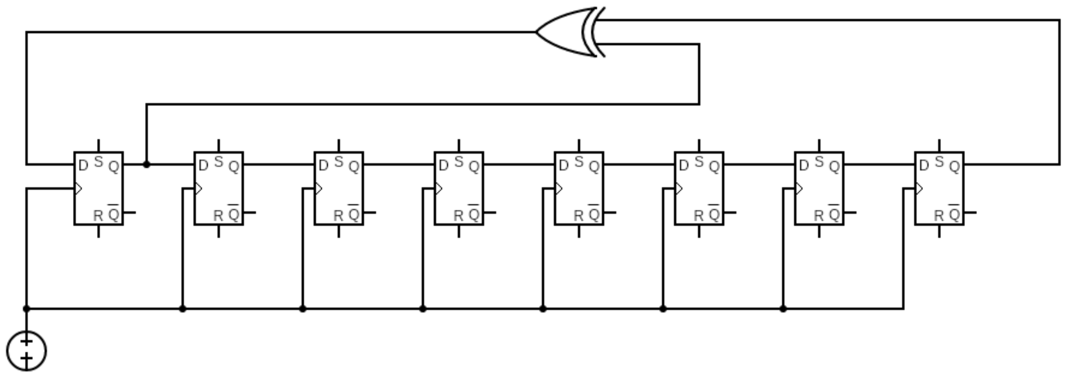


Fig 1. 8-bit Random Number Generator

On generation of the 8-bit stream, we input it into the next part of the circuit by enabling the CLK signal of the FSM, which is the second part of the circuit. The pseudo-randomness of the generated stream is ensured by enabling the CLK signal of the FSM at an arbitrary time. This completes our random pattern generation.

2.2.2 Pattern Checker

To implement this part of our design, we utilise a Finite State Machine. The logic behind this part of the circuit is that our FSM performs a bit-wise check and moves across the reset, intermediate and accept states. Once it reaches the accept state i.e a pattern match has been detected, an LED lights up and a counter makes note of this information.

The logic of the FSM for the detection of pattern **1101** is as follows. The truth table for the FSM is shown subsequently, where A, B and C are the bit indicators of the **current state** and A*, B* and C* are bit indicators of the **next state.**

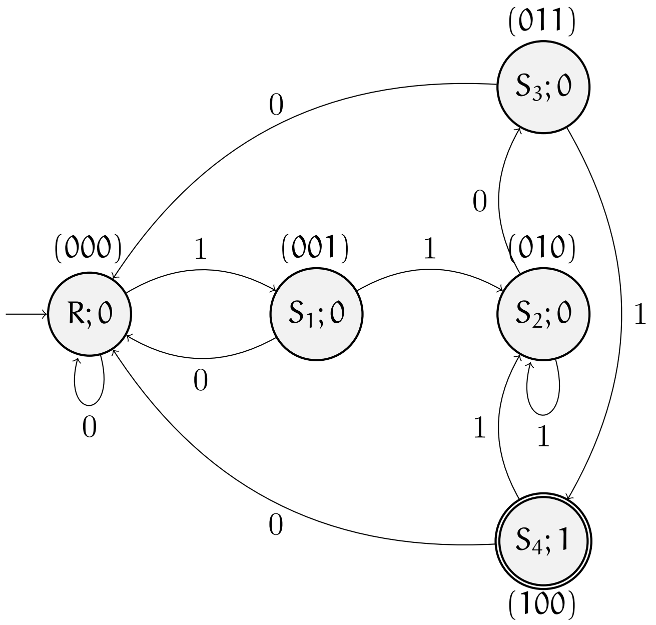


Fig 2. Finite State Machine design for detecting pattern - 1101

Input	A	B	C	A*	B*	C*
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	1	1
0	0	1	1	0	0	0
0	0	0	0	0	0	0
1	0	0	0	0	0	1
1	0	0	1	0	1	0
1	0	1	0	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	1	0

Fig 3. Truth Table for Finite State Machine

Input = 0

BC A	00	01	11	10
0	0	0	0	0
1	0	X	X	X

Input = 1

BC A	00	01	11	10
0	0	0	1	0
1	0	X	X	X

All X's are taken to be 0. The single circled 1 yields the expression $A^* = I \cdot \overline{A} \cdot B \cdot C$

Fig 4. K-Map for A*

Input = 0

A \ BC	00	01	11	10
0	0	0	0	①
1	0	X	X	X

Input = 1

A \ BC	00	01	11	10
0	0	①	0	①
1	①	X	X	X

All X's are taken to be 0.
The circled 1's yield the expression $B^* = I \cdot [\overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot \overline{C}] + \overline{A} \cdot B \cdot \overline{C}$

Fig 5. K-Map for B*

Input = 0

A \ BC	00	01	11	10
0	0	0	0	①
1	0	X	X	X

Input = 1

A \ BC	00	01	11	10
0	①	0	0	0
1	0	X	X	X

All X's are taken to be 0.
The circled 1's yield the expression $C^* = I \cdot \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{I} \cdot \overline{A} \cdot B \cdot \overline{C}$

Fig 6. K-Map for C*

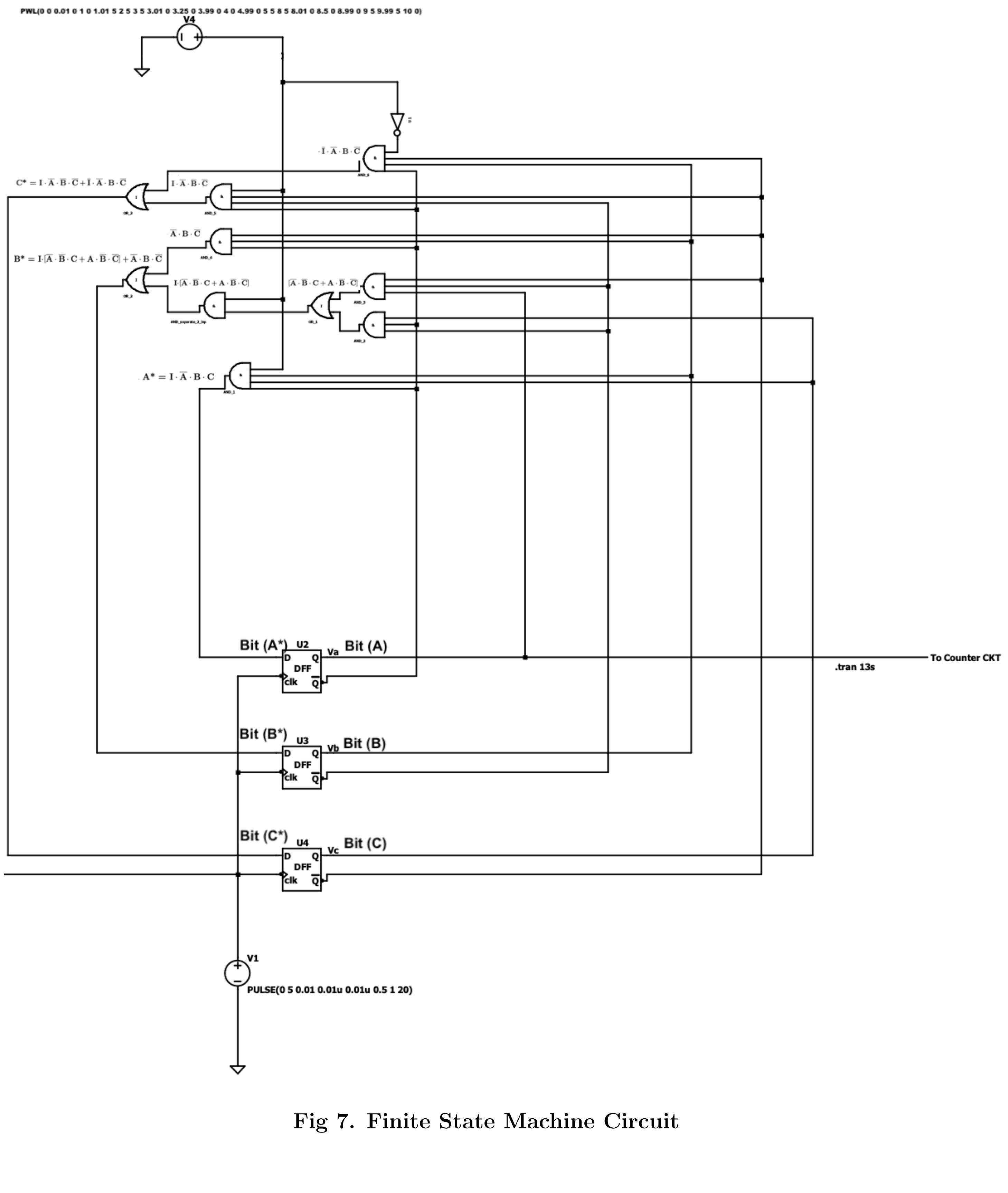

PWL(0 0 0.01 0 1 0 1.01 5 2 5 3 5 3.01 0 3.25 0 3.99 0 4 0 4.99 0 5 5 8 5 8.01 0 8.5 0 8.99 0 9 5 9.99 5 10 0)
V4
U1
$\overline{I} \cdot \overline{A} \cdot B \cdot \overline{C}$
AND_6
$C^* = I \cdot \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{I} \cdot \overline{A} \cdot B \cdot \overline{C}$
$I \cdot \overline{A} \cdot \overline{B} \cdot \overline{C}$
OR_3
AND_5
$\overline{A} \cdot B \cdot \overline{C}$
AND_4
$B^* = I \cdot [\overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot \overline{C}] + \overline{A} \cdot B \cdot \overline{C}$
$I \cdot [\overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot \overline{C}]$
$[\overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot \overline{C}]$
OR_2
AND_seperate_2_inp
OR_1
AND_3
AND_2
$A^* = I \cdot \overline{A} \cdot B \cdot C$
AND_1
Bit (A*)
U2
D
Q
DFF
Clk
$\overline{Q}$
Va
Bit (A)
To Counter CKT
.tran 13s
Bit (B*)
U3
D
Q
DFF
Clk
$\overline{Q}$
Vb
Bit (B)
Bit (C*)
U4
D
Q
DFF
Clk
$\overline{Q}$
Vc
Bit (C)
V1
PULSE(0 5 0.01 0.01u 0.01u 0.5 1 20)

Fig 7. Finite State Machine Circuit

2.2.3 Counter

We will be using a **mod 100 counter** by cascading two **74LS90 BCD (Binary-Coded Decimal)** counters and connecting them to '**BCD to 7-segment Decoder Driver**' for digital display of the count of pattern detection. The digital display for counter is directly available and we need not build it from scratch.

This counter is an integrated circuit that is built on a similar finite state machine model, wherein it uses **JK flip-flops** internally to keep track of the value stored in the counter. The circuit is connected to a 7-segment display, where a particular combination of segments light up to digitally display the stored number. This is capable of counting from 0 to 99. After completing a count from 0-9, the first counter sends a carry bit **(Q_D)** as CLK to the the second counter, and resets to 0. JK flip-flops identify the falling edge of the waveform as a signal to store value and the state transition from 1001 to 0000 makes **(Q_D)** go to 0 through 1 and hence considering a CLK signal.

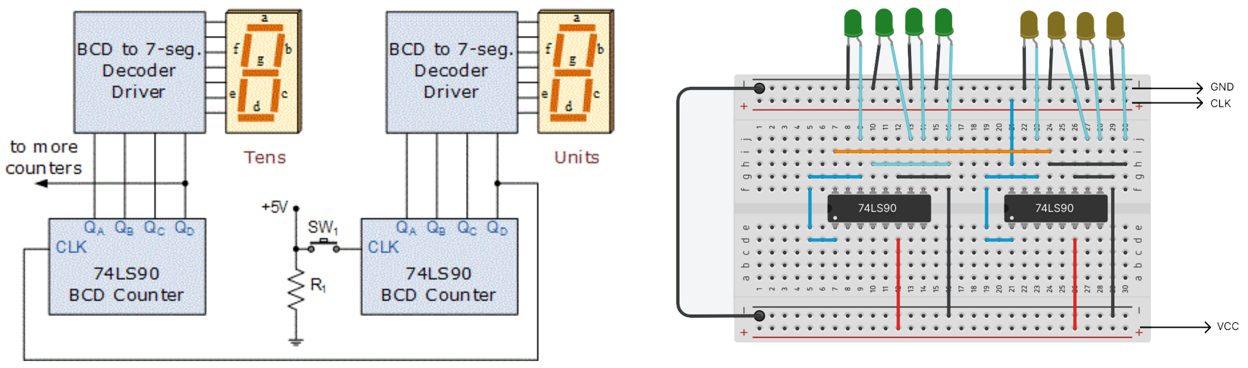


Fig 8. Counter Circuit Design

2.3 Statistical Analysis

Using this circuit, we have been able to collect number of occurrences of the pattern **1101**, for multiple randomly generated bit-strings of length 100. This was done by using a 20 Hz clock signal for time intervals of 5 seconds. The results of our data collection are displayed below. All data analysis has been performed on Python using Pandas, Matplotlib and Seaborn. The code has been attached in the appendix.

	Time	Initial Count	Final Count	Difference
0	5	61	65	4
1	10	65	74	9
2	15	74	77	3
3	20	77	86	9
4	25	86	90	4
...	...	...	...	...
157	790	503	509	6
158	795	509	515	6
159	800	515	520	5
160	805	520	527	7
161	810	527	531	4

162 rows × 4 columns

Fig 9. Data generated by FSM and Randomiser

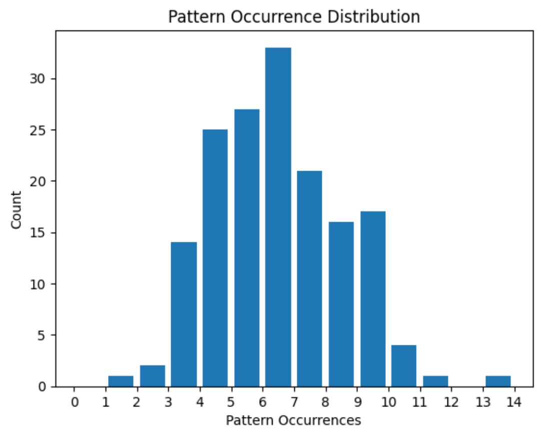


Fig 10. Histogram of Pattern Occurrence Distribution

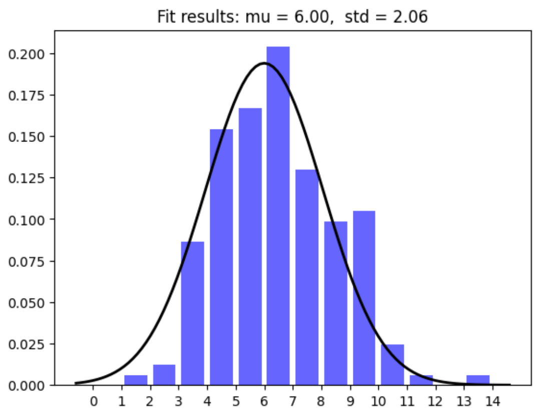


Fig 11. Gaussian Fit

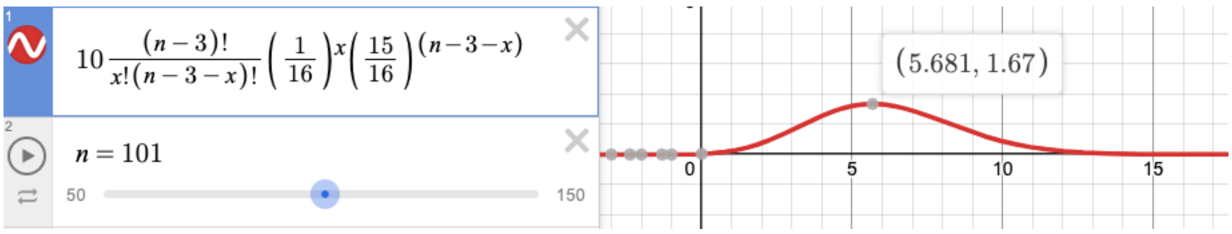
Amazingly enough, the data generated by our digital circuit appears to follow a Gaussian distribution when a large number of data points are considered. The mean of our distribution is **6.00** with a standard deviation of **2.06**.

Mathematical Verification:

On plotting the probability distribution using our knowledge of permutations and combinations, we obtain the formula,

$$\frac{(n-3)!}{x!(n-3-x)!}(\frac{1}{16})^x(\frac{15}{16})^{n-3-x}$$

which on plotting on Desmos, has a mean of **5.68**, which agrees well with our experimental mean.



$$10\frac{(n-3)!}{x!(n-3-x)!}\left(\frac{1}{16}\right)^x\left(\frac{15}{16}\right)^{(n-3-x)}$$

$n = 101$

Fig 12. Mathematically Predicted Distribution

3 Experimental Implementation

The experimental implementation of our project will involve a demonstration that includes the following:

- Showing the randomiser circuit in action. A random bit-stream is generated that is sent as input into the FSM.
- The action of the FSM is demonstrated next wherein it transitions from one state to another based on the input from the randomiser.
- When the FSM reaches its accept state, it adds 1 to the counter circuit which is digitally displayed.
- Few such runs are displayed and the pattern detection is verified.

Video Demonstration

A video demonstration of the project can be found here.

The github repository of the project can be found here.

4 Connection to Physics and Computer Science

The ideas used in this project have important applications in Physics and Computer Science both, in areas such as statistical mechanics and efficient search algorithms.

Statistical Mechanics:

The main motivation behind this project was to show that the **Central Limit Theorem** governs random phenomena around us, in this case - a bit sequence randomly generated by a digital circuit. However, this effect is prevalent throughout physics, and is especially useful in the field of statistical mechanics.

In statistical mechanics, the central limit theorem is used to explain the behavior of large systems of particles that interact with each other. The microscopic behavior of each particle in the system is typically governed by complex and often unknown physical laws. However, the macroscopic behavior of the system can be described by simple statistical laws that are based on the properties of large numbers of particles.

The central limit theorem is particularly useful in statistical mechanics because it allows us to approximate the behavior of large numbers of particles using a normal distribution. This approximation is useful because many statistical laws, such as the **Maxwell-Boltzmann distribution** and the **ideal gas law**, are based on the assumption that the particles in a system are distributed according to a normal distribution.

For example, the Maxwell-Boltzmann distribution describes the distribution of velocities of particles in a gas. This distribution is derived using the assumption that the velocities of the particles are normally distributed. The ideal gas law, which relates the pressure, volume, and temperature of a gas, is also based on the same assumption.

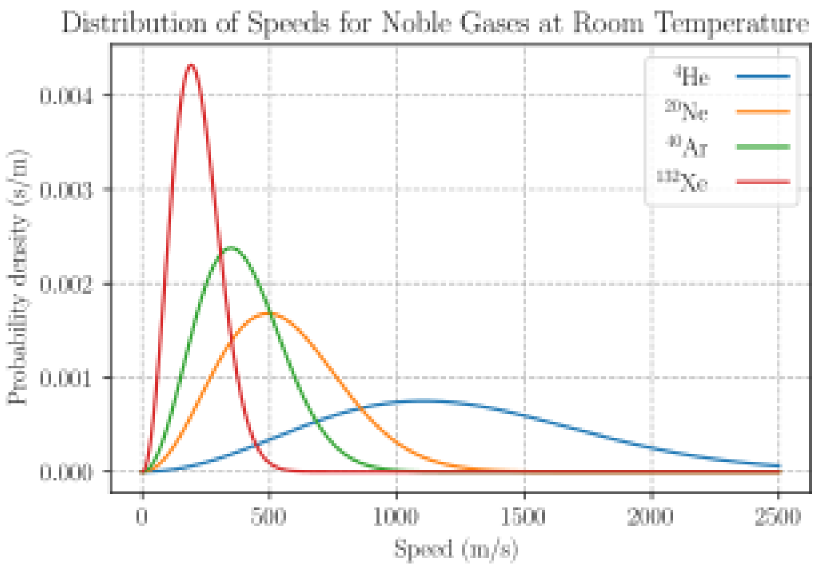


Fig 13. The Maxwell-Boltzmann Distribution

Search Algorithms:

Efficient search algorithms form an important foundation of theoretical computer science, with a number of applications in situations all around us such as looking for records in a massive database. This operation can be very computationally expensive causing systems to fail when tasked with handling large data. Trivial search algorithms include **Linear Search**, which operates in $\mathsf{O}(n)$ time. This can be used to perform pattern matching on a string of length m, which causes the overall time complexity to be $\mathsf{O}(mn)$. Quadratic time complexity cannot be handled by our computers, and hence we require better algorithms. One such solution is the **Knuth–Morris–Pratt algorithm**.

The KMP algorithm is based on the idea of using a partial match table (also known as the failure function) to avoid redundant comparisons when searching for occurrences of a pattern string in a text string. The partial match table stores information about the longest proper prefix of the pattern string that is also a suffix of the current substring being searched in the text string. This information is used to shift the pattern string by a certain number of positions to the right, without missing any potential matches.

Here is a step-by-step explanation of how the KMP algorithm works:

- Construct the partial match table (failure function) for the pattern string. This is done by iterating through the pattern string and finding the length of the longest proper prefix that is also a suffix of each substring of the pattern string. The partial match table is an array that stores this information for each position in the pattern string.
- Start searching for the pattern string in the text string from the beginning. At each position in the text string, compare the current character with the corresponding character in the pattern string. If they match, continue comparing the next characters in both strings. If they do not match, use the partial match table to shift the pattern string to the right by a certain number of positions.
- To determine the number of positions to shift the pattern string, use the value stored in the partial match table for the current position in the pattern string. This value represents the length of the longest proper prefix of the pattern string that is also a suffix of the current substring being searched in the text string. By shifting the pattern string to the right by this value, we can ensure that we do not miss any potential matches.
- Continue searching for the pattern string in the text string using the shifted pattern string until either a match is found or the end of the text string is reached.
- Repeat steps 2 to 4 until all occurrences of the pattern string in the text string have been found.

The KMP algorithm is an efficient and widely used string matching algorithm that can be used in a variety of applications, such as text editors, compilers, and bioinformatics. The **linear time complexity** of the algorithm makes it well-suited for large-scale string matching tasks.

How does our project utilise the KMP algorithm?

We have implemented a similar model of making use of partial match information wherein the automata for our finite state machine reflects the fact that we re-use information obtained at the time of a failed pattern match.

- When the system is in the state ”010”, on receieving input as ”1”, which is the wrong bit for a successful pattern match, we **do not** go back to the reset state. Instead, we loop on that state itself, because the received bit might be the second bit of our pattern 1101 that we are trying to match.
- When the system is in the state ”100” after completing a successful pattern match, it **does not** go to the reset state to check for the next pattern when it receives an input as 1. Instead, it goes to the ”010” state because the incoming 1 might again be part of the start of the next pattern occurrence.

This is how our finite state machine implements a search algorithm that is more efficient than a regular, trivial one.

5 Conclusion

Our project successfully establishes the prevalence of the Central Limit Theorem in random distributions throughout nature, explored here through digital electronics. At first glance, we would not have expected a bit-string governed by a digital circuit to follow such a distribution, but it turns out that it does, which is truly fascinating!

6 Acknowledgements

We would like to extend our deepest gratitude to our professors Prof. Pradeep Sarin and Prof. Maniraj Mahalingam for their constant guidance throughout the duration of the course and giving us the opportunity to undertake this project. This project would also not have been possible without the support of Sir Nitin Pawar, the lab assistants and teaching assistants of EE 224.

7 Appendix

Python code for data analysis is attached below.

```
from google.colab import files
uploaded = files.upload()
df = pd.read_csv('EE - Sheet1.csv', header=0, encoding='unicode_escape')
display(df)
df['Difference'].hist(bins=5, grid=False)

# Add labels and title
plt.xlabel('Pattern Occurrences')
plt.ylabel('Count')
plt.title('Pattern Occurrence Distribution')

# Show plot
plt.show()
from scipy.stats import norm



# Generate some data for this demonstration.
data = df['Difference']

# Fit a normal distribution to the data:
mu, std = norm.fit(data)
```

```
# Plot the histogram.
plt.hist(data, bins=5, density=True, alpha=0.6, color='b')

# Plot the PDF.
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)
p = norm.pdf(x, mu, std)
plt.plot(x, p, 'k', linewidth=2)
title = "Fit results: mu = %.2f,  std = %.2f" % (mu, std)
plt.title(title)

plt.show()
```