# Signal Decoding Techniques

M.H. Khan, *14346011,* J.L.P. Steyn, *16002050*

***Keywords*****—** Root Raised Cosine(RRC), Entropy, Line Code, Efficiency, Block Code, Defferentianl phase shift keying(DBPSK)

## Contents

The authors mentioned are students at the University of Pretoria.

## I. Introduction

The focus of practical 3 is the application of matched filtering and error correction code. The error correction codes methods implemented include Huffman coding, Convolution coding and Linear Block coding. The first burst received required matched filtering to be able to demodulate the signal. The rest of the signals were all dbpsk signals that needed special decoding to extract the data. The second burst was encoded with Huffman coding, the third with a linear block code and the last burst was generated with a convolution encoder.

## II. Matched Filter design

For the implementation of matched filtering, the known pulse shape is correlated with the data that is received. Given that matched filtering is successful the result will be a DBPSK signal which can be demodulated using the algorithm implemented in practical 2.

### A. Methodology

The pulse shape of the signal is know to be a root raised cosine. Hence, if a root raised cosine is correlated with the noisy raw data signal, the signal will be cleaned up to the point where it can be seen to have the characteristics of a DBPSK modulated signal. When implementing the code a common digital modulation python library CommPy contains the necessary functions which allows the user to implement a root raised cosine without having to write out the entire function. However the use of libraries discouraged if the programmer does not understand how the functions work and the way they are interpreted by the compiler.

The root raised cosine was correlated with the data and not convoluted. This is because convolution requires both the data and the root raised cosine to be flipped and time delayed, where correlating achieves the same result with a simpler implementation. The following equation is the impulse response of a root raised cosine[1].

$$h(t) = \begin{cases} \frac{1}{T_s}\left(1+\beta\left(\frac{4}{\pi}-1\right)\right), & t=0 \\ \frac{\beta}{T_s\sqrt{2}}\left[\left(1+\frac{2}{\pi}\right)\sin\left(\frac{\pi}{4\beta}\right)+\left(1-\frac{2}{\pi}\right)\cos\left(\frac{\pi}{4\beta}\right)\right], & t=\pm\frac{T_s}{4\beta} \\ \frac{1}{T_s}\frac{\sin\left[\pi\frac{t}{T_s}(1-\beta)\right]+4\beta\frac{t}{T_s}\cos\left[\pi\frac{t}{T_s}(1+\beta)\right]}{\pi\frac{t}{T_s}\left[1-\left(4\beta\frac{t}{T_s}\right)^2\right]}, & \text{otherwise} \end{cases} \quad (1)$$

where $\beta$ is the roll of factor and Ts is the period.

This piece wise function can be substituted by the commPy library function which is as follows:

$$rrcosfilter(N, \beta, Ts, Fs)$$

N (int) – Length of the filter in samples.
$\beta$ (float) – Roll off factor (Valid values are [0, 1]).
Ts (float) – Period in seconds.

Fs (float) – Sampling Rate in Hz.
Fig. 1 is the result of this function.

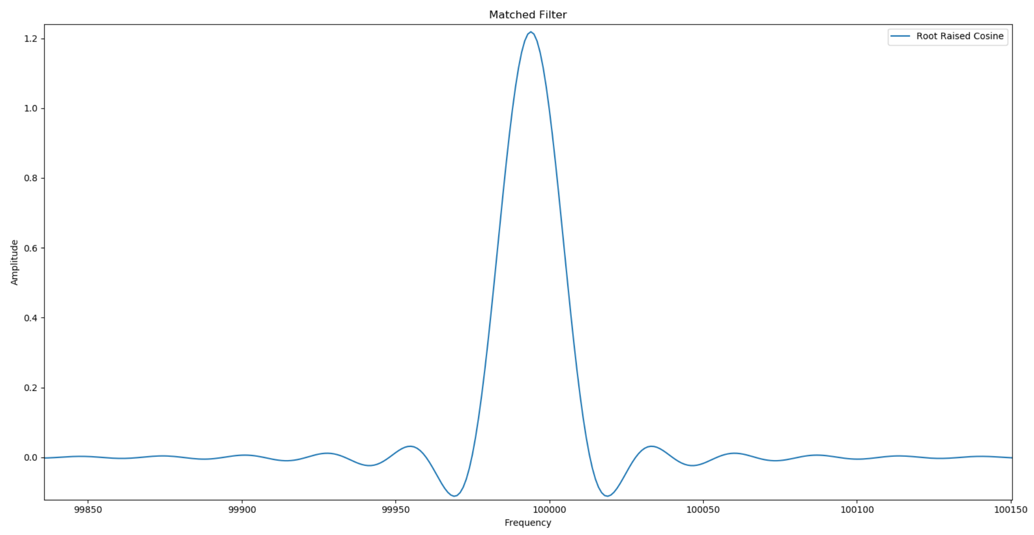


Fig. 1: Matched filter impulse response

The signal in Fig. 1 is impulse response for the root raised cosine, this signal response looks similar to the sinc function but functionally they are very different.

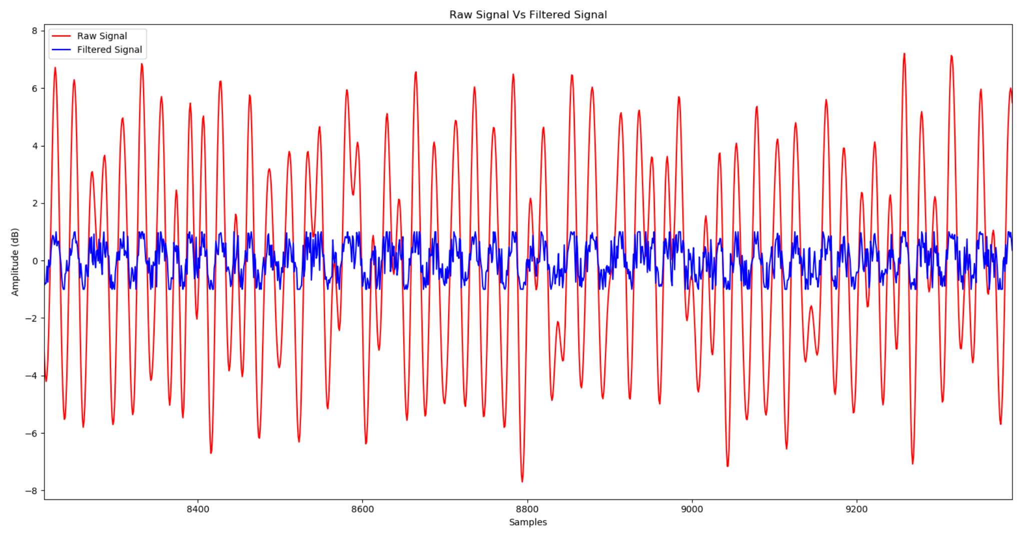


Fig. 2: Filtered data and recorded signal

Fig. 2 above depicts the original recorded data in blue and the signal after the matched filter in red. The signal after filtering looks more like a DBPSK signal from the previous practical. This new signal can be demodulated using a demodulation scheme similar to that of practical 2. The message demodulated from the DBPSK Signal is below.
01011110 01000110 01101001 01110011 01101000 01100101 01110010 01101101 01100101 01101110 00100000 01100001 01110010 01100101 00100000 01110010 01100101 01100101 01101100 00100000 01101101 01100101 01101110 00101110 00100100 00111100 10001101 00011111 00000011 00101111 00001001 10101100 00000001 01101110 01000101
This binary sequence decodes to the message : Fishermen are reel men.</ nE

## III. Huffman Coding

Huffman coding is a compression algorithm that forms the basics of file compression. Huffman coding uses unique binary codes to encode data as well as the probability or occurrence. Huffman coding exploits the fact some characters occur more often than others. Data that occurs more often has a higher probability of occuring and is assigned less bits while data that has a lower probability of occuring gets encoded with more bits. The Huffman code technique works by creating a binary tree of nodes. Fig. 3 below is a condensed form of the binary tree used for Huffman coding. The Huffman code for one language should not be used to decode another language even if the language uses the standard alphabet. Example would be using a English language Huffman tree to decode Afrikaans. This would result in a very low efficiency.

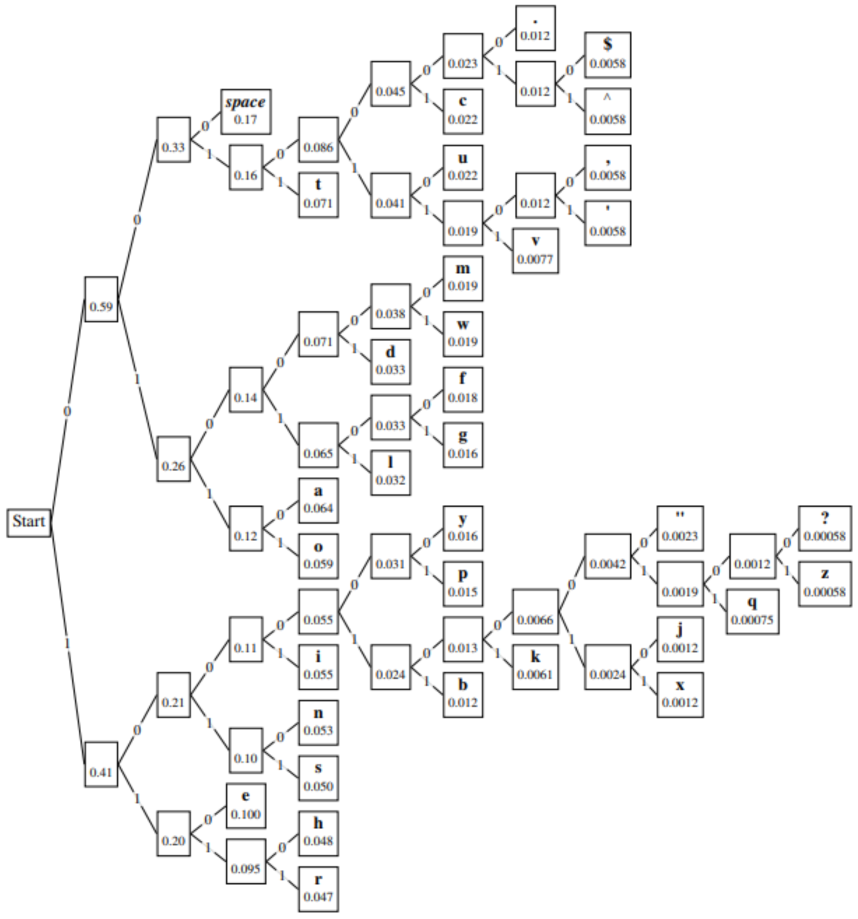


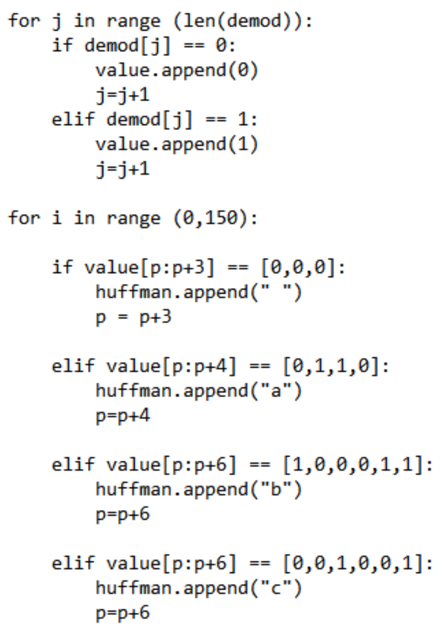
Fig. 3: Huffman tree to decode data[2]

| **Bits** | 00100011 | 0110 | 100010010 | 000 | 0111 | 110 |
|---|---|---|---|---|---|---|
| **Characters** | ^ | a | j | space | o | e |
| **Probability** | 0.0058 | 0.064 | 0.0012 | 0.17 | 0.059 | 0.100 |

TABLE I: Huffman table of a few characters the respective bits and probability.

## A. *Huffman methodology*

Multiple if statements are used to implement a scheme to decode the data. The data is firstly recorded and demodulated using demodulation code from practical 2.



```
for j in range (len(demod)):
    if demod[j] == 0:
        value.append(0)
        j=j+1
    elif demod[j] == 1:
        value.append(1)
        j=j+1

for i in range (0,150):

    if value[p:p+3] == [0,0,0]:
        huffman.append(" ")
        p = p+3

    elif value[p:p+4] == [0,1,1,0]:
        huffman.append("a")
        p=p+4

    elif value[p:p+6] == [1,0,0,0,1,1]:
        huffman.append("b")
        p=p+6

    elif value[p:p+6] == [0,0,1,0,0,1]:
        huffman.append("c")
        p=p+6
```

Fig. 4: Code snippet Huffman

The first loop converts the data file from a 1xN to a Nx1 list array. The second for loop steps through the new array and performs the decoding algorithm. value[p:p+3] == [0,0,0] and huffman.append(" ") decodes the 3 binary bits to a space p+3 reads the current position to the previous position plus 3 and then updates the counter variable. this in done for the entire data set.

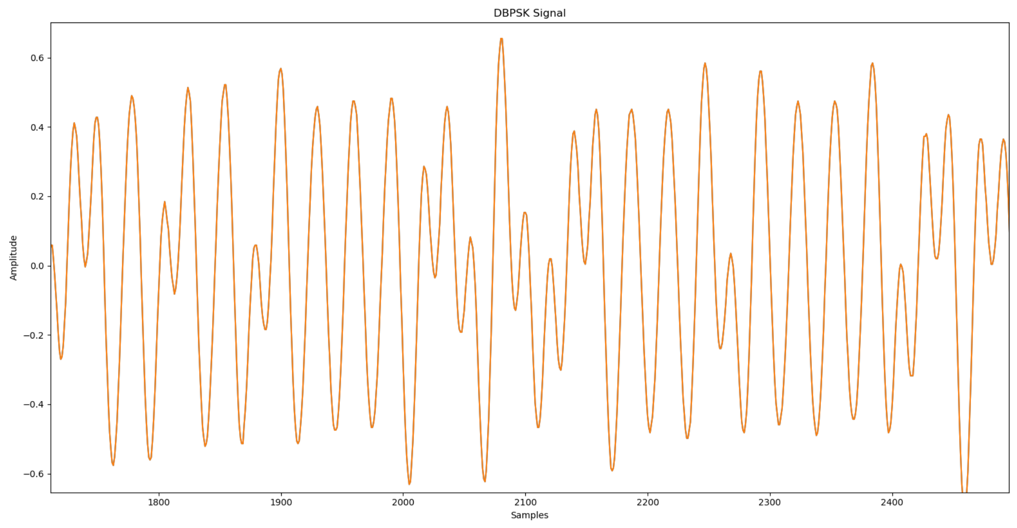
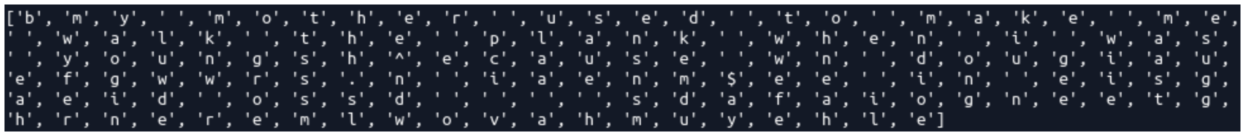


Fig. 5: Huffman DBPSK

Binary data that will be coded:
00001000 11010000 10000000 00100000 11100111 11011011 11000001 01010111 10010010 00001010 11000010 00001010 00010111 00000100 00110000 01000101 10010111 00010100 00011111 01100001 00001010 11011010 10100010 10000100 01111011 01010000 10010000 10001011 01011000 10000001 11001010 10100101 01101111 10001000 11110001 00101100



```
['b', 'm', 'y', ' ', 'm', 'o', 't', 'h', 'e', 'r', ' ', 'u', 's', 'e', 'd', ' ', 't', 'o', ' ', 'm', 'a', 'k', 'e', ' ', 'm', 'e',
' ', 'w', 'a', 'l', 'k', ' ', 't', 'h', 'e', ' ', 'p', 'l', 'a', 'n', 'k', ' ', 'w', 'h', 'e', 'n', ' ', 'i', ' ', 'w', 'a', 's',
' ', 'y', 'o', 'u', 'n', 'g', 's', 'h', '^', 'e', 'c', 'a', 'u', 's', 'e', ' ', 'w', 'n', ' ', 'd', 'o', 'u', 'g', 'i', 'a', 'u',
'e', 'f', 'g', 'w', 'w', 'r', 's', '.', 'n', ' ', 'i', 'a', 'e', 'n', 'm', '$', 'e', 'e', ' ', 'i', 'n', ' ', 'e', 'i', 's', 'g',
'a', 'e', 'i', 'd', ' ', 'o', 's', 's', 'd', ' ', ' ', ' ', ' ', 's', 'd', 'a', 'f', 'a', 'i', 'o', 'g', 'n', 'e', 'e', 't', 'g',
'h', 'r', 'n', 'e', 'r', 'e', 'm', 'l', 'w', 'o', 'v', 'a', 'h', 'm', 'u', 'y', 'e', 'h', 'l', 'e']
```

Fig. 6: Huffman Decoded message

The errors in the above decoding are a result of multiple factors such as incorrectly received bits and the demodulation code not being completely optimised for the data. Secondly the decoding starting point also factors into the decoding and lastly errors in the demodulation code will translate further on in the code. The following equations relay the different characteristics of the Huffman code for the data provided[3]

$$H = -(\sum_i P_i(\log_2 P_i)), \qquad Entropy \tag{2}$$

$$= -(0.064(\log_2 0.064) + 0.012(\log_2 0.012) + 0.022(\log_2 0.022) + ... + P_i(\log_2 P_i))$$

$$= 4.232$$

$$L = \sum_i P_i L_i \tag{3}$$

$$= (0.064 * 4) + (0.012 * 6) + (0.22 * 6) + ... + P_i L_i$$

$$= 4.3058$$

$$Efficiency, \quad n = \frac{100H}{L} \tag{4}$$

$$= \frac{(4.232)(100)}{4.3058}$$

$$= 98.286\%$$

$$Huffman\ Redundancy = 1 - n \tag{5}$$

$$= 1 - 0.98286$$

$$= 100 - 98.286$$

$$= 1.714\%$$

## IV. Linear Block Code

Linear block coding makes use of modula-2 linear matrix math in order to encode data into code words to be transmitted. The received data can be demodulated using the dbpsk demodulator from practical 2 and can then be decoded and errors can be corrected.

### A. *Linear Block Code methodology*

The linear block code for this practical used a codeword length of 6 bits to encode 3 bits of data. This means that the hamming distance is 3 bits, single bit errors can be corrected, and errors of up to 2 bits can be detected. The first 3 bits of the codeword is the data bits, and bits 4-6 are the parity bits.

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

Fig. 7: Generator Matrix[2]

The generator matrix was given as in Fig. 7 and consists of a 3x3 identity matrix (I), and a 3x3 parity matrix (P). Codewords are generated by multiplying the 3 message bits by the generator matrix. This can be realised in modula-2 math as follows:

$$Codeword = \mod 2(M \cdot G).$$

Before decoding the codewords, the transposed (H$^t$) of the parity check matrix (H) needs to be calculated where the parity check matrix consists of the 3x3 transposed of the parity matrix (P$^t$) and the 3x3 identity matrix. This results in:

$$\mathrm{H}^t = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Taking the modulus-2 of the dot product of the received codeword (Y) and the transpose parity check matrix (H$^t$) results in the syndrome (S). The syndrome indicates if and where in the codeword a error has occurred. For the purposes of this practical, 011 indicated an error in the first bit, 101 an error in the second bit, and 111 an error in the third bit. Errors in the rest of the codeword are not of interest, as the data is in the first 3 bits of the codeword.

After correcting the incorrect bits, the data can be extracted from the codeword, and converted to ASCII text to retrieve the joke.

## B. Linear Block Code results

An extract of the binary data received after demodulating with the dbpsk demodulator follows:
0 1 0 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 1 0 0 1 0 0 1 1 1 0 1 0 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 1
0 0 0 1 1 1 1 0 0 0 0 0 1 0 0 1 0 1 0 1 1 1 0 0 0 1 0 1 0 1 1 1 0 0 0 0 1 0 0 0 0 0 1 1 0 0
0 0 0 1 0 1 1 0 1 0 0 0 1 1 1 1 0 0 0 0 0 1 1 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 1 1 1
0 0 0 0 0 1 0 1 1 0 1 1 1 1 1 1 1 0 0 1 0 1 0 1 1 1 1 1 1 1 0 1 0 1 0 0 1 1 0 1 1 0 1 1 0 0 1
1 0 1 0 1 1 0 0 0 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 1 0 0 1 0 0 1 0 1 0 1 1 1 0 1 0 0 0 1 1

The first 10 received codewords, the calculated syndromes and the extracted data bits follows:
Received Codeword: [0 1 0 1 0 0] Syndrome: [0 0 1] Extracted data: [0 1 0]
Received Codeword: [0 1 1 0 0 1] Syndrome: [0 1 1] Extracted data: [1 1 1]
Received Codeword: [1 0 0 0 1 0] Syndrome: [0 0 1] Extracted data: [1 0 0]
Received Codeword: [1 0 0 1 0 0] Syndrome: [1 1 1] Extracted data: [1 0 1]
Received Codeword: [1 1 1 0 1 0] Syndrome: [0 1 1] Extracted data: [0 1 1]
Received Codeword: [1 0 0 1 0 0] Syndrome: [1 1 1] Extracted data: [1 0 1]
Received Codeword: [0 0 1 1 0 0] Syndrome: [0 1 1] Extracted data: [1 0 1]
Received Codeword: [0 0 0 1 0 0] Syndrome: [1 0 0] Extracted data: [0 0 0]
Received Codeword: [0 1 1 1 1 0] Syndrome: [1 0 0] Extracted data: [0 1 1]
Received Codeword: [0 0 0 0 1 0] Syndrome: [0 1 0] Extracted data: [0 0 0]

The resulting data for the entire joke: 01011110 01010111 01101000 01100001 01110100 00100000 01100100 01101111 00100000 01111001 01101111 01110101 00100000 01100111 01100101 01110100 00100000 01110111 01101000 01100101 01101110 00100000 01111001 01101111 01110101 00100000 01100011 01110010 01101111 01110011 01110011 00100000 01100001 00100000 01101010 01101111 01101011 01100101 00100000 01110111 01101001 01110100 01101000 00100000 01100001 00100000 01110010 01101000 01100101 01110100 01101111 01110010 01101001 01100011 01100001 01101100 00100000 01110001 01110101 01100101 01110011 01110100 01101101 01101111 01101110 00111111 00100100

The joke after converting the binary data to ASCII characters:

```
^What do you get when you cross a joke with a rhetorical questmon?$
```

## V. Convolution Code

Convolution coding uses an encoder to generate a 5-bit codeword from 2 bits of data and has one bit of memory, so each codeword depends on the previous data.

### A. Methodology

The encoder for the practical can be seen in Fig. 8.

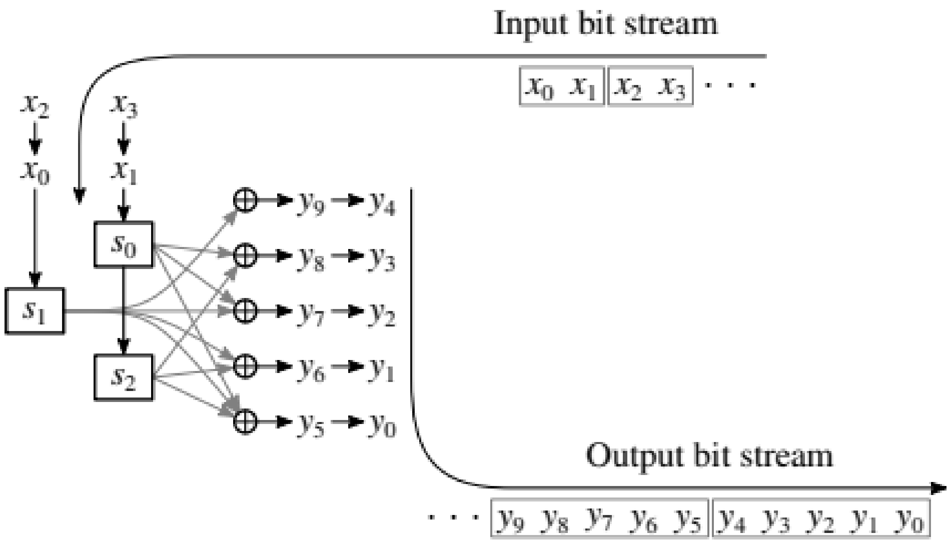


Fig. 8: Convolution encoder[2]

To demodulate convolution code, the entire system can be represented as a state machine. From each state the machine can transition to 4 different states, depending on the received codeword, and each transition representing 2 data bits. The transitions for each of the states werre calculated from the encoder, and is shown in Fig. 9. Convolution coding allows for errors to be detected and corrected. One way of decoding convolution coding, is setting up a trellis diagram. This is also known as the Veterbi algorithm. The trellis diagram for the practical starts at state 000. The errors in going to each new state is indicated in the state block, and at certain intervals, each 4 time steps for the practical, the lowest error path is chosen to extract the data from. The trellis diagram for the 5th character is shown in Fig. 10.

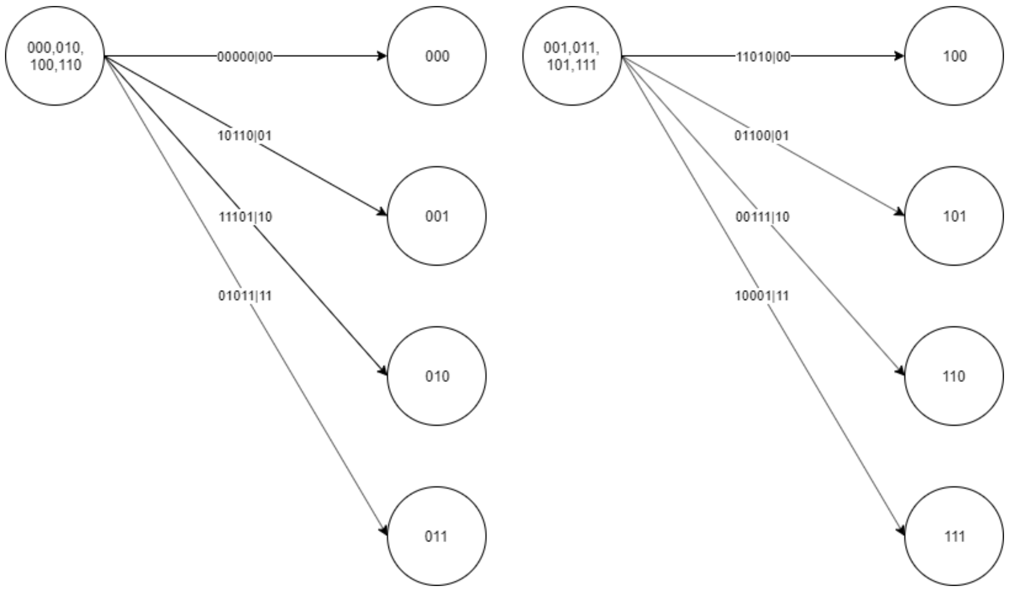


Fig. 9: State Transitions

For the practical, at least the first 10 characters had to be decoded. Creating a program to automatically use the algorithm for multiple steps was problematic because of the large amount of variables needed, as each stage would multiply the amount of variables by 4. It was decided to only write code to determine the next states and the errors to those states, when given a previous state and the data. This allowed the trellis diagram to be relatively quickly and accurately done by hand to decode the first 10 characters. Scans of these trellis diagrams will be added as appendix B.

The data used to decode the first 10 characters follows with the incorrect bits highlighted with a *. 10110 01100 1*1001 00111 10110 01100 01100 11010 10110 *10111 10110 01*000 *11100 00111 00*1*10 101*00 01100 00111 00000 01011 01100 00111 11101 0*0011 10110 00111 10110 0110*1 01100 10001 1101*1 11101 10110 10*101 *01010 01011 1000*1 *101*01 00000 000*10

The data extracted for the first 10 characters: 01011110 01010100 01100101 01100001 01100011 01101000 01100101 01110010 01110011 00100000

Converted to ASCII it spells out: ˆTeachers

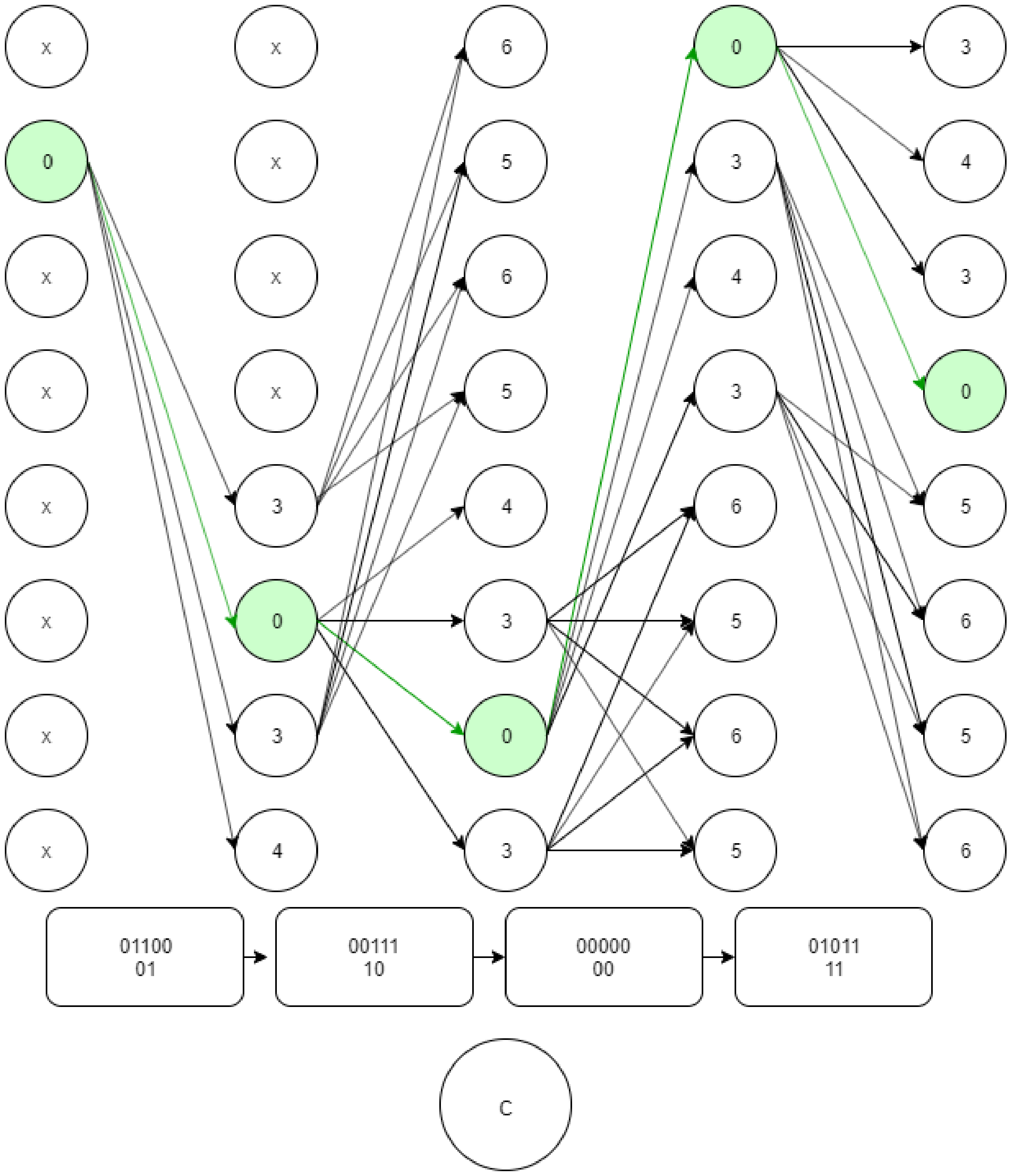


Fig. 10: Trellis diagram for the 5th character (c)

## VI. Discussion of results

### A. Matched Filtering

Raised cosine allows for Nyquist criterion to have zero ISI which means they don't interfere with the next symbol. However, a raised cosine squared does not and while we do correlation with the matched filter we essentially square the symbol shape so if we would use the normal raised cosine end up with a raised cosine squared which is not zero ISI. However, root raised cosine filter and root raised cosine pulse shape will result in a raised cosine which has zero ISI. The root raised cosine impulse response is very similar to a sinc however the two can simply be swapped in place of the other. Looking closely at eq.1 the RRC is a from of a piece-wise sinc function at all values barring $t = 0$ and $t = \frac{Ts}{4\beta}$. Convolving the known pulse shape filter with the data requires both the data and the root raised cosine to be flipped and time delayed which can be a tedious task to perform and for this reason make more sense to correlate the data.

### B. Huffman coding

Huffman coding is able achieve a high level of efficiency and a low level of redundancy. The principal is assigning symbols with high probability of occurrence a lower bit count.

The Huffman code for the English language should not be used to decode another language even if the language uses the standard alphabet. Example would be using a English language Huffman tree to decode Afrikaans. This is because the probability of occurrence of different letter for Afrikaans is totally different to that of English. This results in a much lower efficiency.

### C. Linear Block coding

Linear Block coding is an effective method of encoding data to allow for error correction. It is very simple to implement in software, and should also be easy to implement in hardware because of the use of modula-2 math. This allows for cheap XOR and AND logic gates to be used to do the necessary calculations in hardware.

Unfortunately only single bit errors can be corrected with the hamming distance of 3 that was used in the practical. This caused one letter to be interpreted wrong late in the joke, where a 'i' was interpreted as a 'm'.

### D. Convolution coding

Convolution coding also allows for error correction, although it seems to be more difficult to implement in software. The veterbi algorithm is very easy to use visually to manually decode the data. The algorithm is not always perfect, as there is sometimes ambiguities where the user needs to decide on the most appropriate option, but these errors generally do not affect later parts of the message, as the paths generally end up in the same state after one or two iterations.

## REFERENCES

[1] Masoud Salehi John G. Proakis. *Communication Systems Engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 2001.

[2] W. P. du Plessis. *ESC 320 Practical 3 Guide*. University of Pretoria, 2019.

[3] B. P. Lathi. *Modern Digital and Analog Communication Systems*. Oxford University Press, Oxford, UK, 4th edition, 2010.

## Appendix A
## Computer Code

```
import csv
import numpy as np
import matplotlib.pyplot as plt
from pylab import *
from commpy.modulation import QAMModem
from commpy.filters import rrcosfilter

N = 99994  # output size
rate = 2.4e6

rrcos = rrcosfilter(N*2, 0.8, 1, 24)[1] #Scipy Website Commpy Library root r
plt.plot(rrcos, label='Root Raised Cosine')
plt.ylabel('Amplitude')
plt.xlabel('Frequency')
plt.title('Matched Filter')
plt.legend()



data = np.genfromtxt("matched.txt", dtype=complex128) #reads the master data

data_x = (20*np.log10((abs(np.fft.fft(mag)))))
phase = np.fft.fftshift(np.angle(np.fft.fft(new_data)))
new_data = np.correlate((data),psine,"same")



new_data = np.correlate((data),psine)


rollof=10
rrcos_2=3*np.sinc(np.arange(-rollof,rollof,0.5)*np.pi/rollof)
Filterout_t=np.correlate((data),psine,"same")
mag = np.abs(Filterout_t)
angle = np.angle(Filterout_t)
unwrap = np.unwrap(angle)
```

```
plt.show()


import csv
import numpy as np
import matplotlib.pyplot as plt
from pylab import *
import cmath
from scipy.signal import butter, lfilter, freqz, filtfilt

#code from prac 2

rate = 2.4e6
peak = 1012

data= np.genfromtxt("BPSK_new.txt",dtype=complex)
magnitude = np.abs(data)
phase = np.angle(data)
d=len(data)

unwrapped = -np.unwrap(phase)
freqin = np.gradient(unwrapped)
absfreq = abs(freqin)
#plt.plot(absfreq)
```

```
signal1 = np.zeros(d)
phase1 = np.zeros(d)
clockimp = np.zeros(d)



for i in range(d):
    if (magnitude[i] < 0.4):
        signal1[i] = 0
    if (magnitude[i] >= 0.4):
        signal1[i] = 1



cleanangle = np.zeros(d)
test1 = False
for i in range(d):
    if (absfreq[i] <= 1):
        test1 = False
    if absfreq[i] > 1:
        if test1 == False:
            cleanangle[i] = cleanangle[i] + 1
            test1 = True
            if (cleanangle[i] == 2):
                cleanangle[i] = 0


angle2 = 0
angleout1 = np.zeros(d)
for i in range(d):
    if cleanangle[i] == 1:
        angle2 = angle2 + 1
        if angle2 == 2:
            angle2 = 0
    angleout1[i] = angle2


angleout1 = np.abs(angleout1)



data_x = (20*np.log10((abs(np.fft.fft(magnitude)))))
```

```
phase1 = np.fft.fftshift(np.angle(np.fft.fft(data)))
data_y = np.linspace(0, (1/rate)*d, d)


freq = (peak/d)*2.4*1000000
angle = phase1[peak]


clock = 1*np.cos((2*np.pi*freq*data_y)+angle)




outmag = np.zeros(1500)
outphase = np.zeros(1500)
data2 = np.zeros(1500,dtype = complex)


test123 = False
counter123 = 0
for i in range(d):
    if (clock[i] >= 0.98):
        if test123 == False:
            outphase[counter123] = angleout1[i]
            outmag[counter123] = signal1[i]
            data2[counter123] = data[i]
            clockimp[i] = 1
            test123 = True
            counter123 = counter123 + 1
    if (clock[i] < 0.98):
        test123 = False

phasebefore = np.angle(data2)
magbefore = np.abs(data2)

#plt.plot(outphase)

dataoutput = np.zeros(1500,dtype = int)
for i in range(1500):
    if (outphase[i] != outphase[i-1]):
        dataoutput[i] = '1'
    if (outphase[i] == outphase[i-1]):
        dataoutput[i] = '0'


np.savetxt('demod.txt', dataoutput,newline='\n',fmt='%s')




plt.plot(dataoutput)
plt.ylabel('Amplitude')
plt.xlabel('Samples')
plt.title('Post Demodultion DBPSK')
print(dataoutput)
```

```
import csv
import numpy as np
import matplotlib.pyplot as plt
from pylab import *
import cmath
from scipy.signal import butter, lfilter, freqz, filtfilt

#rate = 2.4e6

#data = np.genfromtxt("BPSK.txt", dtype=complex128)
#mag = np.abs(data)
#data_x = (20*np.log10((abs(np.fft.fft(mag)))))
#
#freq = (1024/24570)*rate
b=12


#works
#b=12


demod=[]
value=[]
huffman = []
demod = np.genfromtxt("demod.txt")
print(demod)
plt.plot(demod)
#plt.plot(np.abs(data))
```

```
#plt.plot(data_x)
plt.show()
j=0



for j in range (len(demod)):
    if demod[j] == 0:
        value.append(0)
        j=j+1
    elif demod[j] == 1:
        value.append(1)
        j=j+1

#This method was done with the assitance of Conru Boshof  u17098204

for i in range (0,150):

    if value[b:b+3] == [0,0,0]:
        huffman.append(" ")
        b=b+3

    elif value[b:b+7] == [0,0,1,0,0,0,0]:
        huffman.append(".")
        b=b+7

    elif value[b:b+8] == [0,0,1,0,0,0,1,0]:
        huffman.append("$")
        b=b+8

    elif value[b:b+8] == [0,0,1,0,0,0,1,1]:
        huffman.append("^")
        b=b+8

    elif value[b:b+8] == [0,0,1,0,1,1,0,0]:
        huffman.append(",")
        b=b+8

    elif value[b:b+8] == [0,0,1,0,1,1,0,1]:
        huffman.append("'")
        b=b+8
    elif value[b:b+4] == [0,1,1,0]:
        huffman.append("a")
        b=b+4

    elif value[b:b+6] == [1,0,0,0,1,1]:
        huffman.append("b")
        b=b+6

    elif value[b:b+6] == [0,0,1,0,0,1]:
        huffman.append("c")
        b=b+6

    elif value[b:b+5] == [0,1,0,0,1]:
        huffman.append("d")
        b=b+5

    elif value[b:b+3] == [1,1,0]:
        huffman.append("e")
        b=b+3

    elif value[b:b+6] == [0,1,0,1,0,0]:
        huffman.append("f")
        b=b+6

    elif value[b:b+6] == [0,1,0,1,0,1]:
        huffman.append("g")
        b=b+6

    elif value[b:b+4] == [1,1,1,0]:
        huffman.append("h")
        b=b+4

    elif value[b:b+4] == [1,0,0,1]:
        huffman.append("i")
        b=b+4

    elif value[b:b+9] == [1,0,0,0,1,0,0,1,0]:
        huffman.append("j")
        b=b+9

    elif value[b:b+7] == [1,0,0,0,1,0,1]:
        huffman.append("k")
        b=b+7

    elif value[b:b+5] == [0,1,0,1,1]:
        huffman.append("l")
        b=b+5

    elif value[b:b+6] == [0,1,0,0,0,0]:
        huffman.append("m")
        b=b+6

    elif value[b:b+4] == [1,0,1,0]:
        huffman.append("n")
        b=b+4

    elif value[b:b+4] == [0,1,1,1]:
        huffman.append("o")
        b=b+4

    elif value[b:b+6] == [1,0,0,0,0,1]:
        huffman.append("p")
        b=b+6

    elif value[b:b+10] == [1,0,0,0,1,0,0,0,1,1]:
        huffman.append("q")
        b=b+10

    elif value[b:b+4] == [1,1,1,1]:
        huffman.append("r")
        b=b+4

    elif value[b:b+4] == [1,0,1,1]:
        huffman.append("s")
        b=b+4

    elif value[b:b+4] == [0,0,1,1]:
        huffman.append("t")
        b=b+4

    elif value[b:b+6] == [0,0,1,0,1,0]:
        huffman.append("u")
        b=b+6
    elif value[b:b+7] == [0,0,1,0,1,1,1]:
        huffman.append("v")
        b=b+7

    elif value[b:b+6] == [0,1,0,0,0,1]:
        huffman.append("w")
        b=b+6

    elif value[b:b+9] == [1,0,0,0,1,0,0,1,1]:
        huffman.append("x")
        b=b+9

    elif value[b:b+6] == [1,0,0,0,0,0]:
        huffman.append("y")
        b=b+6

    elif value[b:b+11] == [1,0,0,0,1,0,0,0,1,0,1]:
        huffman.append("z")
        b=b+11

    elif value[b:b+11] == [1,0,0,0,1,0,0,0,1,0,0]:
        huffman.append("?")
        b=b+11

    else:
        huffman.append("%")
        b=b+1



print(len(huffman))
print(len(value))
print(len(demod))
print(huffman)
```

```
import csv
import numpy as np
import matplotlib.pyplot as plt
from pylab import *
import cmath


data = np.genfromtxt("blockdata.txt",dtype=int) #Data from DBPSK
ht = np.array([[0,1,1],[1,0,1],[1,1,1],[1,0,0],[0,1,0],[0,0,1]]) #HT matrix

dataoutput = np.zeros(800,dtype = int) #Empty output array
for i in range(205): #How many codewords to run the loop for
    testdata = np.array([data[6*i],data[(6*i)+1],data[(6*i)+2],data[(6*i)+3
    print("Received Codeword: ")
    print(testdata)

    syndrome = (np.dot(testdata,ht))%2 #Multiplication for syndrome

    outdata = np.array([data[6*i],data[(6*i)+1],data[(6*i)+2]])

#    print(outdata)
    print("Syndrome: ")
    print(syndrome)
    left = np.array([1,0,0])  #Check for error
    mid = np.array([0,1,0])
    right = np.array([0,0,1])
    leftcheck = np.array([0,1,1])
    midcheck = np.array([1,0,1])
    rightcheck = np.array([1,1,1])
    if (syndrome[0]==0 and syndrome[1]==1 and syndrome[2]==1):  #Correct er
        finaldata = outdata ^ left
    elif(syndrome[0]==1 and syndrome[1]==0 and syndrome[2]==1):
        finaldata = outdata ^ mid
    elif(syndrome[0]==1 and syndrome[1]==1 and syndrome[2]==1):
        finaldata = outdata ^ right
    else:
        finaldata = outdata
    print("Extracted data: ")
    print(finaldata)
    print("=========================================")
    dataoutput[i*3] = finaldata[0]
    dataoutput[(i*3)+1] = finaldata[1]
    dataoutput[(i*3)+2] = finaldata[2]


print(dataoutput) #Output data
```

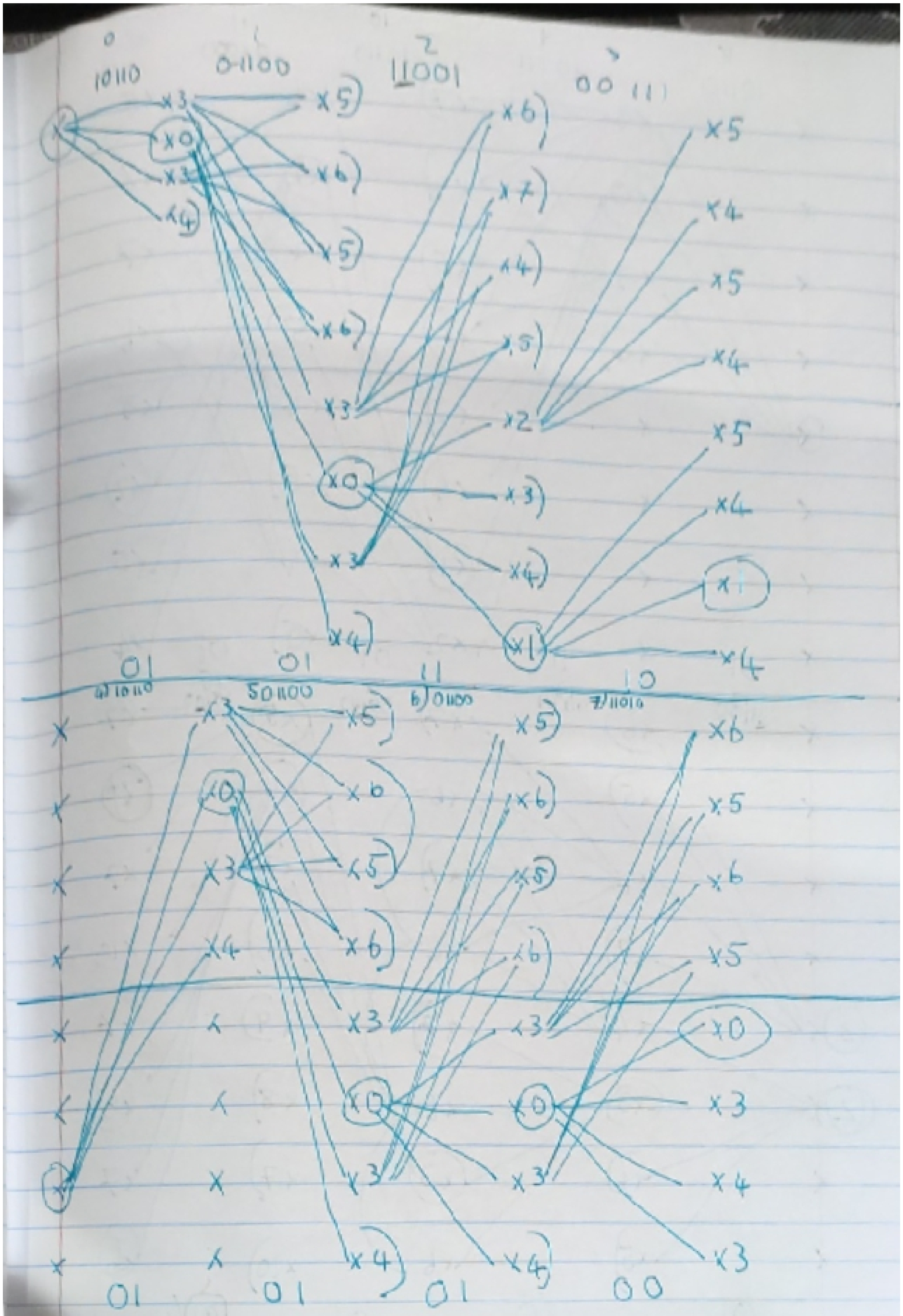
```
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 14 16:50:27 2019

@author: Wikus
"""

import csv
import numpy as np
import matplotlib.pyplot as plt
from pylab import *
import cmath





def compare(A,B):
    if A==B:
        X = 0
    else:
        X = 1
    return(X)

#def errorget(testdata,s2,s1,s0):
```

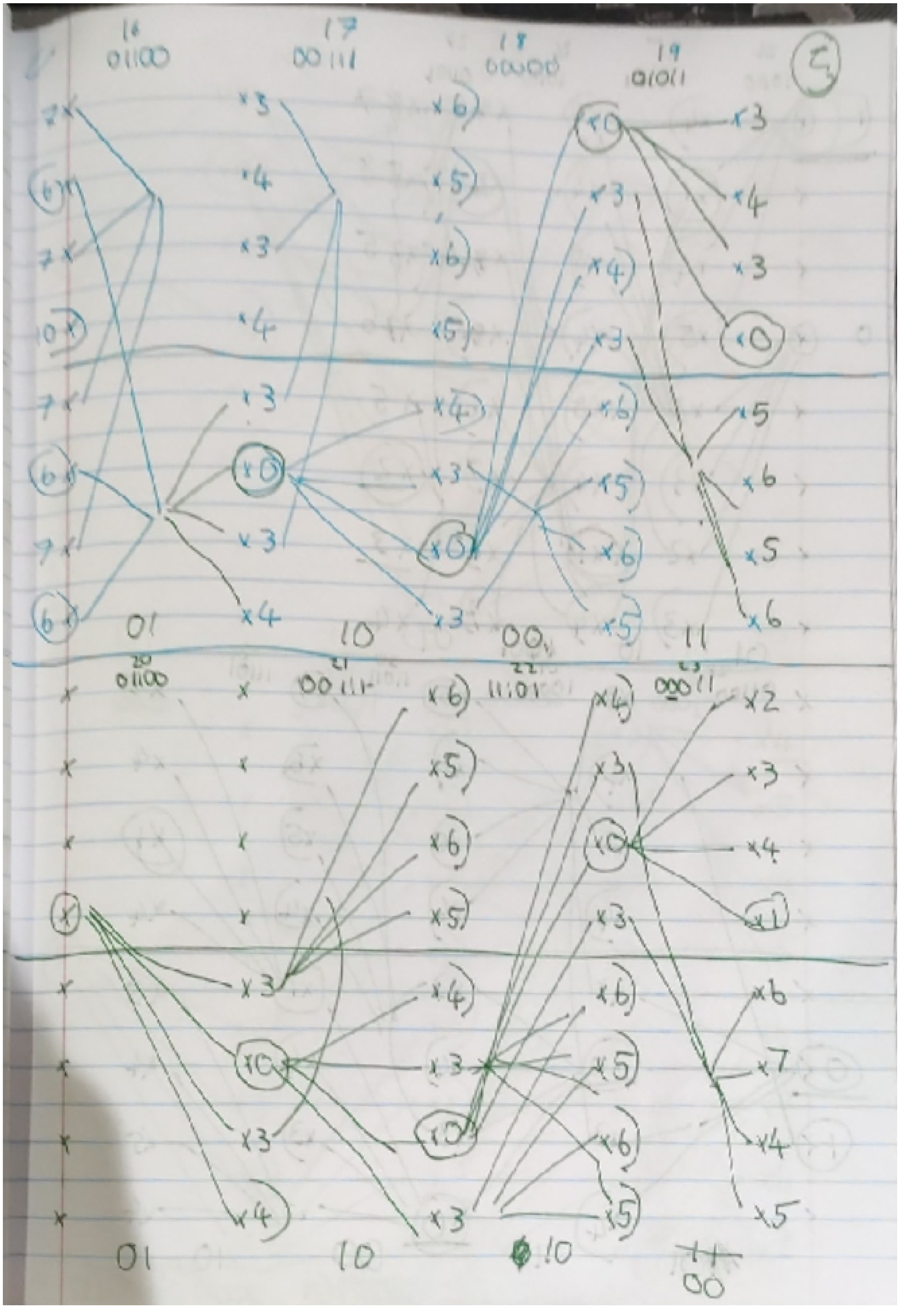
```
#    if s2 == 0 and s1 == 0 and s0 == 0:
#        Error1 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare
#        Error2 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare
#        Error3 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare
#        Error4 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare
#    elif s2 == 0 and s1 == 0 and s0 == 1:
#        Error1 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare
#        Error2 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare
#        Error3 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare
#        Error4 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare
#    elif s2 == 0 and s1 == 1 and s0 == 0:
#        Error1 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare
#        Error2 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare
#        Error3 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare
#        Error4 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare
#    elif s2 == 0 and s1 == 1 and s0 == 1:
#        Error1 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare
#        Error2 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare
#        Error3 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare
#        Error4 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare
#    elif s2 == 1 and s1 == 0 and s0 == 0:
#        Error1 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare
#        Error2 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare
#        Error3 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare
#        Error4 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare
#    elif s2 == 1 and s1 == 0 and s0 == 1:
#        Error1 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare
#        Error2 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare
#        Error3 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare
#        Error4 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare
#    elif s2 == 1 and s1 == 1 and s0 == 0:
#        Error1 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare
#        Error2 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare
#        Error3 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare
#        Error4 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare
#    elif s2 == 1 and s1 == 1 and s0 == 1:
#        Error1 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare
#        Error2 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare
#        Error3 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare
#        Error4 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare
#
#    return Error1,Error2,Error3,Error4
```

```
data2 = np.genfromtxt("trellisdata1.txt",dtype=int)
#s2 = 1
#s1 = 1
#s0 = 1
#
#dl = 1
#dr = 0
#
#s1 = dl
#s2 = s0
#s0 = dr
#
#
#
#y4 = s1
#y3 = s0 ^ s2
#y2 = s0 ^ s1
#y1 = s1 ^ s2
#y0 = s0 ^ s1 ^ s2
#
#print(y0,y1,y2,y3,y4,'and',s2,s1,s0)

data = data2[0:len(data2)]

number = 11

currentplace = (number*5)
testdata = data[currentplace:currentplace+5]

print(testdata)

state = 2

for i in range(1):
##    currentplace = 0
##
##    testdata1 = data[currentplace:currentplace+5]
##    state = newstate
##    possiblestates = []
##
##
    if state == 1:
        Error1 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare(0
        Error2 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare(1
        Error3 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare(1
        Error4 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare(0
        possiblestates = [1,2,3,4]
    elif state == 2:
        Error1 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare(0
        Error2 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare(1
        Error3 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare(1
        Error4 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare(0
        possiblestates = [5,6,7,8]
    elif state == 3:
        Error1 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare(0
        Error2 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare(1
        Error3 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare(1
        Error4 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare(0
        possiblestates = [1,2,3,4]
    elif state == 4:
        Error1 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare(0
        Error2 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare(1
        Error3 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare(1
        Error4 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare(0
        possiblestates = [5,6,7,8]
    elif state == 5:
        Error1 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare(0
        Error2 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare(1
        Error3 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare(1
        Error4 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare(0
        possiblestates = [1,2,3,4]
    elif state == 6:
        Error1 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare(0
        Error2 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare(1
        Error3 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare(1
        Error4 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare(0
        possiblestates = [5,6,7,8]
    elif state == 7:
        Error1 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare(0
        Error2 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare(
        Error3 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare(
        Error4 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare(
        possiblestates = [1,2,3,4]
    elif state == 8:
        Error1 = compare(1,testdata[0]) + compare(1,testdata[1]) + compare(
        Error2 = compare(0,testdata[0]) + compare(1,testdata[1]) + compare(
        Error3 = compare(0,testdata[0]) + compare(0,testdata[1]) + compare(
        Error4 = compare(1,testdata[0]) + compare(0,testdata[1]) + compare(
        possiblestates = [5,6,7,8]


print(Error1)
print(Error2)
print(Error3)
print(Error4)
```

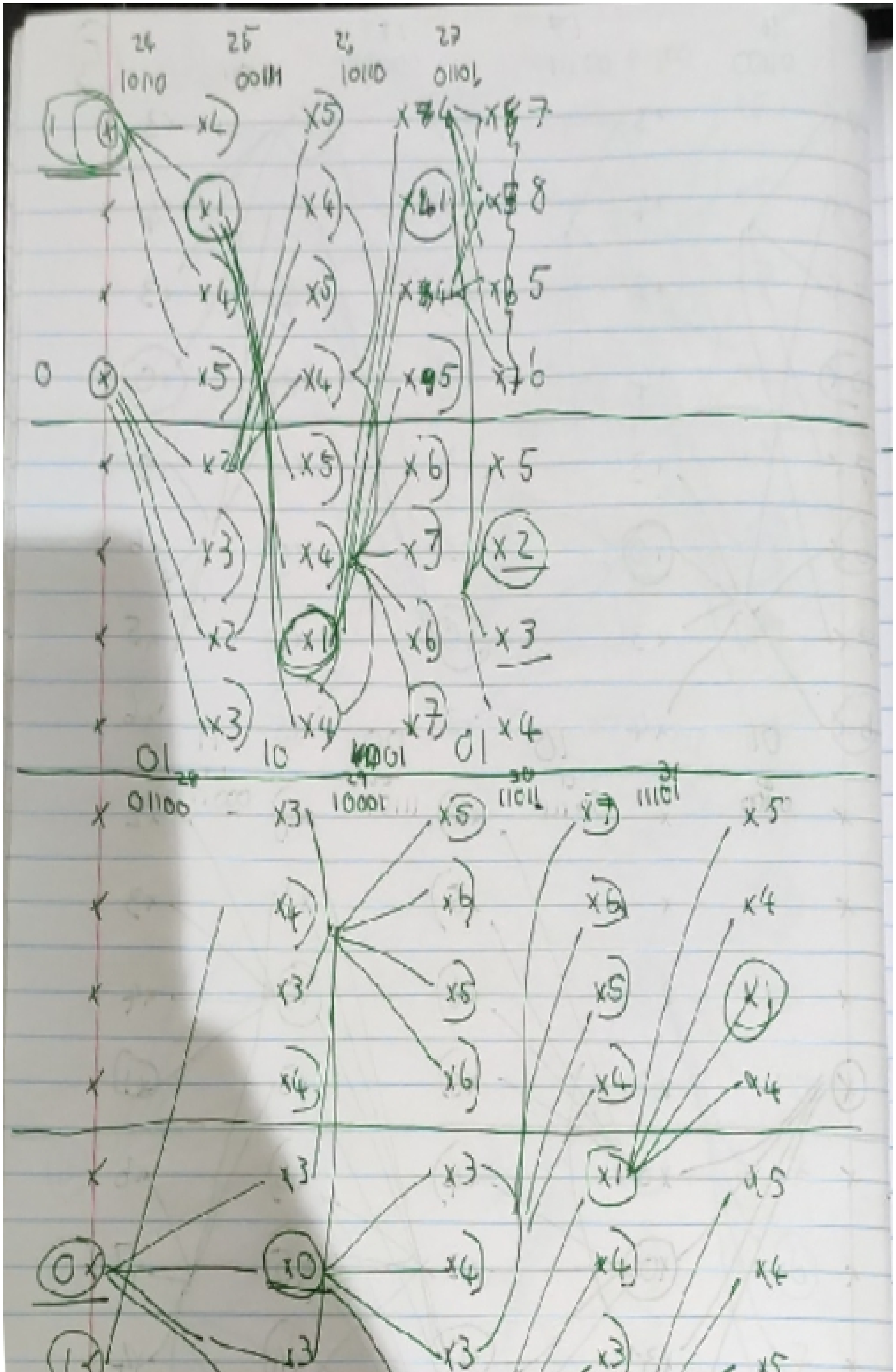
.

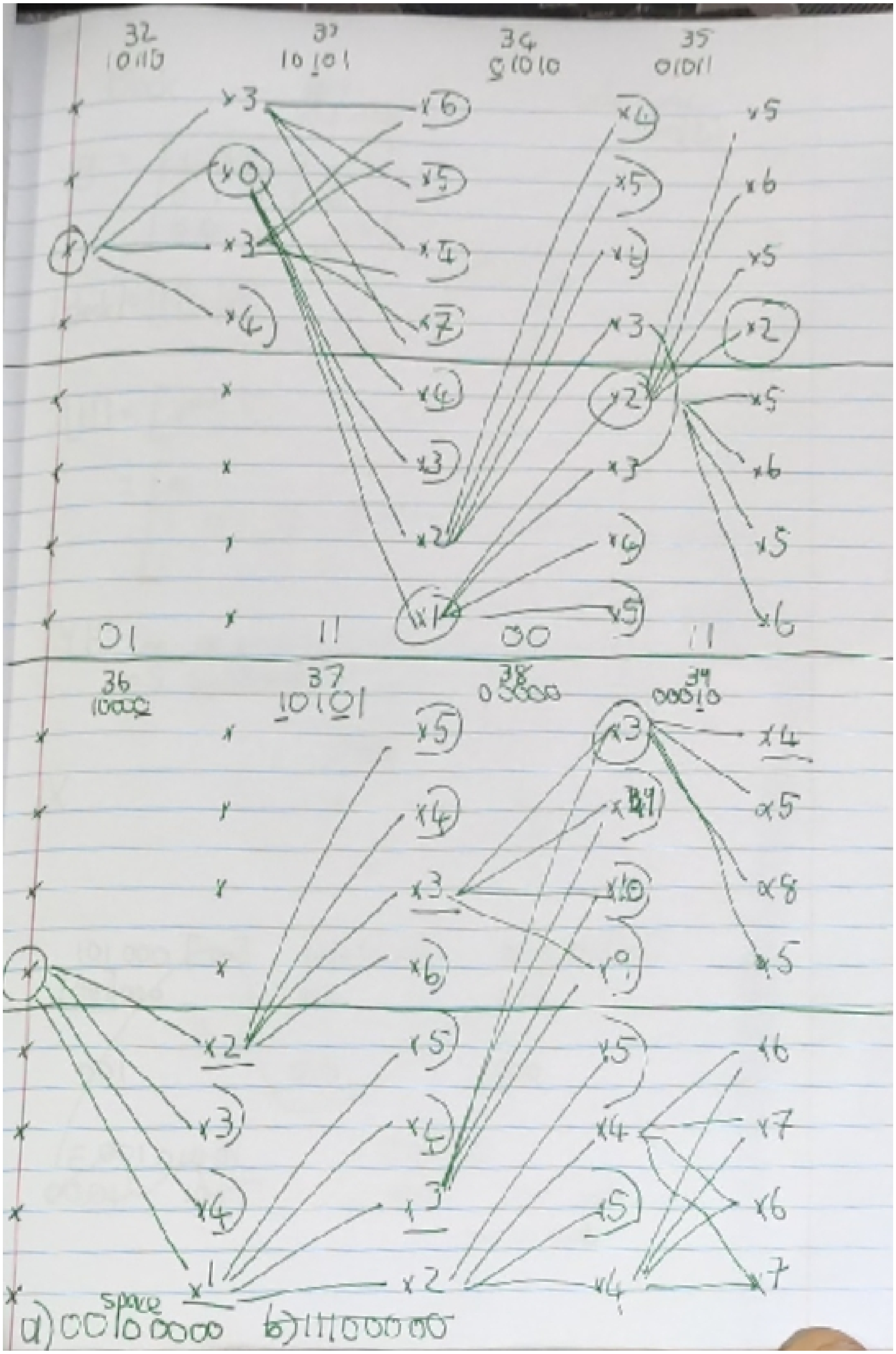
## Appendix B
## Trellis diagram





5
16
01100
17
00111
18
00000
19
01011
01
10
00
11
20
01100
21
00111
22
11101
23
00011
01
10
10
00


24
1010
25
0011
26
1010
27
0101
28
01100
29
10001
30
1101
31
1110
01
10
01


32
10110
33
10101
34
01010
35
01011
01
11
00
11
36
10000
37
10101
38
00000
39
00010
space
a) 00100000
b) 1100000