UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Faculty of Engineering, Built Environment and Information Technology

EDC 310

Digital Communication

Practical 3 Assignment

Name and Surname	Student Number	Signature	% Contribution
Kwaku Bediako	u04465483		0
Lefa Raleting	u14222460		100

By signing this assignment we confirm that we have read and are aware of the University of Pretoria's policy on academic dishonesty and plagiarism and we declare that the work submitted in this assignment is our own as delimited by the mentioned policies. We explicitly declare that no parts of this assignment have been copied from current or previous students' work or any other sources (including the internet), whether copyrighted or not. We understand that we will be subjected to disciplinary actions should it be found that the work we submit here does not comply with the said policies.

November 12, 2020

Contents

1 Introduction

In practical two, two multi-path detection algorithm were investigated, Decision feedback equaliser(**DFE**) and Maximum likelihood sequence estimation(**MLSE**) algorithms.These encoding schemes showed a decrease in bit error rate the higher the Signal To Noise ratio(**SNR**).

This characteristic is important for successful transmission and reception of information, however its not the only important factor. SNR relates to power, the higher the SNR the higher the power consumption, or the power needed to transmit the information and receive the information.

So for digital communication as the designer its it important to design a system that has a high throughput and low power consumption. Thus the need for research on error correction coding or error control coding process.

Error correction coding is a process which the encoding scheme is done in such a way that the errors that occur in the information after transmission and modulation, can be corrected in the receiver after the demodulation of the detected information.

This encoding scheme adds redundant information to the transmitted information in a controlled fashion, and the decoder uses this redundant information to help correct errors. The leads to reduced bit error rate at a low SNR compared to an uncoded scheme.

Another encoding scheme which will be studied is known as Convolutional coding, it is almost like the linear code blocks encoding scheme introduced above however additional bits are not sent with the transmitted data but rather the parity bits.

Convolutional coding utilizes a sliding shift register in order to calculate the parity bits of length K. K is known as the constraint length. The longer K is the more bits that can be influenced or corrected, this comes at a slight trade off. The bigger the constraint length the slower the decoding.

The fist experiment will be conducted by developing two simulation platforms to encode and decode BPSK-modulated information through and AWGN channel that has no multi-path.

The method that will be used to determine the most probable sequence of uncoded transmitted symbols is syndrome decoding and the encoding process that will be used is linear block coding , with a rate of 0.5.

The second experiment will be conducted by developing two simulation platforms to encode and decode BPSK-modulated information through and AWGN channel that has no multi-path.

The method that will be used to determine the most probable sequence of uncoded transmitted symbols is MLSE and the encoding process that will be used is Convolutional coding , with a rate of 2/3.

2 Theoretical background

2.1 Linear block codes

Linar block codes are the most basics error correction codes. In order to get the codeword the source bits are multiplied with the generator matrix (**G**) and adding the AWGN channel but with no multipath.

$$c = sG + n \tag{2.1}$$

where **c** is the code word ,**s** is the source bits and **n** is the AWGN channel.

The generator matrix at a rate $R_c = \frac{1}{2}$ can be expressed as:

$$\begin{aligned} G &= [\mathbf{I}_{kxk} \quad \mathbf{P}] \\ &= \begin{bmatrix} 1 & ... & 0 & P_{11} & ... & P_{k-n} \\ . & ... & . & . & ... & . \\ 0 & . & 1 & P_{400} & ... & P_{k*(k-n)} \end{bmatrix} \end{aligned} \tag{2.2}$$

Where I is the identity matrix,P is the parity matrix , k=400 and n=200 resulting in a rate of 0.5.

The information is transmitted using the code completed in practical one for the BPSK simulation platform. Once the information is transmitted the information is the then detected in the receiver end.

The most probable sequence of uncoded transmitted symbols are determined by using the syndrom decoding method. The symdrom decoding method works by determining the syndrom vector in order to determine the error possition and flipping the bit at that position. The syndrom vector can be determined as follows:

$$\mathbf{z} = \mathbf{c}\mathbf{H}^T \tag{2.3}$$

Where **z** is the syndrom vector , **c** is the estimated bits(codeword) and **H** is the parity check bits which can be determined as follows:

$$\mathbf{H} = [\mathbf{P}^T \quad \mathbf{I}_{(k-n)x(k-n)}] \tag{2.4}$$

If $\mathbf{z} = \mathbf{0}$ then there were no errors, however if $\mathbf{z} \neq \mathbf{0}$ then search for **z** in the $\mathbf{H}^T$ matrix and if its not there search for two lines in the $\mathbf{H}^T$ matrix that if XORED produce **z**.

The number of errors that a given code can correct (**t**) is determined by d_min, for this experiment $d_min = 4$, where:

$$\begin{aligned} t &= \frac{d_min - 1}{2} \\ &= \frac{4 - 1}{2} \\ &= 1.5 \\ &\approx 2 \end{aligned} \tag{2.5}$$

2.2 Convolutional coding

Convolutional codes are ver good error correction codes able to correct burst errors in wireless communication Encoding works by by shifting the bits through a shift register iteratively in order to encode a block of uncoded bits, while the contents of the shift register are multiplied as determined by the encoder connections. The encoder takes **K** bits at a time and produces p parity bits. Decoding is performed using the MLSE algorithm, similar to how sequence detection in the presence of multipath is performed.

3 Design/Method

3.1 Linear block codes

To simulate communication between a transmitter and receive, bits are generated randomly using a uniform number generator rated and mapped to symbols using the BPSK modulation schemes constellation maps developed in practical 1.

To simulate the noise that's normally added to the signal during transmission, additive white Gaussian noise (**AGWN**) is used to simulate this. The following formula is used to emulate this:

$$r_t = [\sum_{n=0}^{L-1} s_{t-n} * c_n] + \sigma * (\alpha) \tag{3.1}$$

Where:

- $\sigma = \frac{1}{\sqrt{10^{\frac{SNR}{10}} * 2 * R_c}}$ and M is the number of symbols in the respective constellation map.
- α is a sample from a zero mean Gaussian distribution
- L is the length of the CIR

The code used for Linear block codes encoding can be found bellow:

The code for Linear block codes decoding can be found bellow:

4 Simulations and Results

4.1 Linear block codes

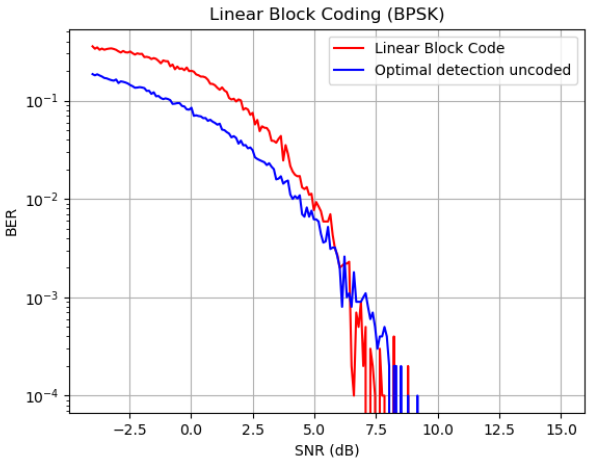

Figure 4.1: Linear Block Coding BER

4.2 Convolutional coding

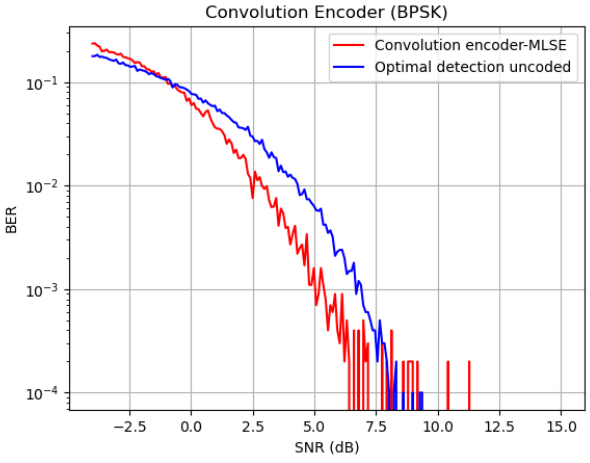

Figure 4.2: Convolution Encoder BER

4.3 Convolution encoder vs Linear block codes

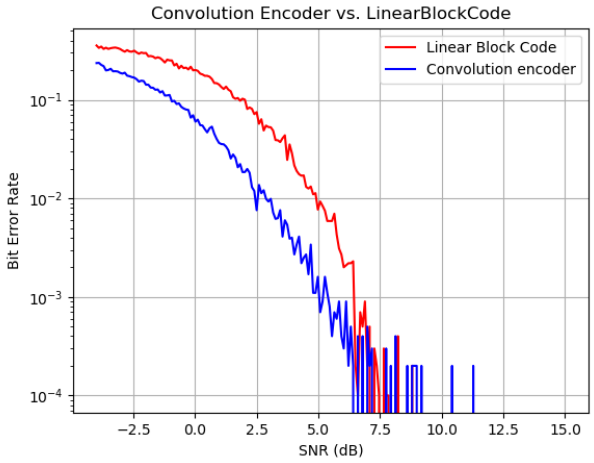

Figure 4.3: Convolution Encoder vs Linear block code BER

5 Discussion

It can be observed that the encoding procedure for linear block code is rather uncomplicated and easy to do. Given the generator matrix G in the practical experiment, this completely characterized the code. Using the G matrix, the generated bits S and *equation 2.1* the desired n-bit code-word was produced. Linear block code offers an advantage of being able to correct the error in received bits on the receiver end.

The number of bit errors that can be corrected rely on the number of bits by which the code-word that was generated by the code differs. Looking at **Figure 4.1** the disadvantage of linear block codes can be observed at the low SNR (Signal to Noise Ratio) ranges. This is due to the fact at low SNR the transmitted data is more likely to be corrupted and because Linear block codes introduce additional bits to assist with the decoding. This however increases the amount of data that can be corrupted in transmission and leads to a higher BER(Bit Error Rate) compared to optimal detection of uncoded transmission data bit.

At higher SNR the linear block code BER seems to decay at the same rate as the uncoded optimal detection BER, although at higher SNR linear block coding performs better than the latter implementation ,the performance improvement is marginal and in the case of BPSK transmissions isn't justified especial if operating at low SNR values.

Linear block coding has an overall encoding complexity of O(nk) operations. This means O(k) operations are needed for each of n k parity bits in **c**.

Convolutional coding although it has high computational complexity ,makes it up in the ability to correct error at low SNR. From **Figure 4.2** it can be seen that the Covultional coding BER is much lower than the optimal detection for uncoded bits. This is due to how the parity bits carefully selected by functions that run over different set of K bits.

Convolutional coding has lower BER than the uncoded optimal detection at lower SNR, this makes more energy efficient, because less amount of power is necessary to transmit the same data using Convolutional coding than the uncoded optimal detection. Convolutional coding has a shortfall which is computational complexity that increases exponentially as the length of the code K increase.

Now taking a closer look at the comparison of the two error detection and reduction coding scheme, in **Figure 4.3**. The performance of the coding algorithms can be compared, the Covolutional coding scheme performs much better than linear block coding from low SNR to high SNR, making it more power efficient, however what Convolutional coding gains in power efficiency it losses in computational complexity.

So depending on the application, one needs to consider these characteristics.

6 Conclusion

Linear block coding is better when processing data that is not coming in large streams, such as sensors networks.Linear block coding as a simple implementation and easy, However doesn't not improve energy efficiency and doesn't remove errors at a significant amount.

Convolutional coding is better for transmission and reception of very large data streams. Convolutional coding has more of a energy efficiency than linear block codes for large data , However Convultional coding has a shortfall which is computational complexity that increases exponentially in the length of the code K.

In conclusion the experiment was conducted successfully, and performance of the different coding schemes in comparison was obtained and characteristics of the coding algorithms were modeled.

7 References

[1] B. Wichmann and D. Hill, “Building a random number generator”, Byte, pp 127-128, March 1987.

[2] G. Marsaglia and T.A. Bray, “A convenient method for generating normal variables”, SIAM Rev., Vol. 6, pp 260-264, 1964.

[3] A.Grami,”Baseband digital transmission, ”Introduction to digital communication”,(2016), pp 257-293

8 Appendix A

8.1 Code

8.1.1 Linear block codes

```
# -*- coding: utf-8 -*-
"""
Created on Thu Nov 12 13:09:12 2020

@author: user
"""
import numpy as np
from itertools import combinations
from operator import add
import math
import matplotlib.pyplot as plt

#    ------------------------------------------------------------------------

#                             Varivables
#    ------------------------------------------------------------------------

size= 100


G_1= [[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0],
      [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1],
      [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0],
      [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1],
      [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0],
      [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1],
      [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1],
      [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1],
      [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1]]

#G_1 = np.array(G_1).reshape((-1, 14)).tolist()
#s_1= BPSK(bitgenerator(size))
#s_1 = [1, 0, 0, 1, 0, 1, 1, 1]

#    ------------------------------------------------------------------------

#                             Bit generator
#    ------------------------------------------------------------------------


def bitgenerator(size):

    return  [np.random.randint(2) for i in range(size)]

#    ------------------------------------------------------------------------
```

```
#                            BPSK generator
#
    ------------------------------------------------------------------------

def BPSK(bits):
    bpsk = []
    for k in bits:
        if k == 1:
            bpsk.append(1)
        else:
            bpsk.append(-1)
    return bpsk

#
    ------------------------------------------------------------------------

#                             Linear block code
#
    ------------------------------------------------------------------------


#
    ------------------------------------------------------------------------

#                            Parity check matrix (H)
#
    ------------------------------------------------------------------------


def  parityCheck( G):
    G=np.array(G)
    p= G[0:G.shape[0],G.shape[0]:G.shape[1]]

    H= np.concatenate((p.transpose(),np.identity(G.shape[1]-G.shape
    [0])),1)

    return H

#
    ------------------------------------------------------------------------

#                            Encoding
#
    ------------------------------------------------------------------------


def codeword(sent,G):#sent is a list , G is a list, n is noise
    G=np.array(G)
    s=np.array(sent).reshape((-1,G.shape[0]))
    codewords= s.dot(G) %2
    return codewords.flatten()


#
    ------------------------------------------------------------------------
```

```
#                              Decoding
#
    ------------------------------------------------------------------------


def  decoder(Recievedbits,G): #G is a list and Rec is a list too
    G=np.array(G)
    c=np.array(Recievedbits).reshape((-1,G.shape[1]))
    H=parityCheck(G)
    z=c.dot(H.T)%2

    for k in range(0,len(c)):
        if(sum(z[k]>=1)):
            index=[]
            for i in H.T: # Search for the syndrom Vec in H.T matrix
                if( (i == z[k]).all()):
                    index.append(1)
                else:
                    index.append(0)
            #check if z was found in the H.T matrix
            if(sum(index)>0): #found something change  cest
                c[k] ^= np.array(index) #xor

            else: #nothing was found so find combinations
                for com in   combinations(range(len(H.T)),2):
                    OR = H.T[com[0]].astype(int)^H.T[com[1]].astype(
    int)
                    if(OR==z[k]).all():
                        c[k][com[0]] ^= 1
                        c[k][com[1]] ^= 1
                        break
    return c[:,:G.shape[0]].flatten()
#
    ------------------------------------------------------------------------


def Add_noise(transmitted,RC, SNR):
    M=2
    Gnoise= np.random.normal(0,size=len(transmitted))
    gama = 1 / np.sqrt(math.pow(10, (SNR / 10)) * 2 * RC)
    # print(gama)
    new = [i * gama for i in Gnoise]
    R = list(map(add, transmitted, new))
    return R

#k=[1,-1,1,1]


#
    ------------------------------------------------------------------------

#                              Detection
#
    ------------------------------------------------------------------------


def BPSKDetection(comp):
    points = [-1, 1] #sybols
```

```
    Bpoints = [[0], [1]]
    recieved = -1
    minDistance = 99
    decoded = []
    Bdecoded = []
    for y in comp:
        if(y>0):
            decoded.append(1)
            Bdecoded.append(1)
        else:
            decoded.append(-1)
            Bdecoded.append(0)

    return decoded, Bdecoded  # recieved


#    ------------------------------------------------------------------------
#                       Bit Error calculation
#    ------------------------------------------------------------------------


def bit_errors(sent, recieved):
    error = 0
    for k in range(0,len(recieved)):
        if sent[k] != recieved[k]:
            error += 1
    BER = error / len(recieved)*100

    return BER



s_1= bitgenerator(100)
def tester():

    xValues = np.linspace(-4, 15, 100)
    yvalues=[]
    yvalues1=[]

    for x in xValues:
        ber=0
        ber1=0
        for i in range(0,10):
            s_1= bitgenerator(100)
            symbol,bits=BPSKDetection(Add_noise(BPSK(codeword(s_1,G_1)
    .tolist()),0.5,x))
            symbol1,bits1=BPSKDetection(Add_noise(BPSK(s_1),1,x))
        #bits = [item for sublist in bits for item in sublist]
        #print(Add_noise(codeword(s_1,G_1).tolist(),0.5,1000))
        #print(bits)
        #c_est = [1,0,0,0,0,1,1,1,1,0,1,0,1,0]
            ber+= bit_errors(s_1,decoder(bits,G_1).tolist())
            ber1+= bit_errors(s_1,bits1)

```

```
        ber=ber/10
        ber1=ber1/10
        yvalues.append(ber)
        yvalues1.append(ber1)
    plt.semilogy(xValues,yvalues, label="Linear Block Code")
    plt.ylabel('BER')
    plt.xlabel('SNR (dB)')

    plt.semilogy(xValues,yvalues1, label="Optimal detection uncoded")




    #At the end
    plt.title(" BER vs SNR")
    plt.legend()

tester()
```

8.1.2 Convolution encoder

```
# -*- coding: utf-8 -*-
"""
Created on Thu Nov 12 17:56:06 2020

@author: user
"""

import numpy as np
from copy import deepcopy
from collections import deque
import matplotlib.pyplot as plt

from LBC import bitgenerator,BPSK,Add_noise,BPSKDetection,bit_errors,
    decoder,decoder

K=3

class MNode:
    def __init__(self, status):
        self.status = status
        self.heuristic = None
        self.Predecessor = None


class Viterbi_MLSE:
    def __init__(self, received, G):
        self.K = len(G)
        self.received = np.array(received).reshape((-1, self.K))
        self.BlockN = len(self.received)
        self.Root = None

    def Surving_path(self, lists, new_item):
        for item in lists:
            if new_item.status == item.status:
                if new_item.heuristic > item.heuristic:
                    return False
                else:
                    lists.remove(item)
                    lists.append(new_item)
                    return True

        lists.append(new_item)
        return True


    def Gen( self,val):
        val.reverse()
        c1= val[0]
        c2= val[0]^val[2]
        c3= val[0]^val[1]

        return [c1,c2,c3]

    def ShiftRegister(self, x_state, m):
        status = deepcopy(x_state)
```

```
        status.pop()
        status = list(reversed(status))
        status.append(m)
        return list(reversed(status))



    def MLSE_Decode(self):
        bits = [1, 0]
        self.Root =  MNode([0,0] )
        self.Root.heuristic=0.0

        New_list = []# deque()
        New_list.append(self.Root)
        tail = None


        for t in range(self.BlockN):
            Old_list = deepcopy(New_list)
            New_list.clear()

            if t <= self.BlockN - (self.K - 1):
                while len(Old_list) > 0:
                    T_root = Old_list.pop()
                    for bit in bits:
                        T_node = MNode(self.ShiftRegister(T_root.
status, bit))
                        T_node.Predecessor = T_root
                        heuristic = self.calculate_cost(self.received[
t],
                                                        self.Gen(T_node.
status + [T_root.status[1]]))

                        T_node.heuristic = heuristic + T_root.
heuristic

                        self.Surving_path(New_list, T_node)
            else:
                while len(Old_list) > 0:
                    T_root = Old_list.pop()
                    # shift in 0
                    T_node = MNode(self.ShiftRegister(T_root.status,
0))
                    T_node.Predecessor = T_root
                    heuristic = self.calculate_cost(self.received[t],
                                                    self.Gen(T_node.status
+ [T_root.status[1]]))

                    T_node.heuristic = heuristic + T_root.heuristic

                    if self.Surving_path(New_list, T_node):
                        tail = T_root



```

```
        temp_tail = tail
        arr1 = []


        while temp_tail is not None:
            arr1.append(temp_tail.status[0])
            temp_tail = temp_tail.Predecessor
        send = list(reversed(arr1))

        return [0] +send + [0]

    def calculate_cost(self, conv, conv_est):
        conv_est = 2 * np.array(conv_est) - 1
        #conv = 2* np.array(conv)-1
        delta = list(map(abs,conv - conv_est))
        delta = np.power(np.array(delta),2)
        return delta.sum()









#    ------------------------------------------------------------------------------



def Gen( val):
    val.reverse()
    c1= val[0]
    c2= val[0]^val[2]
    c3= val[0]^val[1]

    return [c1,c2,c3]

def encoder(sentbits): #this function is used for encoding
    codeword=[]
    for k in range(len(sentbits)-2):
        val= sentbits[k:k+3] #reads 3 bits at a time
        codeword+=Gen(val)

    return codeword


s_1= bitgenerator(100)
def tester():

    xValues = np.linspace(-4, 15, 30)
    yvalues=[]
    yvalues1=[]

    for x in xValues:
        ber=0
        ber1=0
```

```
        for i in range(0,10):
            s_1= bitgenerator(100)
            symbol,bits=BPSKDetection(Add_noise(BPSK(encoder(s_1))
    ,0.5,x))
            symbol1,bits1=BPSKDetection(Add_noise(BPSK(s_1),1,x))
            Q= Viterbi_MLSE(symbol,[4,5,6])

            ber+= bit_errors(s_1,Q.MLSE_Decode())
            ber1+= bit_errors(s_1,bits1)

        ber=ber/10
        ber1=ber1/10
        yvalues.append(ber)
        yvalues1.append(ber1)
    plt.semilogy(xValues,yvalues, label="Linear Block Code")
    plt.ylabel('BER')
    plt.xlabel('SNR (dB)')

    plt.semilogy(xValues,yvalues1, label="Optimal detection uncoded")




    #At the end
    plt.title(" BER vs SNR")
    plt.legend()

tester()
```