Stellenbosch University

Department of Electrical and Electronic Engineering

Project (E) 448

Forward Error Correction code construction based on FIR and IIR Filters

Author:
Benjamin Marthinus Du Preez
18486460

UNIVERSITEIT
STELLENBOSCH
UNIVERSITY

Report submitted in partial fulfillment of the requirements of the module Project (E) 448 for the degree Baccalaureus in Engineering in the Department of Electrical and Electronic Engineering at the University of Stellenbosch

Study leader: Prof Jaco Versfeld

Date: October 2017

Acknowledgements

I would like to express my sincere gratitude to the following people:

- Prof Jaco Versfeld, my study leader, for his expert support and for, literally, always having his door open for advice even during this very busy year for him. I would not have been able to complete this project without him.
- Johan van Dyl du Preez, my father, for being a real role model and for his wisdom and advice over the last four years.
- Leentjie du Preez, my mother, for her undying support throughout my undergraduate studies and for proofreading all my reports no matter how uninteresting they might have been to her.
- Eendrag Men's Residence and the friends that I made there for allowing me to walk away from Stellenbosch University with so much more than just a degree.

Plagiarism declaration

BYLAE 1

UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

Plagiaatverklaring / Plagiarism *Declaration*

1 Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.

2 Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
I agree that plagiarism is a punishable offence because it constitutes theft.

3 Ek verstaan ook dat direkte vertalings plagiaat is.
I also understand that direct translations are plagiarism.

4 Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.

5 Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

Studentenommer / *Student number*	**Handtekening / *Signature***
Voorletters en van / *Initials and surname*	**Datum / *Date***

Abstract

It is well known that the construction of some linear block codes can be represented by FIR filters, notably cyclic codes. This project furthers this research at the intersection of the fields of information theory and digital signal processing. Binary linear block code construction with both FIR and IIR filters is investigated. The largest minimum Hamming distance achievable by these codes is found for various short code lengths and dimensions. The results show that the largest minimum distances achieved by these codes are equal to the largest known minimum distances achievable by binary linear block codes in the vast majority of the cases considered. We present an algorithm to calculate the transfer function of IIR filters that construct the same binary code as any specified FIR filter. Finally, an IIR filter structure is found that can construct optimal binary codes for $k = 2$ and seemingly any code length.

Samevatting

Dit is 'n welbekende feit dat die konstruksie van sekere lineêre blokkodes, soos sikliese kodes, voorgestel kan word deur middel van FIR (eindige impulsrepons) filters. Hierdie projek voeg verder toe tot hierdie navorsing wat lê by die snypunt van die velde van informasieteorie en klassieke digitale seinprosessering. Binêre lineêre blokkode konstruksie met beide FIR en IIR (oneindige impulsrepons) filters word ondersoek. Die grootste minimum Hamming afstand bereikbaar deur dié kodes word gevind vir 'n verskeidenheid kodelengtes en dimensies. Die resultate dui aan dat grootste minimum afstand deur hierdie kodes bereik gelyk is aan die grootste bekende bereikbare minimum afstand vir binêre kodes in die oorgrote meerderheid van gevalle. 'n Nuwe algoritme is ontwikkel om die oordragsfunksies van IIR filters te bereken wat dieselfde binêre kodes konstrueer as 'n gegewe FIR filter. Laastens, word 'n IIR filter gevind wat optimale binêre kodes kan konstrueer vir $k = 2$ en skynbaar enige kodelengte.

Contents

List of Figures

List of Tables

List of acronyms

DTFT	Discrete-time Fourier transform
FEC	Forward error correction
FIR	Finite impulse response
IIR	Infinite impulse response
LCCDE	Linear constant coefficient difference equations
LTI	Linear time-invariant
MLE	Maximum likelihood estimation

Chapter 1

Introduction

1.1 Background

Forward error correction (also referred to as channel coding) forms part of the broader fields of coding and information theory. Other areas of research related to FEC are cryptography and data compression. Error correction was largely pioneered by Richard W. Hamming. In his landmark 1950 paper "Error Detecting and Error Correcting Codes" [1], Hamming introduced the first error-correcting code: the (7, 4) Hamming code.

Error-correcting codes work on the principle of introducing a pattern of redundant bits to a message for transmission. A decoder at the receiver then attempts to recover the original message. This is the opposite of compression (source coding) algorithms, which attempt to shrink data size by removing redundancy. The choice of the redundant bits introduced depends on both the message bits and the encoding algorithm used. These redundant bits encode information about the message in such a way that the decoder could still successfully recover the original message when up to a certain number of errors have been introduced by a noisy channel (typically bit-flips). So, in other words, with FEC errors are not just detected, up to a certain number of them can be corrected as well and therefore re-transmission of the original signal isn't required. A more technical explanation of FEC and an overview of the relevant concepts required to understand this work follow in Chapter 2.

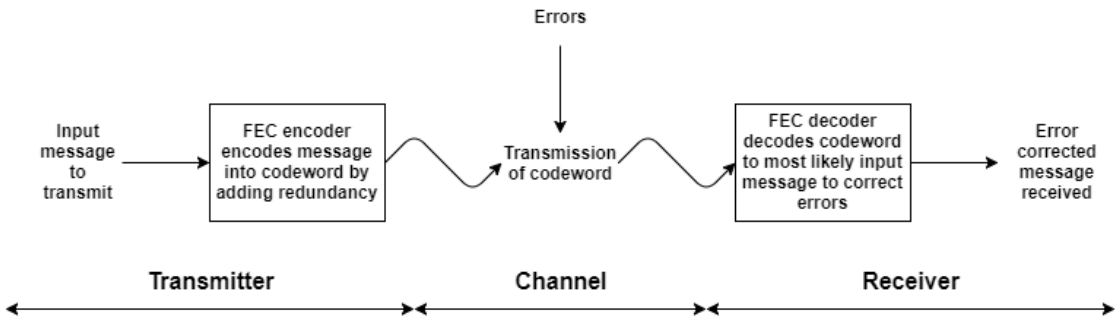

Figure 1.1: Illustration of the FEC concept

Since they rely on introducing redundancy, FEC codes effectively sacrifice transmission rate for higher accuracy. As explained by Hamming [1]: the redundancy of the code can be defined as the ratio $\frac{n}{k}$, where n is the number of symbols in the encoded codeword and k is the number of symbols in the original message. The inverse ratio, $\frac{k}{n}$, is known as the code rate [14] and gives the ratio of useful information bits to the total number of bits transmitted. A higher redundancy, while ideally increasing the number of errors that can be corrected, clearly implies a lower code rate, which results in lower efficiency since more bits need to be transmitted to get the same message across. The tradeoff between

accuracy and efficiency always has to be considered when working with codes in the context of the application. One example of an application for which FEC is highly suited is deep-space exploration. The emergence of forward error correction greatly facilitated space communications - so much so that the mathematician James L. Massey called the combination of deep-space exploration and channel coding "a marriage made in heaven" [2]. The vast distances and harsh conditions encountered in outer space would have made missions such as Voyager near impossible without forward error correction. The new greater tolerance for errors also allowed spacecraft to use weaker antennas, which saved large amounts of money in manufacturing [2].

The core problem in forward error correction is finding codes that maximise error correcting capabilities while minimising the redundancy that needs to be introduced. The ultimate goal is to reach the Shannon limit, which states the upper bound of code performance derived by Shannon in 1948 [3], but gives no information on how to construct codes reaching the limit. This has spawned years of research into FEC code construction, which has led to the discovery of a large variety of error correcting code types. This project is also one such investigation into code construction.

1.2 Project aims

This work is an investigation into using concepts from classical digital signal processing, namely FIR and IIR filters, to construct binary error correcting codes, i.e. using FIR and IIR filters as the encoder of the code. The primary objectives of this project are the following:

> Use FIR and IIR filters to create new FEC codes. Investigate the length of the codes achieved, as well as the largest minimum Hamming Distance achieved for a given code length n and dimension k.

Minimum Hamming distance is a measure of a code's error correcting capability that is introduced in the next chapter.

Investigation of the use of filters for code construction is motivated by the fact that some well known existing code types, such as cyclic codes, can be thought of as FIR filters. A 2016 Master's thesis from the University of the Witwatersrand went further and investigated IIR filters, by looking at non-binary block code construction with IIR filters in GF(4) [4]. This project focuses on binary, i.e. GF(2), block code construction. Also investigated here is the link between the codes created by FIR and IIR filters and the possibility of filter structures that create multiple good codes. Puncturing of the codes produced by the filters is not considered.

1.3 Structure of the report

Chapter 2 introduces all the concepts necessary to understand the work done in this project. The sub-steps that need to be completed to achieve the primary objectives are introduced and discussed in Chapter 3. Chapter 4 discusses the results achieved. Finally, the conclusions of this work are made in Chapter 5.

Chapter 2

Literature review

This chapter gives an overview of the background knowledge necessary to understand the work done in this project. Since the project attempts to combine two major knowledge areas - classical digital signal processing and error correction codes - the literature study is also divided into two topics. The first section, on digital filters, reviews the relevant digital signal processing techniques. This section is very brief, since the work discussed here is typically covered at undergraduate level. The last two sections cover error correcting codes and the necessary mathematical background required to understand them. These sections are more extensive, since error correcting codes aren't typically covered in detail at an undergraduate level in engineering courses.

2.1 LTI digital filters

Linear time-invariant (LTI) systems map, or transform, a certain input signal to an output signal. LTI systems are called linear because they obey the principle of superposition, and time-invariant because the way they map the input to the output does not vary with time. They are absolutely fundamental to electrical engineering: electrical circuits built from resistors, capacitors and inductors are a good example of a continuous-time LTI system [5]. Both continuous and discrete-time LTI systems can be described fully by its impulse response. This project is only concerned with discrete time systems. In the frequency domain, LTI systems transform (or filter) the spectrum of an input signal by attenuating certain frequency components. The terms filter and LTI system are therefore often used interchangeably.

2.1.1 Mathematical background

In the time domain a digital filter is characterised by discrete convolution:

$$y[n] = h[n] * x[n] = \sum_{i=-\infty}^{\infty} h[i] \cdot x[n-i], \tag{2.1}$$

where $x[n]$ is any input sequence (signal), $h[n]$ is the system impulse response and $y[n]$ is the output sequence. If $x[n]$ has length N, and $h[n]$ has length M, then $y[n]$ has length $L = N + M - 1$.

If the system output depends only on past and present inputs, it is referred to as causal. For causal systems the impulse response $h[n]$ is zero for $n < 0$. If the input is also zero for $n < 0$, (2.1) simplifies to:

$$y[n] = \sum_{i=0}^{n} h[i] \cdot x[n-i]. \tag{2.2}$$

In real-time applications non-causal filters are not physically realizable. This project deals almost exclusively with causal filters.

The discrete convolution operation can also be written as a matrix multiplication, $\mathbf{y} = \mathbf{x}A$, where $\mathbf{y}$ and $\mathbf{x}$ are vectors representing the input and output of the system, respectively. A is a Toeplitz (diagonal-constant) matrix containing the impulse response coefficients. If the input again has length N and the impulse response has length M, A is given by:

$$A = \begin{bmatrix} h_0 & h_1 & h_2 & \cdot & \cdot & \cdot & \cdot & \cdot & h_{M-1} & 0 & 0 & 0 & \cdot & \cdot & 0 \\ 0 & h_0 & h_1 & h_2 & \cdot & \cdot & \cdot & \cdot & \cdot & h_{M-1} & 0 & 0 & \cdot & \cdot & 0 \\ 0 & 0 & h_0 & h_1 & h_2 & \cdot & \cdot & \cdot & \cdot & \cdot & h_{M-1} & 0 & \cdot & \cdot & 0 \\ \vdots & & & & & & & & & & & & & & \vdots \\ 0 & 0 & \cdot & \cdot & \cdot & 0 & h_0 & h_1 & h_2 & \cdot & \cdot & \cdot & \cdot & \cdot & h_{M-1} \end{bmatrix}.$$

A has the dimensions $(N \times L)$, where the width of the matrix, L, is the output length. The full length of L is equal to $N + M - 1$. This formulation of discrete convolution is important, because it describes the filtering operation in a form similar to multiplication with a generator matrix: a fundamental error correction concept that will be discussed later. An important observation that can now be made, is that only the first L samples of the impulse response, at most, can appear in the matrix. In other words, the first k samples of the filter output are only determined by the first k samples of the impulse response. This observation will be used again later in this project. Finally, a third way to formulate discrete convolution is with polynomial multiplication. This will be introduced along with cyclic codes in Section 2.3.3.

Also relevant is the z-transform, which can be viewed as a discrete-time version of the Laplace transform. The z-transform transforms a time-domain input sequence onto a complex frequency plane (the z-plane), as a function of the complex variable z:

$$H(z) = \sum_{n=-\infty}^{\infty} h[n] \cdot z^{-n}, \tag{2.3}$$

where $z = r \cdot e^{j\omega}$. It is good to note that when the value of $r = 1$, the z-transform reduces to the discrete-time Fourier transform (DTFT), which gives the frequency response of the sequence. In other words: the magnitude of the frequency response of a sequence is equivalent to the magnitude along the unit circle in the z-plane. Compare this to how the frequency response of a continuous-time system is equivalent to the Laplace transform evaluated along the imaginary axis in the s-plane (i.e. $s = j\omega$). Another very useful property of the z-transform is that multiplication in the z-domain is equivalent to discrete convolution in the time domain:

$$Y(z) = H(z) \cdot X(z), \tag{2.4}$$

where $H(z)$ is the transfer function: the z-transform of the impulse response $h[n]$. So, if the system transfer function is known, the system output can be computed by simple multiplication.

Note from (2.2) that the term z^{-n} captures time information, i.e. the coefficients of the polynomial $H(z)$ produced by the z-transform are the values of the original discrete-time sequence $h[n]$. This clearly illustrates the time shifting property of the z-transform,

which is defined below:

$$\mathcal{Z}\{x[n-k]\} = z^{-k} \cdot \mathcal{Z}\{x[n]\}. \tag{2.5}$$

An important class of LTI systems in which we are interested can be represented using linear constant-coefficient difference equations (LCCDEs) [6]:

$$y[n] = -\sum_{k=1}^{N} a_k \cdot y[n-k] + \sum_{k=0}^{M} b_k \cdot x[n-k]. \tag{2.6}$$

Taking the z-transform of both sides of this, using (2.5), and then rearranging, yields:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k \cdot z^{-k}}{1 + \sum_{k=1}^{N} a_k \cdot z^{-k}}. \tag{2.7}$$

This gives us the standard form for the transfer function of a filter, which is used throughout this report. The numerator describes the location of poles in the z-plane, while the denominator describes the locations of zeros. When we write filters in this form, we can describe the filter completely in terms of its a and b coefficients, and indeed software signal processing packages typically accept arrays of b and a coefficients as the only parameters for their filtering functions. Next we look at the two categories that discrete time filters can be divided into: FIR and IIR filters.

2.1.2 FIR filters

Finite impulse response (FIR) filters are filters whose impulse response are non-zero over a finite interval only. In other words: their impulse response $h[n]$ is a finite-length sequence. A FIR filter is an all-zero filter [7]. Since it only has zeros, we can see from Equation 2.7, that its transfer function simplifies to the form:

$$H(z) = \sum_{k=0}^{M} b_k \cdot z^{-k}. \tag{2.8}$$

When comparing this with the definition of the z-transform (2.3), it is clear that $h[k] = b_k$ for a FIR filter, i.e. the impulse response sequence has the coefficients of the transfer function as values:

$$h[k] = \{b_0, b_1, b_2, ..., b_M\}.$$

The finite length of a FIR filter means that the discrete convolution sum (2.1) becomes finite, so it can be implemented directly if M memory locations are available. This is exactly the same as simply setting $a_k = 0$ for all values of k in (2.6) and then implementing the LCCDE.

2.1.3 IIR filters

Infinite impulse response (IIR) filters are filters whose impulse response are non-zero over an infinite range. In other words: their impulse response $h[n]$ has infinite length. The transfer function of an IIR filter contains poles and can be expressed in the full form of (2.7).

Since the impulse response isn't finite, direct implementation of the discrete convolution

sum (2.1) is not possible, because it would require infinite memory. We can, however, solve this problem by using recursion - we can implement IIR filters by using an LCCDE.

Analog filters are IIR filters - their response might attenuate towards zero over time, but it isn't truly finite. Design techniques for analog filters are well established and extensively researched [7]. These readily available techniques can be used to design a continuous-time IIR filter first, before transforming it to a discrete-time IIR filter via a technique such as the bilinear transform. Finally, if desired, the IIR filter can be transformed into a FIR filter by windowing it in the time domain to make the impulse response finite.

2.1.4 Software realization of FIR and IIR filters

IIR and FIR filters can be realized in software using a block diagram of the LCCDE. Various configurations exist for the implementation of the same transfer function, notably direct form I and direct form II. The MATLAB signal processing toolbox and Python's Scipy Signal package use the direct form II transposed implementation of the LCCDE [8][9]. Figure 2.1 illustrates direct form II transposed. Keep in mind that z^{-1} represents a one sample time delay (as shown in Equation 2.5).

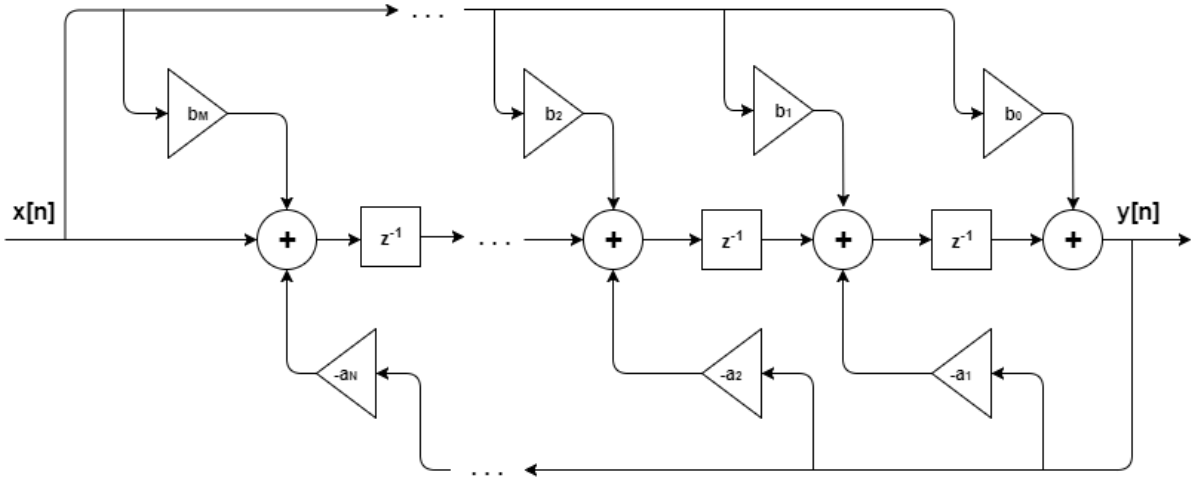

Figure 2.1: Direct Form II transposed implementation of a digital filter

2.2 Abstract algebra

Understanding FEC requires knowledge of some fundamental concepts from abstract algebra, that are not typically covered in undergraduate engineering courses. Three such concepts are relevant: groups, fields and rings. Another relevant abstract algebra concept is vector spaces, but most readers should already be familiar with this. These concepts are relevant for FEC since we often work with sets that only contain a finite number of elements.

2.2.1 Groups

A group is a set of elements for which one operation is defined, under which the group is closed (i.e. if two elements in the group are combined using this operation, the result

is also in the group). Each element has an inverse element - if it is combined with this inverse element using the operation the result is an element known as the identity element. Groups are associative, but not necessarily commutative. Commutative groups are known as Abelian groups.

2.2.2 Fields and Finite fields

A field is similar to a group, but less general. It is a set of elements for which addition and multiplication is defined as operations. Under either addition or multiplication a field is a commutative group. This means that, in fields, addition of an element with its inverse element (subtraction) or multiplication of an element with it's non-zero inverse element (division) results in an identity element. Furthermore, these operations are commutative. So, effectively, four operations are defined for fields: addition, subtraction, multiplication and division. The field is closed under all these operations. Both the associative and distributive properties hold for fields. An example of a field is the set of all real numbers $\mathbb{R}$.

A finite field, or Galois field, is a field that only contains a finite number of elements. It is written using the notation for a Galois field or as $\mathbb{F}_n$, where $\mathbb{F}$ represents the field the elements are from and n the number of elements in the finite field. In general a Galois field is written as GF(p^n), where p is a prime number. A Galois field of the form GF(p), i.e. $n = 1$, is called a prime field and contains the integers 0, ..., $p-1$. In a prime field any operation is equal to the usual arithmetic operations one would perform on integers modulo p. This ensures that the field is closed. For example in GF(3): $2 + 2 = 4 \bmod 3 = 1$. When working with binary error correcting codes where each element is only 1 bit, one is working in the prime field GF(2) (also denoted as $\mathbb{Z}_2$), which is the field containing only two elements - the integers 0 and 1. Therefore, GF(2) arithmetic will be used for the remainder of this report. For this, addition and subtraction is conveniently the same as XOR, while multiplication and division is equal to AND. A summary of the basic operations are given in Table 2.1

Operation	With modular arithmetic	Programming equivalent
$a + b$	$(a + b)mod(2)$	$a \oplus b = a$ XOR b
$a - b$	$(a + (-b))mod(2) = (a + b)mod(2)$	$a \oplus b = a$ XOR b
$a \times b$	$(a \times b)mod(2)$	$a \cdot b = a$ AND b
$a \div b, b \neq 0$	$(a \times \frac{1}{b})mod(2) = (a \times b)mod(2)$	$a \cdot b = a$ AND b

Table 2.1: Simple arithmetic operations in GF(2)

In coding theory, as will be mentioned in Section 2.3, codes are usually represented by vectors. Polynomials can be used as an alternative to vectors, where the polynomial coefficients are elements of the finite field. For example, consider the binary codeword $\mathbf{a}$ of length n in the vector space $\mathbb{Z}_2^n$ and its alternate representation as a polynomial:

$$\mathbf{a} = (a_0,\ a_1,\ \ldots\ ,\ a_{n-1}) \rightarrow a(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}. \tag{2.9}$$

The addition and multiplication of polynomials in finite fields are the same as usual, except that the resulting coefficients are always in the field, i.e. in GF(2) coefficients are taken

modulo 2 to ensure that the field is closed. If a polynomial has no roots in a finite field it is called irreducible over that field. It can be shown that any irreducible polynomials over GF(2) of degree m can divide $(x^{2^m-1}+1)$ [11]. If an irreducible polynomial also doesn't divide (x^n+1) with n smaller than 2^m-1, it is called a primitive polynomial.

2.2.3 Vector spaces

Recall that the notation $\mathbb{F}^n$ is used for a vector space that consists of n-dimensional vectors over the field $\mathbb{F}$. When $\mathbb{F}$ also a finite field, the aforementioned notation is combined with the notation for finite fields. For example, the vector space containing binary vectors of length n is represented as $\mathbb{Z}_2^n$.

2.2.4 Rings and Polynomial rings

Rings are similar to fields. The only major difference is that the multiplication may not be commutative and inverse of multiplication (division) need not be defined. In other words: a ring is not closed under division and the result of such an operation might therefore not be an element of the ring. The other properties of rings are the same as fields. The set of all integers $\mathbb{Z}$ is an example of a ring (not a field, since $\mathbb{Z}$ is not closed under division).

A polynomial ring $F[x]/(x^n+1)$ is the set of polynomials forming a ring, which has a degree less than n and with coefficients that are elements of the field F. For binary codes this implies a set of 2^n polynomials. Polynomial rings are important for the mathematical description of cyclic codes - which will be discussed in Section 2.3.3. The result of operations on polynomial rings are taken mod (x^n+1), to ensure that the ring is closed.

2.3 Forward error correction

Lin and Costello explain the mathematical idea behind FEC well in their textbook *Error Control Coding* [11]. A message that has to be transmitted across a channel is defined as a row vector of length k:

$$\mathbf{m} = (m_1,\ m_2,\ m_3,\ \ldots\ ,\ m_k),$$

or using matrix notation:

$$\mathbf{m} = [m_1\ m_2\ m_3\ \ldots\ m_k].$$

For a binary message this vector is in the vector space $\mathbb{Z}_2^k$, i.e. it contains k elements and it consists only of 0s and 1s. There are therefore 2^k unique messages that can be represented by $\mathbf{m}$. An encoder then adds $n-k$ redundant bits, known as parity check bits [14], to transform $\mathbf{m}$ into $\mathbf{c}$, which is a new, longer, vector of length n in $\mathbb{Z}_2^n$, referred to as a codeword:

$$\mathbf{c} = (c_1,\ c_2,\ c_3,\ \ldots\ ,\ c_n),\ n > k.$$

The codeword is the data transmitted across the channel, where it might be corrupted by noise, and is then decoded at the receiver to attempt to recover the original message. Since the codeword is of length n there are 2^n possible vectors that could arrive at the receiver, but because each of the 2^k possible unique messages is encoded into only one unique corresponding codeword there are only 2^k valid codewords. The set of all valid codewords, a matrix denoted by C and referred to as the codebook, is therefore a k-dimensional subspace of $\mathbb{Z}_2^n$. This implies there are 2^n-2^k unique vectors that could be

received which is not a valid codeword. The fact that there are more unique combinations of possible received data than unique valid codewords is exactly where the error-correcting capability of a code lies: if any other data arrives at the receiver than a valid codeword, we know an error occurred, the decoder assumes that the valid codeword the most similar to the received data was sent and then recovers the original message corresponding to that valid codeword. Note that if a message contains errors in such a way that a wrong, but valid, codeword was received, the errors can't be detected or corrected. This is called an undetectable error. Figure 2.2 visually summarises the simple but powerful FEC concept explained in the previous few paragraphs.

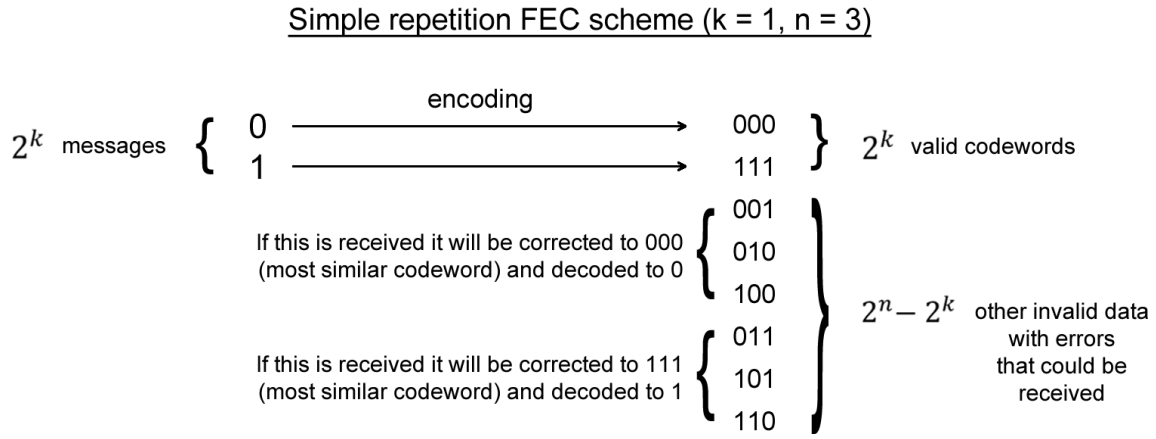

Figure 2.2: Illustration of the working of a simple FEC scheme

There are two major classes of FEC codes: block codes and convolutional codes [14].

2.3.1 FEC definitions

We define the following concepts, used in the subsequent subsections.

Definition 2.1 The Hamming distance $d(\mathbf{a}, \mathbf{b})$ between $\mathbf{a}$ and $\mathbf{b}$, is the number of components in which they are different [11].

Hamming distance is a measure of how similar different data vectors are - when correcting errors, invalid data is corrected to the codeword with the smallest relative Hamming distance.

Definition 2.2 The Hamming weight $w(\mathbf{a})$ of $\mathbf{a}$, is the number of non-zero components of $\mathbf{a}$ [11].

Note that $d(\mathbf{a}, \mathbf{b}) = w(\mathbf{a} + \mathbf{b})$

Definition 2.3 Minimum Hamming distance d_{min} is the smallest Hamming distance in the code C [11].

$$d_{min} = min\{d(\mathbf{a}, \mathbf{b}) : \mathbf{a}, \mathbf{b} \in C, \mathbf{a} \neq \mathbf{b}\}. \tag{2.10}$$

The minimum Hamming distance is one of the most important parameters of a code - along with it's k and n values. The reason for this is that a code's minimum Hamming

distance determines how many errors it can correct. It can be shown that a code can detect up to $d_{min} - 1$ errors and correct t errors [11][14], where:

$$t = \lfloor \frac{d_{min} - 1}{2} \rfloor. \tag{2.11}$$

For example, refer to the code in Figure 2.2. From inspection one can easily see that it has a minimum Hamming distance of $d_{min} = 3$, and can detect up to 2 errors, but only correctly correct 1 error.

Definition 2.4 Minimum weight w_{min} is the smallest Hamming weight in the code C [11].

$$w_{min} = min\{w(\mathbf{a}) : \mathbf{a} \in C\}. \tag{2.12}$$

2.3.2 Linear block codes

Block codes convert a message sequence of a fixed length k into a codeword of length n. A code is linear if the linear combination of two valid codewords is also a valid codeword [14]. One very useful property of linear block codes is that its minimum Hamming distance is equal to its minimum weight [11], which reduces computational complexity when d_{min} has to be found:

$$d_{min} = w_{min}. \tag{2.13}$$

Encoding

In linear block codes the message is encoded by multiplication with a matrix G, called a generator matrix, to yield a codeword:

$$\mathbf{c} = \mathbf{m}G. \tag{2.14}$$

Recall that $\mathbf{c}$ is a $(1 \times n)$ row vector and $\mathbf{m}$ is a $(1 \times k)$ row vector. G has the dimensions $k \times n$, its rows are linearly independent and form a basis for C [12]. In general there exists several generator matrices for the same code [12], i.e. several G matrices generate the same codebook, only the mapping order from $\mathbf{m}$ to $\mathbf{c}$ changes. Two codes with the same codebook are considered equivalent, no matter the order of the rows in C. A linear block code C is referred to as a (n, k) block code. For block codes we can also define the matrix H, of dimension $(k \times n)$, which is called the parity check matrix of the code. If H is a parity check matrix the following holds:

$$\mathbf{c}H^T = H^T\mathbf{c} = \mathbf{0}, \tag{2.15}$$

where $\mathbf{c}$ is a valid codeword, i.e. $\mathbf{c} \in C$. H can therefore be used to validate that received data is indeed codeword - if the received data (denoted by $\mathbf{r}$) is not a valid codeword, then:

$$\mathbf{r}H^T \neq \mathbf{0}. \tag{2.16}$$

Systematic codes are codes which still contain the original, unaltered, message bits. Their codewords can be partitioned into two parts: one containing the message and one containing the redundant parity bits. This makes them easy to decode. To achieve this their generator matrix is in the standard systematic form:

$$G = [P, I_k], \tag{2.17}$$

where I_k is the $(k \times k)$ identity matrix, which keeps the k original bits in the codeword and P is the $k \times (n-k)$ parity matrix, which is unique to the code and generates the n redundant (parity) bits. Any linear block code can be put into systematic form [13], i.e. G of any linear block code can be written in the standard form.

One very useful property of systematic codes is that their parity check matrix can be constructed from their generator matrix [11] using

$$H = [I_{n-k}, P^T]. \tag{2.18}$$

Decoding

For error correction in a linear block code we correct invalid received data to the closest valid codeword. For this we define $\mathbf{s}$, the syndrome, for a received vector $\mathbf{r}$:

$$\mathbf{s} = \mathbf{r}H^T. \tag{2.19}$$

From (2.12) and (2.13) it is known that an error has occurred (i.e. $\mathbf{r} \notin C$) if $\mathbf{s} \neq \mathbf{0}$. The vector $\mathbf{e}$ containing the errors that occurred is defined as:

$$\mathbf{e} = \mathbf{r} + \mathbf{c}. \tag{2.20}$$

Recall that there is no difference between addition and subtraction in GF(2), therefore also $\mathbf{r} = \mathbf{e} + \mathbf{c}$. This leads to:

$$\mathbf{s} = (\mathbf{e} + \mathbf{c})H^T = \mathbf{e}H^T + \mathbf{0} = \mathbf{e}H^T. \tag{2.21}$$

If the syndrome as defined in (2.19) is not equal to $\mathbf{0}$, then a value of $\mathbf{e}$ is chosen which satisfies (2.21) with the lowest possible $w(\mathbf{e})$. In other words: we assume that the errors which occurred is the difference between $\mathbf{r}$ and the closest valid codeword $\mathbf{c}$. The valid codeword which was transmitted, $\mathbf{v}$, is then given by $\mathbf{v} = \mathbf{e} + \mathbf{r}$. The message represented by $\mathbf{v}$ is then recovered. This is called minimum distance decoding.

These are the fundamentals of the encoding of linear block codes and the principles behind their error correction capabilities. The detail of the decoding algorithms used for block codes will not be explored here, since that is not the focus of this project - this project is about the construction of new codes.

2.3.3 Cyclic codes

Cyclic codes are an important subclass of block codes. Cyclic codes are effective not just for detecting random errors, but also burst errors. Their algebraic properties also ease decoding [11]. Cyclic codes are relevant for this project because, as will be shown, the way they encode data can be thought of as a FIR filter. A linear code is a cyclic code if every cyclic shift of a codeword in C, is also a codeword in C [11].

The polynomial representation of codewords, as shown in Equation 2.9, is used with cyclic codes. If $\mathbf{v} \in C$ then we can write $v(x) \in C$. The polynomial form offers a convenient way to formulate the shifting property of cyclic codes:

$$x^i v(x) = v^{(i)}(x) \in \mathbb{Z}[x]/(x^n + 1). \tag{2.22}$$

Cyclic codes are encoded by multiplying a message polynomial with a generator polynomial to yield a code polynomial:

$$c(x) = (a_0 + a_1 x + ... + a_{k-1}x^{k-1})g(x), \tag{2.23}$$

where the coefficients a_0 to a_{k-1} are $\in \mathbb{Z}_2$.

The generator polynomial is a polynomial that divides $(x^n + 1)$. In fact, it can be shown that all factors of $(x^n + 1)$ of degree $(n - k)$ generate an (n, k) cyclic code. These codes, however, are not all the same - some factors generate better codes than others [11].

The encoding process can also be represented by a generator matrix populated by coefficients of the generator polynomial:

$$G = \begin{bmatrix} g_0 & g_1 & g_2 & \cdot & \cdot & \cdot & \cdot & \cdot & g_{M-1} & 0 & 0 & 0 & \cdot & \cdot & 0 \\ 0 & g_0 & g_1 & g_2 & \cdot & \cdot & \cdot & \cdot & \cdot & g_{M-1} & 0 & 0 & \cdot & \cdot & 0 \\ 0 & 0 & g_0 & g_1 & g_2 & \cdot & \cdot & \cdot & \cdot & \cdot & g_{M-1} & 0 & \cdot & \cdot & 0 \\ \vdots & & & & & & & & & & & & & & \vdots \\ 0 & 0 & \cdot & \cdot & \cdot & 0 & g_0 & g_1 & g_2 & \cdot & \cdot & \cdot & \cdot & \cdot & g_{M-1} \end{bmatrix}.$$

Note that, in general, this matrix can't be partitioned into an identity and parity matrix and it is therefore not systematic. However, as was noted earlier, systematic codes are often desirable. As shown in Lin and Costello [11], a cyclic code of a generator matrix can be rewritten to form a new generator matrix in systematic form, by dividing x^{n-k+1} with $g(x)$ and using the remainder, denoted by $b_i(x)$.

$$x^{n-k+1} = a(x)g(x) + b_i(x) \tag{2.24}$$

The systematic generator matrix can then be formed using $b_i(x)$:

$$G = \begin{bmatrix} b_{00} & b_{01} & b_{02} & \cdots & b_{0,n-k-1} & 1 & 0 & 0 & \cdots & 0 \\ b_{10} & b_{11} & b_{12} & \cdots & b_{1,n-k-1} & 0 & 1 & 0 & \cdots & 0 \\ b_{20} & b_{21} & b_{22} & \cdots & b_{2,n-k-1} & 0 & 0 & 1 & \cdots & 0 \\ \vdots & & & & & & & & & \vdots \\ b_{k-1,0} & b_{k-1,1} & b_{k-1,2} & \cdots & b_{k-1,n-k-1} & 0 & 0 & 0 & \cdots & 1 \end{bmatrix}.$$

Since the generator matrix is now in systematic form, the parity check matrix of the code can also easily be found using Equation 2.18.

The polynomial multiplication during encoding makes cyclic codes analogous to filters, since the discrete convolution of the coefficients of two polynomials yields the coefficients of the polynomial that would result from their multiplication. For example, consider the (7,4) cyclic code with generator polynomial $g(x) = 1 + x + x^3$, encoding the message $m(x) = x^2 + x^3$. The code polynomial is then given by:

$$c(x) = (x^2 + x^3)(1 + x + x^3) = x^2 + x^4 + x^5 + x^6,$$

or, with vector representation, $\mathbf{c} = (0, 0, 1, 0, 1, 1, 1)$. Write $g(x)$ and $m(x)$ as sequences instead of polynomials, i.e. $g[i] = \{1, 1, 0, 1\}$ and $m[i] = \{0, 0, 1, 1\}$. Now, using these sequences, perform discrete convolution (2.1). The result is $c[i] = \{0, 0, 1, 0, 1, 1, 1\}$, which

is the desired codeword. We can therefore consider $g[i]$ an impulse response. By using the z-transform (2.3) the encoding can then be represented by a transfer function. For this example:

$$G(z) = 1 + z + z^{-3}.$$

Comparing with 2.8, it is clear that this is a FIR filter. Another way to convince oneself that the encoding process for cyclic codes is discrete convolution, is by noting from the given shape of the non-systematic generator matrix that it is a Toeplitz matrix. Recall Section 2.1.1, where it was mentioned that this is a way to formulate discrete convolution.

2.3.4 Limits on code performance

The Griesmer and Hamming bounds give an upper limit on the minimum Hamming distance achievable by codes for certain values of n and k. These bounds are achievable by binary codes. The Griesmer bound for binary codes is

$$n \geq \sum_{i=0}^{k-1} \lceil \frac{d}{2^i} \rceil. \tag{2.25}$$

The Hamming bound for binary codes is given by

$$2^k \geq \frac{2^n}{\sum_{i=0}^{t} \binom{n}{i}}, \tag{2.26}$$

where

$$t = \lfloor \frac{d_{min} - 1}{2} \rfloor.$$

2.4 Summary of chapter

This chapter gave a condensed overview of the background to this project. Broadly speaking this chapter looked at two different topics: digital signal processing and forward error correction. Firstly, the relevant digital signal processing techniques were discussed - digital filters along with their mathematical descriptions in the time-domain and z-plane, using discrete convolution and transfer functions, respectively. Also mentioned was the typical implementation of these concepts in software as described by block diagrams. Secondly, an overview was given of FEC coding. After touching on the algebraic background required to understand forward error correction codes, most importantly finite fields, the first subsection gave a broad explanation of the FEC concept. Then some fundamental FEC concepts were defined, before the focus shifted to different types of codes, including cyclic codes. Finally two upper bounds on linear binary codes which we hope to achieve were mentioned.

The next chapter explains the methodology used in the construction of FEC codes with FIR and IIR filters.

Chapter 3

Methodology

This chapter describes the methodology followed to achieve the objectives of this project that were stated in Chapter 1. The first section gives a background on the software development, before the creation of the new FEC codes with filters is covered. Next, a look is taken at the investigation into the codes achieved. Finally the chapter discusses the attempts to find codes with desirable properties and the minimum Hamming distance achievable for given values of n and k.

3.1 Software development

3.1.1 Choice of development platform

MATLAB has a built-in signal processing toolbox, which provides an array of functions to perform tasks such as discrete convolution and signal filtering. The Numpy and Scipy packages for scientific computing provide similar functionality in Python.

One slight problem presents itself when attempting to mix classical signal processing techniques with error correction codes: many standard signal processing libraries are not written for finite fields and of course, as discussed earlier, the math used for binary codes as in this project is all in the Galois field GF(2).

Solving this problem is very straightforward in MATLAB and Octave - since it offers a built-in function which accepts any matrix or vector as parameter and constrains it to a desired Galois field. It also offers a function gffilter() which is specifically designed to filter sequences in Galois fields. In Python, however, there is no easy way to limit arrays to GF(2), so, to achieve the correct results, each element of the output of a signal processing operation has to be taken modulo 2.

A third option is to implement the signal processing functions needed for this project from scratch using GF(2) arithmetic, in an attempt to try and achieve optimal performance. For this purpose the author developed a discrete convolution function and a signal filtering function, which implements Direct Form II using Python with Numpy (code in Appendix D). The filtering functions of the three possible options were then tested with various lengths of input messages and filter coefficients and the respective execution times were recorded. The results can be seen in Table 3.1 and Table 3.2.

As seen the execution time of MATLAB is slower than the Scipy option, but not dramatically so. In addition, MATLAB offers a complete communications toolbox with plenty of built-in functionality for FEC tasks along with good documentation and support. Python does not. This was deemed by the author to count enough in MATLAB's favour to make it the preferred option for use in this project. MATLAB implementations of all the algorithms discussed in this chapter can be found in Appendix D.

	10000 bits	100000 bits	1000000 bits
MATLAB function: gffilter(b, a, x)	0.018 s	0.17 s	1.8 s
Scipy (Python) function: lfilter(b, a, x)mod(2)	0.00098 s	0.004 s	0.034 s
Custom direct Python implementation	0.0286 s	0.271 s	3.22 s

Table 3.1: Approximate execution time of filtering random binary sequences of different lengths with the FIR filter $H(z) = 1 + z^{-1} + z^{-3}$ on the author's computer

	10000 bits	100000 bits	1000000 bits
MATLAB function: gffilter(b, a, x)	0.018 s	0.17 s	1.81 s
Scipy (Python) function: lfilter(b, a, x)mod(2)	0.00098 s	0.0041 s	0.053 s
Custom direct Python implementation	0.042 s	0.4 s	3.77 s

Table 3.2: Approximate execution time of filtering random binary sequences of different lengths with the IIR filter $H(z) = \frac{1+z^{-1}+z^{-3}}{1+z^{-1}+z^{-2}}$ on the author's computer

3.1.2 Signal processing in MATLAB

Recall that if a digital filter whose impulse response is of length M is used to filter an input of length N, the output sequence is of length $L = M + N - 1$. In MATLAB, however, the outputs of the filtering functions filter() and gffilter() are of the same length as the input, N. If the full output of length L is desired, we, therefore, need to pad the input with $M-1$ zeros before passing it to the filtering function. Zero-padding, therefore, allows control over the output length of a filter in MATLAB. Note that, for the case of an IIR filter, the impulse response is infinite ($N \to \infty$), so the output of any input filtered by it should also be of infinite length. In summary: by zero-padding the input sequence in MATLAB, a FIR filter can encode a message into a codeword with a length of up to $L = M + N - 1$, while an IIR filter can encode the same message into a codeword of any desired length.

3.2 Creation of codes

3.2.1 Construction of message book

For any specified value of k there exists 2^k possible binary message vectors. The message book is simply a collection of all of these. First an array was created containing the decimal numbers 0 to $2^k - 1$, each entry was then expanded into a binary vector of length k to yield a $(2^k \times k)$ matrix with message vectors as rows

3.2.2 Construction of codebook

The codebook is the collection of all the valid codewords, i.e. all the encoded messages. Creating the codebook is done by simply encoding each row in the message book of length k to yield a codeword of length n. The encoding is done by filtering the messages with a FIR or IIR filter with a certain combination of coefficients. The final result is a $(2^k \times n)$ matrix.

3.3 Linearity of created codes

All block codes created by FIR or IIR filters are linear. This is not just an empirical observation made by the author, it can also be proven.

Proof. For a binary code to be linear it must satisfy:

$$\mathbf{c}_1 + \mathbf{c}_2 = \mathbf{c}_3, \quad \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3 \in C,\ \mathbb{Z}_2^n. \tag{3.1}$$

Let $T\{\ \}$ represent the encoding performed by a FIR or IIR filter.

$$\begin{aligned} \mathbf{c}_1 &= T\{\mathbf{m}_1\}, \quad \mathbf{c}_1 \in \mathbb{Z}_2^n,\ \mathbf{m}_1 \in \mathbb{Z}_2^k \\ \mathbf{c}_2 &= T\{\mathbf{m}_2\}, \quad \mathbf{c}_1 \in \mathbb{Z}_2^n,\ \mathbf{m}_2 \in \mathbb{Z}_2^k \\ \therefore \mathbf{c}_3 = \mathbf{c}_1 + \mathbf{c}_2 &= T\{\mathbf{m}_1\} + T\{\mathbf{m}_2\} \\ \mathbf{c}_3 &= T\{\mathbf{m}_1 + \mathbf{m}_2\} \end{aligned}$$

Where the final step follows from the fact that LTI systems obey the principle of superposition. By definition all vectors in $\mathbb{Z}_2^k$ are valid message vectors and $\mathbf{m}_1 + \mathbf{m}_2$ is therefore in the message book. In any block code all vectors in the message book are encoded to a corresponding codeword of length n in C. Therefore, since $\mathbf{c}_3$ is formed from a vector in message book, it is not only in $\mathbb{Z}_2^n$, it is also a codeword in C and (3.1) is satisfied. □

This is a very useful property to be aware of, since it can be used to simplify some operations such as the calculation of minimum Hamming distance.

3.4 Constructing a G matrix for created codes

One can construct a generator matrix, G, by choosing k linearly independent rows from the codebook. However, this can become an expensive process for large k values since up to 2^k codewords need to be searched. In the case of this project, even though G is unknown, the filter coefficients used to encode the message book is known, so a second means to encode data exists. Now note that:

$$I_k G = G. \tag{3.2}$$

Since multiplication with the generator matrix is how block codes encode data, this is the same as stating that encoding the identity matrix yields G. The $(k \times n)$ matrix constructed by encoding the rows of a $(k \times k)$ identity matrix via filtering is therefore a generator matrix.

Since the first row of I_k has the form $(1, 0, 0, ..., 0)$, it is an impulse. A second observation that can therefore be made is that the result of filtering the first row of I_k is simply the impulse response of the filter. Since the next $k - 1$ rows in I_k are simply shifted versions of the first, the corresponding rows in G are also just shifted versions of the first (this follows from the time-invariance property of LTI systems). For all codes created by filters, a matrix with k rows that are simply shifted versions of the impulse response is a valid generator matrix. Using this information G can easily be constructed by simply calculating the impulse response once and the codebook need not be searched. This is the approach followed in this project.

Another way to arrive at this exact same result is by simply writing the convolution that the encoding filter performs as matrix multiplication with a $(k \times n)$ Toeplitz matrix (recall Section 2.1.1). This Toeplitz matrix is then a generator matrix.

Finally, since all rows in the G-matrix are also codewords in C for all block codes, the codebook of any code created by a causal filter will also contain k non-circular shifts of the impulse response.

However, this means that G isn't usually in the standard systematic form. Recall that all linear block codes can be put into systematic form (Section 2.3.2). This can be achieved either by performing Gaussian elimination on G or by multiplying G with A^{-1}, where A is the $(k \times k)$ sub-matrix of G that needs to become an identity matrix.

Algorithm 1 Extraction of the generator matrix of a code

procedure GETGMATRIX(b, a, n, k, sys) ▷ *sys* is = true if G should be systematic
  G = a $(k \times n)$ matrix of zeros
  impulseResponse = length n impulse response of filter with coefficients b and a
  currentRow = *impulseResponse*
  for $i = 1$ to k **do**
    ith row of G = *currentRow*
    currentRow = non-circular shift of *currentRow* to the right
  end for
  if *sys* = true **then** ▷ A systematic G is required
    G = result of Gaussian elimination performed on G
  end if
  return G ▷ Return the generator matrix
end procedure

3.5 Investigation of codes achieved

3.5.1 Calculating the minimum Hamming distance

The brute force approach to calculating the minimum Hamming distance of a code involves calculating the Hamming distance between all the codewords in the codebook and then finding the smallest one. However, this requires

$$\binom{2^k}{2} = \frac{2^k!}{2!(2^k-2)!} = \frac{2^k(2^k-1)}{2} \tag{3.3}$$

comparisons, which is obviously problematic when working with large codes. Since it is known that all codes created by FIR and IIR filters are linear, the property of linear block codes defined in (2.13) can be exploited, which means only 2^k comparisons are needed.

3.5.2 Check for bounds achieved

The minimum Hamming distance found is used to check if the (n, k) code achieves either the Griesmer (2.25) or Hamming bound (2.26) for the given values of n and k.

Algorithm 2 Minimum Hamming distance calculation

```
procedure MinHammingDistance(C)
    height = the height of codebook C        ▷ Equal to 2^k, where k is message length
    width = the width of codebook C                          ▷ Codeword length, n
    minDistance = width                          ▷ Initialise to largest possible value
    i = 0
    repeat
        currentDistance = the Hamming weight of the ith row in the codebook
        if currentWeight < minDistance then
            minDistance = currentWeight
        end if
        i = i + 1                                                  ▷ Increment i
    until i = height
    return minDistance                  ▷ Return the minimum Hamming distance
end procedure
```

3.5.3 Check for cyclicity

Recall that the condition for a code to be a cyclic code, is that every cyclic shift of a codeword in C is also a codeword in C. One can therefore test for cyclicity by investigation of the codebook. A pseudocode description of this is given as Algorithm 3.

Algorithm 3 Check for code cyclicity

```
procedure IsCyclic(C)
    height = the height of codebook C        ▷ Equal to 2^k, where k is message length
    isCyclic = true                  ▷ Initialise to true, set to false is necessary
    for all i = 2 to height do  ▷ Loop through codebook (assume first codeword is 0)
        shiftedCodeword = circular shift of the ith row of C
        shiftExists = false
        for all j = 1 to height do ▷ Check shift of codeword i against rest of codebook
            if shiftedCodeword = jth row of C then
                shiftExists = true
                Break the inner loop
            end if
        end for
        if shiftExists = false then      ▷ If no shift of codeword i exists, C isn't cyclic
            isCyclic = false
            Break outer loop
        end if
    end for
    return isCyclic              ▷ Return the true or false, indicating if code is cyclic
end procedure
```

3.6 The relation between codes created by IIR and FIR filters

Creating a FIR filter from an IIR filter is a common operation in digital signal processing. This is done by simply windowing the impulse response of the IIR filter in the time domain. The FIR filter coefficients are the same as the samples of the shortened impulse response (recall Section 2.1.2). This is illustrated in Figure 3.1. If the window is L samples wide, the result is therefore a FIR filter structure that exactly matches the impulse response of the original IIR filter up to the Lth sample.

Recall that digital filtering is characterised in the time domain by discrete convolution. Refer to the appropriate equation (2.2). In this project, the filter input is the message of length k that should be encoded. The filter output is the encoded message, or codeword, of length n. By inspecting the equation, note that, since the output has length n, only the first n samples of the impulse response has an effect on the result. In other words only the first n samples of the impulse response determine how a code of length n is encoded. That is, if the impulse response of an IIR filter is truncated to a length equal to or larger than the code length the resulting FIR filter will still produce the same code of length n.

One advantage of IIR filters for creating error correcting codes is that the same IIR filter can be used to create a code of any length due to their property of infinite impulse response. The code length FIR filters can create is limited by the length of its finite impulse response. This was already briefly mentioned in Section 3.1.2.

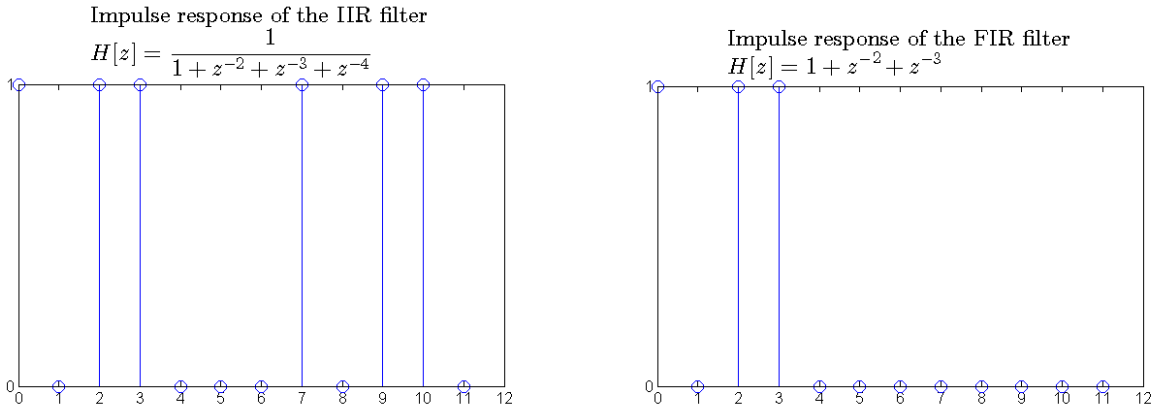

Figure 3.1: The impulse response of the IIR filter on the left is truncated using a window of width $L = 7$ to produce a FIR filter. Since both filters have the same impulse response for the first seven samples, they produce the same codes for code lengths of up to $n = 7$. If $n > 7$ they produce different codes

3.7 Search script

With the necessary functions for code creation developed our attention turns to finding constructions for codes with desirable properties and finding the largest minimum Ham-

ming distance achievable for given values of n and k as stated in the project objective. Intuitively it seems as though there could be a straightforward relation between the filter coefficients or generator matrix and the minimum Hamming distance, which would allow us to predict the minimum distance of a code without searching the codebook, or even tell us how to construct a code with the largest possible minimum distance. This, however, is a deceptively hard problem. With the exception of some particular examples for which shortcuts are known (such as Hamming codes which has minimum distance of 3 or Reed-Solomon codes which has minimum distance $n - k + 1$) no fast way has been found to compute the minimum Hamming distance of a code. A paper in 1978 suggested that no algorithm exists to compute the minimum distance of linear codes in polynomial time [15]. This was proved in 1997 by Vardy, who showed in his paper "The intractability of computing the minimum distance of a code" that calculating the minimum distance of a linear binary code is NP-hard [16]. For this reason most known linear codes which achieve large minimum distances have been found by computer searches running on powerful computers.

In light of that, this project uses known methods to design a computer search algorithm rather than focusing on attempting to find an explicit relation between the coefficients and minimum distance which probably does not exist. This algorithm finds the largest minimum Hamming distance achievable by binary codes created by FIR and IIR filters for given n and k values and finds the filter impulse responses that generated them. After these optimal codes have been found some further investigation is done to see if any patterns emerge.

A computer search considers all possible coefficient combinations and finds the codes created with these filters along with the minimum Hamming distance they achieve. This is a computationally expensive operation. Chandran did a similar search for codes in GF(4) for his Masters dissertation at the University of the Witwatersrand [4]. Chandran approached the problem by generating a message book (as in Section 3.2.1), generating all the possible b and a filter coefficients and then creating a codebook for each filter coefficient combination (as in Section 3.2.2). These codebooks were then searched to find the minimum Hamming distance and generator matrices of the achieved codes. By exploiting some shortcuts from observations and structuring the order of operations better, we can create an improved search algorithm that dramatically improves execution time.

For binary codes all filter coefficients are obviously required to be in GF(2), i.e. they are binary numbers. Recall that, according to the notation used in Equation 2.7, the number of coefficients in the numerator or denominator is one higher than the order. Therefore, for a filter that constructs binary codes with numerator order M, 2^{M+1} possible combinations of b coefficients exist. The denominator of order N can have 2^{N+1} possible a coefficient combinations. This can be reduced slightly by noting that:

- The numerator coefficients shouldn't all be zero. This reduces the number of coefficient combinations in the numerator by one, to $2^{M+1} - 1$
- The first coefficient of the denominator, a_0, should always be 1 (as in Equation 2.7). This reduces the number of coefficient combinations in the numerator by a factor of two to 2^N

Finally we also need to determine what the highest order of M and N are that we need to consider in our search. Refer to Equation 2.6. We are working with a causal filter with zero input, i.e. $y[i] = x[i] = 0$, for $i < 0$. If the output length is L, the index or position of the last output element is obviously $L - 1$. By inspecting (2.6) we can see that if the order of M or N is larger than this index it contributes nothing further to the output. The highest order of either M or N that need to be considered is therefore $L - 1$. For the rest of this project the output length is given by the symbol n instead of L, since the output length is the code length. The highest order M and N that needs to be considered in our search is therefore one less than the code length, i.e. equal to $n-1$.

Putting all this together, for a binary code of length n, the number of transfer function coefficient combinations that need to be considered is given by:

$$\begin{aligned} & 2^N(2^{M+1} - 1) \\ \therefore\ & 2^{n-1}(2^n - 1). \end{aligned} \tag{3.4}$$

This implies that the search algorithm has exponential runtime. This is the big bottleneck restricting the length of codes that can be found by search. Finding good codes quickly becomes intractable for surprisingly short lengths of n, as seen in Figure 3.2.

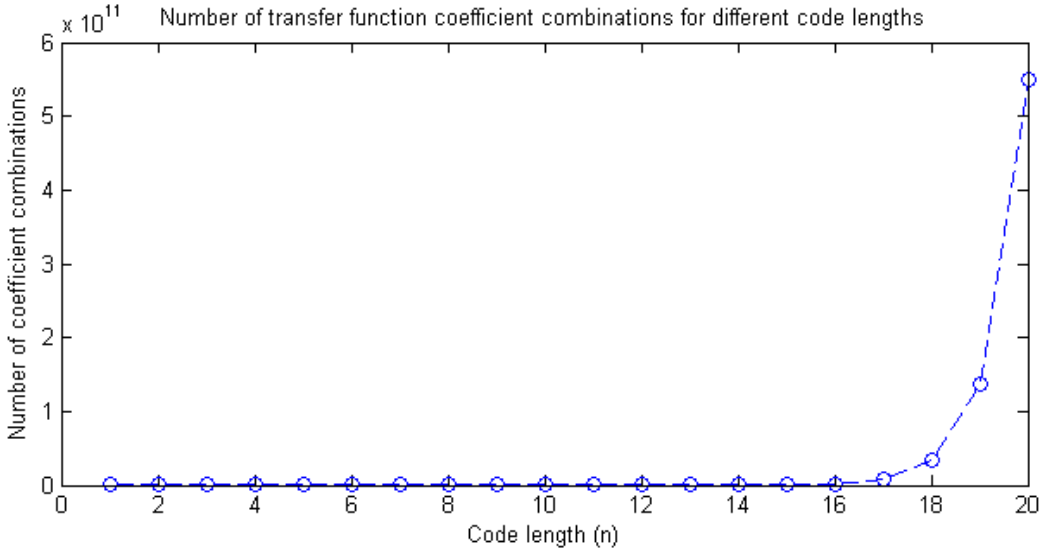

Figure 3.2: The number of transfer functions that need to be searched for different code lengths n if both IIR and FIR filters are considered

3.7.1 Reducing the number of coefficients that has to be searched

The first observation we use is the one made in Section 4.3 and by Chandran in his conclusion: any finite length code that can be constructed by an IIR filter, can also be constructed with a FIR filter. This means that we can find the largest minimum Hamming distance achievable by only searching FIR filters, since all the codes that could be created with IIR filters can also be created by one of the FIR filters that will be searched anyway. This allows us to reduce the number of coefficients that has to be searched to find the largest distance achievable by a factor of 2^{n-1}, from $2^{n-1}(2^n - 1)$ to $2^n - 1$.

3.7.2 Reducing the number of codeword weights that has to be considered

Even though we are only dealing with linear codes, we still need to consider 2^k codewords to find the minimum Hamming distance of a code, which means we are dealing with an exponential time algorithm. However, since the search algorithm is only interested in finding optimal codes for given n and k values it is not necessary to find the minimum distance of all the created codes; if a codeword is found with weight less than the largest minimum distance found up to that point it is already known that the code is not optimal and the codebook does not have to be searched any further. We, therefore, don't need to construct or search the full codebooks of a large portion of the codes.

3.7.3 Program flow

With all that put together the flow of the designed search algorithm can be represented by the flowchart in Figure 3.3. A MATLAB implementation is included in Appendix D as search_script_fir.m. If only the largest minimum Hamming distance is desired and

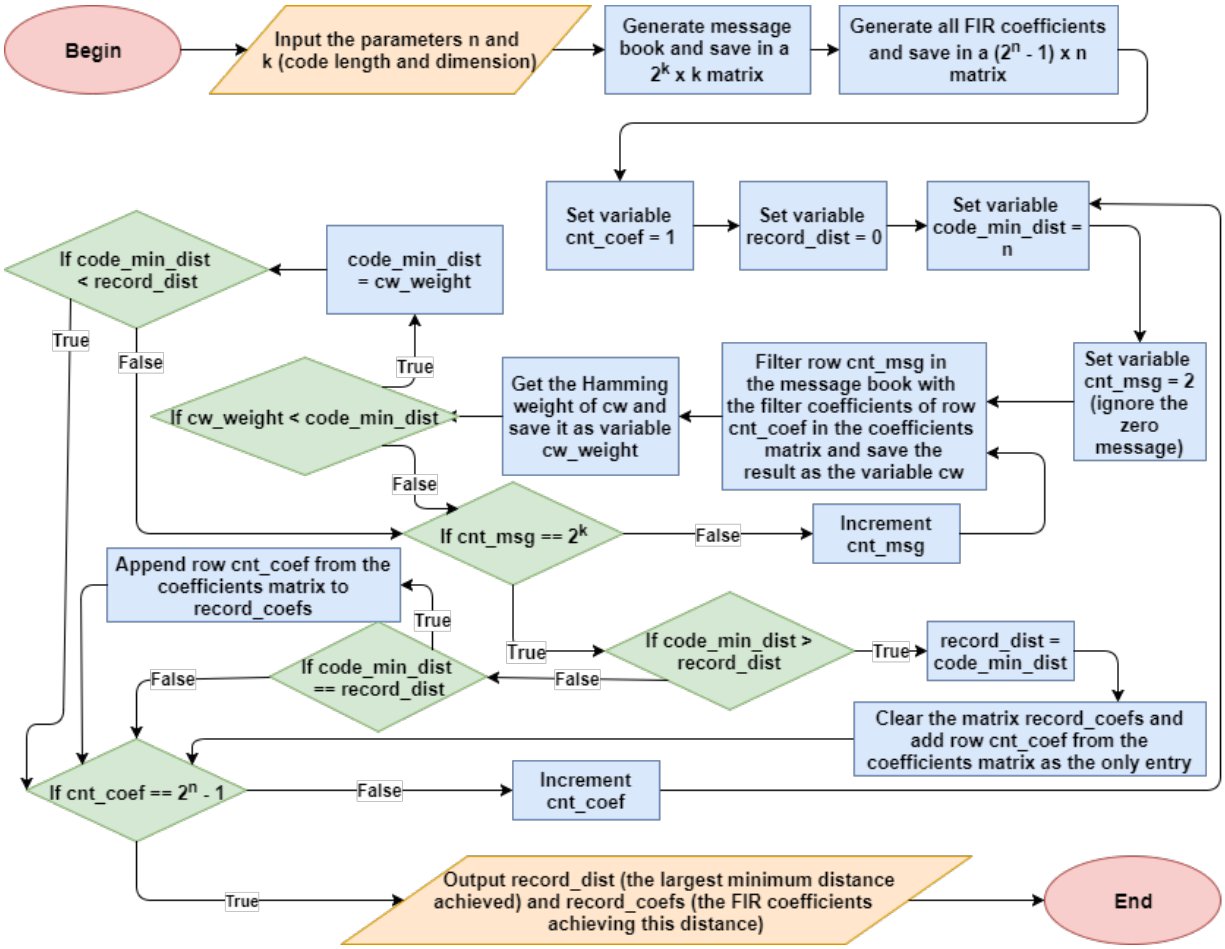

Figure 3.3: Computer search algorithm to find the FIR filter coefficients (i.e. impulse responses) producing the largest minimum Hamming distance for given code length n and dimension k

the programmer has no interest in also finding the FIR filter structures that achieved them, the algorithm can be simplified for more speed. The comparison $code\_min\_dist <$

record_dist can be changed to *code_min_dist* $\leqslant$ *record_dist* (i.e. we can stop searching the codebook already when it is clear that the code won't improve the previous record of largest minimum distance achieved instead of waiting to see if it will match the record), because we don't care what the filter structures are that also achieve the largest minimum distance - we only want to know what the largest minimum distance achieved is.

3.8 A method to find IIR filter structures that create the same binary code as a specified FIR filter

The fact that the described search algorithm only searches FIR filter coefficients significantly improves execution speed and allows us to find codes for larger n and k values than would be possible otherwise on a standard computer, but it comes with one drawback: it doesn't give us any indication of IIR filter coefficients that also produce optimal codes.

Converting an IIR filter to a FIR filter that has the same time-domain impulse response as the original IIR filter up to a certain point, is trivial and a common task in digital signal processing: the impulse response of the IIR filter is simply windowed in the time-domain. We are interested in doing the reverse of that operation. In other words, we want to find all the IIR filters which have impulse responses whose first n samples are the same as the first n samples of the impulse response of a given FIR filter. Those IIR filters will produce the same error correction codes as the original FIR filter for code length n. This, however, is a non-trivial task. In general, there are many solutions to this problem, but since we are working in a finite field and are only looking for filters with a maximum of n coefficients in the numerator or denominator, the number of solutions is still finite.

As far as this author is aware the main method used to achieve this in general signal processing is the Prony least squares method for filter design. This algorithm uses a minimisation technique to find approximate IIR filter coefficients for a specified finite impulse response and returns one solution. This achieves almost the same as what we want to achieve, however we are interested in the exact solutions, and all of them.

To solve this problem this section develops a simple recursive algorithm that can find all the solutions directly. It is also shown that, for binary codes, there are exactly 2^{n-1} filters with numerator and denominator orders of $n-1$ or less that produce the same code as any FIR filter for code length n.

Consider a FIR filter with a known impulse response $h[i] = \{h_0, h_1, \dots, h_{n-1}\}$ that generates a code of length n. We start with the LCCDE description of the filter (2.6). If we set the input equal to an impulse, the output is the known impulse response. The only unknowns remaining in the equation are the filter coefficients a and b. It is clear that there are many solutions. We already know that one solution is setting all the a coefficients to zero and solving for the b coefficients, which yields a FIR filter; but we are interested in finding the IIR filter solutions too. In other words, our task is then to find all the possible a and b coefficient combinations that satisfy this equation. To do this we

write the problem as a linear system in matrix form.

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 & h_0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 & h_1 & h_0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 & h_{n-3} & h_{n-4} & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 & h_{n-2} & h_{n-3} & \cdots & h_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ h_{n-2} \\ h_{n-1} \end{bmatrix}$$

The left matrix has dimensions $n \times (2n-1)$. It consists of an $(n \times n)$ identity matrix and an $(n \times n-1)$ matrix containing impulse response coefficients. Note again, that we only consider IIR filters with numerator and denominator orders $M,\ N \leq n-1$, since any higher order coefficients has no effect on the first n samples of the impulse response and therefore has no impact on the code being produced, as can be seen from the system and as explained in Section 3.4. If the first n calculated coefficients solve the system, any number of higher order coefficients can be added to the numerator or denominator in any combination and we still have a valid solution. In other words if we don't limit the number of coefficients, the number of solutions becomes infinite. Higher order IIR filter coefficients are therefore irrelevant in the calculation and would require unnecessary memory blocks to implement anyway.

To solve the system we work from the top, solving each row after the row before it. It is apparent that

$$b_0 = h_0. \tag{3.5}$$

Thereafter we have,

$$\begin{aligned} b_1 + h_0 a_1 &= h_1, \\ b_2 + h_1 a_1 + h_0 a_2 &= h_2, \\ b_3 + h_2 a_1 + h_1 a_2 + h_0 a_3 &= h_3, \\ b_4 + h_3 a_1 + h_2 a_2 + h_1 a_3 + h_0 a_4 &= h_4, \end{aligned} \tag{3.6}$$

and so forth until we have n equations in total.

For the first equation only one solution exists. If we solve from the top down there are two unknowns in each of the rest of the equations when we arrive at it: one b coefficient and one a coefficient. Only four possible scenarios exist for each equation:

- $h_0 = 0$ and the equation is satisfied if the unknowns are both set to 0.
  Solutions: $b_i = 0$, $a_i = 0$ or $b_i = 0$, $a_i = 1$ (a_i has no effect)
- $h_0 = 1$ and the equation is satisfied if the unknowns are both set to 0.
  Solutions: $b_i = a_i = 0$ or $b_i = a_i = 1$
- $h_0 = 0$ and the equation is not satisfied if the unknowns are both set to 0.
  Solutions: $b_i = 1$, $a_i = 0$ or $b_i = 1$, $a_i = 1$ (a_i has no effect)
- $h_0 = 1$ and the equation is not satisfied if the unknowns are both set to 0.
  Solutions: $b_i = 0, a_i = 1$ or $b_i = 1, a_i = 0$

Therefore there is one solution for the first row (3.5) and two solutions for rest of the rows (3.6), i.e. there are $n-1$ rows with two solutions. Since the solution to each row is obviously dependent on all the rows before it, the implementation of this algorithm is recursive. That is, each of the two solutions in a certain row is continued by two more solutions in the next row, which is then each continued by two more solutions in the next row and so forth. This means that there are $1 \times 2^{n-1} = 2^{n-1}$ solutions to the system in total. In other words, for any FIR filter, there exists 2^{n-1} filters with numerator and denominator order of $n-1$ or less that produce the same binary impulse response up to sample n. One of these solutions is a FIR filter (the original FIR filter) and all the other solutions are IIR.

We can confirm this result by recalling that there are $2^n - 1$ FIR filter structures that produce unique impulse responses up to sample n, if we ignore the filter where all b coefficients are zero as we have done before. If each of these FIR filters has 2^{n-1} corresponding filters producing the same code of length n, we have $2^{n-1}(2^n - 1)$ filter structures in total. This is the same result we arrived at in Equation 3.4. In conclusion: by searching for optimal FIR filters first and then using this method to find all their corresponding IIR filters, we can find all of the $2^{n-1}(2^n - 1)$ transfer function coefficient combinations that produce optimal codes while only having to search a fraction of them.

A pseudocode description of the algorithm implementation is provided as Algorithm 4. A MATLAB implementation of the algorithm is included in Appendix D as the function find_all_equivalent_iirs.m.

If only one IIR filter is desired instead of all the possible solutions, only one of the two solutions for each row can be chosen every time according to a certain strategy. This allows a solution to be found in polynomial time, which greatly improves execution speed for large values of n. The solution one might be most interested in is the IIR filter structure with the lowest order denominator and numerator, since it requires the least memory blocks to implement. A heuristic method that often, but not always, works well to find this lowest order solution is to always choose $b_i = a_i = 0$ for the first two of the four scenarios, $b_i = 1, a_i = 0$ for the third and $b_i = 0, a_i = 1$ for the fourth scenario. If $h_0 = 1$ this method delivers an all-pole filter. In the experience of the author the filters produced by this strategy has periodic impulse responses, so often deliver the best answer when the impulse response to be matched has a periodic pattern with a short period.

Algorithm 4 Find IIR filter structures that create the same binary code as a specified FIR filter

```
procedure FindEquivalentIIRs(bFIR, n)        ▷ bFIR is an array FIR filter coefficients, and n the number of samples in the impulse response to be matched
    iirFilters = an empty (1 × 2n) matrix ▷ each row in this matrix will hold an a and b coefficients pair, the first n columns holding b coefficients and the last n columns a
    if n > the length of bFIR then
        impulseResponse = zero padded bFIR to the length of n
    else
        impulseResponse = the first n samples of bFir
    end if
    first entry of bIIR = first sample of impulseResponse
    first entry of aIIR = 1                                   ▷ a0 is always = 1
    SolveNewRow(bIIR, aIIR, 2)
    procedure SolveNewRow(bIIR, aIIR, i) ▷ solve for the ith b and a coefficients
        if i = n + 1 then                          ▷ Recursion must end and result saved
            append bIIR and aIIR horizontally to form a row vector
            append that row vector vertically to iirFilters
        else
            if the current row of the system is satisfied with a = b = 0 then
                if the first sample of impulseResponse = 0 then          ▷ if h0 = 0
                    SolveNewRow(bIIR, aIIR, i+1)                         ▷ b = 0, a = 0
                    ith element of aIIR = 1
                    SolveNewRow(bIIR, aIIR, i+1)                         ▷ b = 0, a = 1
                else
                    SolveNewRow(bIIR, aIIR, i+1)                         ▷ b = 0, a = 0
                    ith element of bIIR = 1
                    ith element of aIIR = 1
                    SolveNewRow(bIIR, aIIR, i+1)                         ▷ b = 1, a = 1
                end if
            else
                if the first sample of impulseResponse = 0 then          ▷ if h0 = 0
                    ith element of bIIR = 1
                    SolveNewRow(bIIR, aIIR, i+1)                         ▷ b = 1, a = 0
                    ith element of aIIR = 1
                    SolveNewRow(bIIR, aIIR, i+1)                         ▷ b = 1, a = 1
                else
                    ith element of bIIR = 1
                    SolveNewRow(bIIR, aIIR, i+1)                         ▷ b = 1, a = 0
                    ith element of bIIR = 0
                    ith element of aIIR = 1
                    SolveNewRow(bIIR, aIIR, i+1)                         ▷ b = 0, a = 1
                end if
            end if
        end if
    end procedure
    return irrFilters
end procedure
```

3.9 Attempting to find IIR filters that produce multiple good codes

As already mentioned before, IIR filters can encode a message into a codeword of any length, due to the fact that their impulse responses are infinite. Therefore if a group of impulse responses are found which extend each other and all produce optimal codes, they may be represented with a single IIR filter instead of a collection of FIR filters. For example, assume it is known that the impulse responses

$$h_1[i] = \{1, 1, 0\},$$
$$h_2[i] = \{1, 1, 0, 1\},$$
$$h_3[i] = \{1, 1, 0, 1, 1\},$$
$$h_4[i] = \{1, 1, 0, 1, 1, 0\},$$

all produce optimal codes for a certain fixed k and $n = 3, 4, 5$ and 6, respectively. Instead of having to specify four different FIR filter structures, we would be able to use the method developed in Section 3.8 to find a general IIR filter structure able to produce all those codes as a general solution. In this case one such IIR filter would be $H(z) = \frac{1}{1+z^{-1}+z^{-2}}$.

This possibility is investigated in this project, with a MATLAB script that scans through the FIR filter structures, i.e. impulse responses, which achieved the largest minimum Hamming distance and checking to see if any repetition emerges which we can then generalise into an IIR filter using the method developed in Section 3.8.

3.9.1 Program flow

The flow of the script can be represented by the flowchart in Figure 3.4. A MATLAB implementation is included in Appendix D as search_for_reusable_iir_filters_script.m.

3.10 Finding cyclic codes

Finding filters that generate cyclic codes for a certain n and k may be of interest. A brute force approach to do this would simply be trying all the different coefficient combinations, constructing their codebooks, and then testing it for cyclicity using Algorithm 3. This operation, however, is very computationally intensive, because it would require the full codebook to be constructed and examined for every coefficient combination. In addition Algorithm 3 is an exponential time algorithm. We therefore desire a more direct method which allows us to find (n, k) cyclic codes and the transfer functions that generate them. This section develops such an algorithm.

Recall from the Section 2.3.3 that any polynomial $g(x)$ of order $n - k$ which divides $x^n + 1$ is a generator polynomial of a (n, k) cyclic code. The section also mentions that multiplication with a generator polynomial is the same as discrete convolution with its coefficients, and that we can therefore construct the cyclic code with a filter that has an impulse response equivalent to the coefficients of $g(x)$. Similarly, polynomial division is the same as discrete deconvolution.

This information is used to find cyclic codes. We test every polynomial of order $n - k$ to

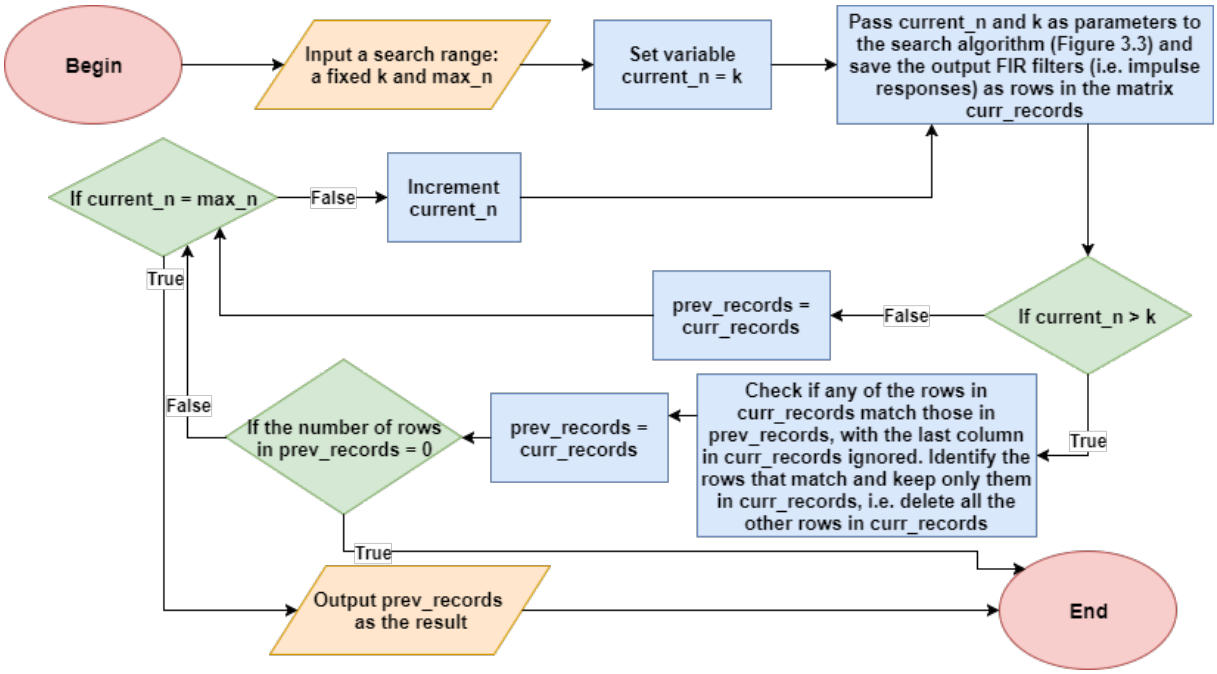

Figure 3.4: Computer search algorithm to find impulse responses that produce optimal binary codes for a certain k and range of n

see if it divides $x^n + 1$. This is done with discrete deconvolution. If such a polynomial does indeed divide $x^n + 1$, it is known that it is a generator polynomial for a cyclic code and the coefficients of the polynomial therefore gives the impulse response of the filter that constructs the code. This can be directly implemented with a FIR filter that has coefficients equal to the impulse response. For example, suppose it is found that the following polynomial of order $n - k$ is the generator polynomial of a $(7, 4)$ cyclic code:

$$g(x) = 1 + x + x^3. \tag{3.7}$$

This can then be represented by the following impulse response of length n,

$$h[i] = \{1, 1, 0, 1, 0, 0, 0\}, \tag{3.8}$$

and therefore be directly implemented with the following FIR filter:

$$H(z) = 1 + z^{-1} + z^{-3}. \tag{3.9}$$

If desired we can then use the method developed in the previous section to find IIR filters that match the specified impulse response up to sample n and therefore produce the same cyclic code. For the given example, one such IIR filter would be:

$$H(z) = \frac{1}{1 + z^{-1} + z^{-2} + z^{-4}}. \tag{3.10}$$

The procedure can easily be extended to find the best codes among the cyclic codes found by constructing their codebooks and finding the largest minimum Hamming distance achieved. A MATLAB implementation is included in Appendix D as find_cyclic_codes_script.m

3.11 Summary of chapter

This chapter discussed the methodology followed for the construction and investigation of linear block codes in this project. This includes, but is not limited to, the following aspects: the construction of the message and codebook, generator matrix construction, minimum Hamming distance calculation, checking for cyclicity in codes, finding a relation between codes constructed by FIR and IIR filters, a computer search script for optimal codes, finding IIR filters that construct the same code as a given FIR filter, attempting to find IIR filters that construct multiple good codes, and, finally, specifically searching for good cyclic codes. Design choices, motivation, and an overview of the implementation using a flowchart or pseudo-code is given for each of these aspects.

The next chapter discusses the results achieved.

Chapter 4

Results

4.1 Largest minimum distances of constructed codes

The search program described in Section 3.7 was used to find the largest minimum Hamming distance achievable by binary codes constructed with filters for all $n, k \leq 20$, as required by the project objectives. The complete documentation of these results is shown in Table 4.1. Codes were constructed that reached both the Griesmer and Hamming bounds. The largest minimum distances achieved are equal to the largest known distances achieved by binary linear block codes in all but two cases (this is determined by comparison with the tables found at [17]). These two exceptions are the $(8, 4)$ and $(18, 9)$ codes. In both cases the largest minimum distance we achieved was one less than the largest known distance.

n \ k	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
1	1																			
2	2	1																		
3	3	2	1																	
4	4	2	2	1																
5	5	3	2	2	1															
6	6	4	3	2	2	1														
7	7	4	4	3	2	2	1													
8	8	5	4	3	2	2	2	1												
9	9	6	4	4	3	2	2	2	1											
10	10	6	5	4	4	3	2	2	2	1										
11	11	7	6	5	4	4	3	2	2	2	1									
12	12	8	6	6	4	4	4	3	2	2	2	1								
13	13	8	7	6	5	4	4	4	3	2	2	2	1							
14	14	9	8	7	6	5	4	4	4	3	2	2	2	1						
15	15	10	8	8	7	6	5	4	4	4	3	2	2	2	1					
16	16	10	8	8	7	6	6	5	4	4	3	2	2	2	2	1				
17	17	11	9	8	8	7	6	6	5	4	4	3	2	2	2	2	1			
18	18	12	10	8	8	8	7	6	5	4	4	4	3	2	2	2	2	1		
19	19	12	10	9	8	8	8	7	6	5	4	4	4	3	2	2	2	2	1	
20	20	13	11	10	9	8	8	8	7	6	5	4	4	4	3	2	2	2	2	1

Table 4.1: The largest minimum Hamming Distance achieved in this project for a given code length n and dimension k

In addition, FIR and IIR filters constructing cyclic codes were succesfully found using the method of Section 3.10. Since so many different codes were constructed during the course of this project - for each combination of n and k values there are often thousands of transfer functions that construct codes reaching the optimal distance listed in Table 4.1 - it is impossible to list them all here. Section 4.2 contains the construction of a few notable codes to serve as examples of the process followed. Section 4.3 also gives a thorough example of the complete process of code construction followed for this project.

4.1.1 The $(8, 4)$ and $(18, 9)$ codes

The codebook of the $(8, 4)$ code that we could not achieve was constructed by using the steps provided in [17]. This codebook is non-systematic and does not contain k shifted versions of the impulse response (which is required for codes constructed with causal filters as previously determined in Section 3.4). An experiment was done to see if this code could be constructed by a non-causal filter: and it could not. Details of this further investigation is omitted here due to space constraints, but included in Appendix C for the interested reader.

4.2 Notable examples of codes constructed

4.2.1 (7, 4) Hamming code

There are two unique FIR filters (i.e. two unique impulse responses) that construct optimal $(7, 4)$ codes with minimum distance 3, and therefore $2(2^6)$ filters constructing optimal $(7, 4)$ codes in total. The two FIR filters are each given below, along with one of the example IIR filters that construct the same code. Both these codes are cyclic. A generator matrix is also given, in both the shifted impulse response form as it was calculated, and the transformation to the standard systematic form.

The first of the two:

- FIR filter transfer function

$$H(z) = 1 + z^{-2} + z^{-3}$$

- IIR filter transfer function

$$H(z) = \frac{1}{1 + z^{-2} + z^{-3} + z^{-4}}$$

- Generator matrix

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

- Additional code properties: cyclic, achieves Griesmer bound, achieves Hamming bound

The second code:

- FIR filter transfer function

$$H(z) = 1 + z^{-1} + z^{-3}$$

- IIR filter transfer function

$$H(z) = \frac{1}{1 + z^{-1} + z^{-2} + z^{-4}}$$

- Generator matrix

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

- Additional code properties: cyclic, achieves Griesmer bound, achieves Hamming bound

For the cases considered in the rest of the chapter, there are generally much more than two unique impulse responses that construct optimal codes for the given values of n and k. Therefore only one example will be given in each case.

4.2.2 (15, 11) Hamming code

The largest minimum Hamming distance found for $n = 15$ and $k = 11$ is 3. The search script returned 15 FIR filter structures which create $(15, 11)$ codes that achieve this distance. Only two of them are cyclic: the codes created by the FIR filters with coefficients $b = [1\ 1\ 0\ 0\ 1]$ and $b = [1\ 0\ 0\ 1\ 1]$. If these coefficients are written in polynomial form, it can be observed that they divide $x^n + 1 = x^{15} + 1$, which confirms that they should be cyclic. An example of one of these codes is given below.

- Example FIR filter transfer function

$$H(z) = 1 + z^{-1} + z^{-4}$$

- Example IIR filter transfer function

$$H(z) = \frac{1 + z^{-1} + z^{-2} + z^{-7}}{1 + z^{-1} + z^{-2} + z^{-3}}$$

- Generator matrix

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

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

- Additional code properties: cyclic, does not achieve Griesmer bound, achieves Hamming bound

4.2.3 Golay code

The $(23, 12)$ perfect binary Golay code is a perfect code (achieves the Hamming bound), with a minimum Hamming distance of 7. Several constructions were found for $(23, 12)$ codes with minimum distance 7. One of these constructions that also produces a cyclic code is shown.

- Example FIR filter transfer function

$$H(z) = 1 + z^{-2} + z^{-4} + z^{-5} + z^{-6} + z^{-10} + z^{-11}$$

- Example IIR filter transfer function

$$H(z) = \frac{1}{1 + z^{-2} + z^{-5} + z^{-8} + z^{-9} + z^{-10} + z^{-11} + z^{-12}}$$

- Generator matrix

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

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

- Additional code properties: cyclic, does not achieve Griesmer bound, achieves Hamming bound

4.3 The relation between codes created by IIR and FIR filters

Reducing the coefficients to be searched to only FIR coefficients greatly improves the execution time of the search script. Where it was previously not possible to search for much larger values than n or k equal to 10, the improved search algorithm only started to become slow for code lengths near 20. The method of finding IIR filters from a given FIR filter works well. Since the algorithm was used many times and in many different ways, the results are best illustrated with a particular example. Consider the familiar case, already mentioned earlier in Section 4.2.1, where the optimal codes and largest minimum Hamming distance need to be found for $n = 7$, $k = 4$.

The search script (Section 3.7) finds that the largest minimum Hamming distance achieved is 3 and returns the following FIR filters which achieved it:

$$H(z) = 1 + z^{-2} + z^{-3}$$
$$H(z) = 1 + z^{-1} + z^{-3}$$

Let us continue the example with the FIR filter $H(z) = 1 + z^{-2} + z^{-3}$. We can extract the generator matrix of the code it generates using the method of Section 3.4, and then put that in systematic form:

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

For interest's sake, note that this is the (7,4) Hamming code of minimum distance 3. Using the method of Section 3.8 on this FIR filter, we then successfully find $2^{n-1} = 64$ filter structures that produce the same code. For brevity they won't all be listed here, but some of the IIR filters found are tabulated in Table 4.2.

b coefficients	a coefficients
[1 0 1 1 0 0 1]	[1 0 0 0 0 0 1]
[1 0 1 1 0 1 0]	[1 0 0 0 0 1 0]
[1 0 1 1 1 0 0]	[1 0 0 0 1 0 1]
[1 0 1 1 1 1 1]	[1 0 0 0 1 1 0]
[1 0 1 0 0 1 0]	[1 0 0 1 0 0 1]
[1 0 1 0 0 0 0]	[1 0 0 1 0 1 1]
[1 0 0 1 0 0 0]	[1 0 1 0 1 1 1]

b coefficients	a coefficients
[1 1 1 0 1 0 0]	[1 1 0 0 0 0 0]
[1 1 1 0 1 1 0]	[1 1 0 0 0 1 0]
[1 1 1 0 0 0 1]	[1 1 0 0 1 0 0]
[1 1 1 1 1 1 1]	[1 1 0 1 0 0 0]
[1 1 1 1 0 0 1]	[1 1 0 1 1 1 1]
[1 1 0 0 1 0 1]	[1 1 1 0 1 1 0]
[1 1 0 1 1 1 0]	[1 1 1 1 1 1 0]

Table 4.2: Fourteen of the sixty-four filter transfer function coefficient pairs that produce the same code of length 7 as the FIR filter $H(z) = 1 + z^{-2} + z^{-3}$

Each of these filter structures has the same impulse response as $H(z) = 1 + z^{-2} + z^{-3}$ for the first seven samples:

$$h[i] = \{1, 0, 1, 1, 0, 0, 0\}$$

They therefore all produce the same code for $n = 7$. To illustrate we can choose any of the 64 filters we calculated. Consider the second example in Table 4.2: $b = [1\ 0\ 1\ 1\ 0\ 1\ 0], a = [1\ 0\ 0\ 0\ 0\ 1\ 0]$, with the transfer function:

$$G(z) = \frac{1 + z^{-2} + z^{-3} + z^{-5}}{1 + z^{-5}}.$$

The impulse response of this transfer function is:

$$g[i] = \{1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, ...\}. \tag{4.1}$$

It is clear that it matches the FIR filter impulse response for the first seven samples. Calculating the systematic generator matrix of $G(z)$ yields:

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

Which is, as expected, exactly the same as the generator matrix for the code constructed by $H(z)$.

To summarise: using this process we have found all 128 transfer functions with numerators and denominators of order $n - 1$ or less that that produce binary $(7, 4)$ codes of minimum distance 3. There are $2^{n-1}(2^n - 1) = 8192$ possible transfer functions which were all candidates to achieve this, but we only had to search the FIR filters among them, which reduced the scope of the search to only $(2^n - 1) = 127$ transfer functions. The search yielded two FIR filters. By using our FIR to IIR conversion method we found all the IIR filters that match the impulse response of these two FIR filters for the first n (seven) samples. This yielded 128 solutions. We could therefore find all 128 solutions for our original problem while only having to search and calculate the minimum distance of a fraction of them. This difference becomes even more pronounced for larger code lengths and it makes a major difference in the execution speed of the search script for large values of n.

4.4 IIR filters that achieve optimal minimum Hamming distance for multiple code lengths

The investigation into finding IIR filters that produce optimal codes for a fixed k and any n (as explained in Section 3.9) yielded only two results. The first does not really count as it is trivial. The second is more interesting.

For $k = 1$ and any n, the largest minimum Hamming distance is equal to n. The codebook contains only two entries and there is always just one impulse response of length n that achieves this optimal code (which is simply the impulse response that constructs a repetition code) and this result is therefore trivial. This single optimal code is produced by the impulse response where the first n samples are equal to 1. We can generalise this into an IIR filter. The IIR filter,

$$H(z) = \frac{1}{1 + z^{-1}},$$

which has an infinite impulse response that is always equal to 1 therefore produces the optimal code for any code length, n, if $k = 1$.

The second result found IIR filters that seem to be able to construct optimal codes for all code lengths if $k = 2$. This result is not trivial since there are in general, unlike the previous case, many impulse responses that construct optimal codes for $k = 2$. To illustrate this, consider Table 4.3 containing the FIR filter coefficients / impulse responses that the search script returned for $k = 2$ and different code lengths.

$n = 4, k = 2$
[0 1 1 0]
[1 0 1 0]
[1 0 1 1]
[1 1 0 0]
[1 1 0 1]
[1 1 1 0]

$n = 5, k = 2$
[1 0 1 1 0]
[1 0 1 1 1]
[1 1 0 1 0]
[1 1 0 1 1]
[1 1 1 0 1]

$n = 6, k = 2$
[1 0 1 1 1 0]
[1 1 0 1 1 0]
[1 1 1 0 1 0]

$n = 7, k = 2$
[0 1 0 1 1 1 0]
[0 1 1 0 1 1 0]
[0 1 1 1 0 1 0]
[1 0 0 1 1 1 0]
[1 0 1 0 1 1 0]
[1 0 1 0 1 1 1]
[1 0 1 1 0 1 0]
[1 0 1 1 0 1 1]
[1 0 1 1 1 0 0]
[1 0 1 1 1 0 1]
[1 0 1 1 1 1 0]

$n = 7, k = 2$ (continued)
[1 1 0 0 1 1 0]
[1 1 0 1 0 1 0]
[1 1 0 1 0 1 1]
[1 1 0 1 1 0 0]
[1 1 0 1 1 0 1]
[1 1 0 1 1 1 0]
[1 1 1 0 0 1 0]
[1 1 1 0 1 0 0]
[1 1 1 0 1 0 1]
[1 1 1 0 1 1 0]
[1 1 1 1 0 1 0]

Table 4.3: FIR filter b coefficients / impulse responses achieving optimal minimum Hamming distance for the given values of n and k, as returned by the search script

The number of solutions keeps on growing. For example, if $n = 20$ and $k = 2$, the search script returns 2640 unique solutions. From investigation of these results, however, a pattern emerges: a periodic repetition of the bits $(1, 1, 0)$ is always one of the results. The cells of these results are shown in bold in Table 4.3. This trend was identified by the search script developed for Section 3.9 and held for all $n \leq 20$. Using the method developed in this project IIR filters was found that produce this $(1, 1, 0)$ repetition impulse response

up to atleast $n = 20$. One of the results was the IIR filter with the following transfer function:

$$H(z) = \frac{1}{1 + z^{-1} + z^{-2}}, \tag{4.2}$$

or alternatively,

$$H(z) = \frac{1 + z^{-1}}{1 + z^{-3}}. \tag{4.3}$$

These transfer functions have periodic impulse responses, periodically repeating the $(1, 1, 0)$ pattern. What makes this result very interesting is that these transfer functions seem to construct optimal codes not only for $n \leq 20$, but indeed for all values of n. The transfer function was tested for all code lengths of $n \leq 256$ and always produced the largest known minimum distance for binary codes as tabulated in [17]. Unfortunately [17] does not provide the largest known minimum distance for $n > 256$, but it seems like this trend might very well continue as $n \to \infty$. In addition, the codes produced by (4.3) are cyclic if the code length is a factor of three,

$$\frac{n}{3} \in \mathbb{Z},\ n > k.$$

This was first observed empirically, but can also be proven. The full proof derived by the author follows.

Proof. We want to prove that $G(z) = \frac{1+z^{-1}}{1+z^{-3}}$ generates a cyclic code if $\frac{n}{3} \in \mathbb{Z},\ n > k$. First we show that this is true for the base case $n = 3$. When $n = 3$, $G(z)$ can be truncated to a FIR filter $G_1(z) = 1 + z^{-1}$ that has the impulse response $\{1, 1, 0\}$. Representing that in polynomial form:

$$g_1(x) = 1 + x \tag{4.4}$$

Using long division:

$$\frac{x^n + 1}{g_1(x)} = \frac{x^3 + 1}{1 + x} = x^2 + x + 1. \tag{4.5}$$

Which shows that the polynomial $g_1(x)$ of order $n-k$ divides x^n+1 and therefore generates a $(3, 2)$ cyclic code. We can therefore write:

$$g_1(x)q_1(x) = x^3 + 1 \tag{4.6}$$

We will use this information and generalise it to obtain a generator polynomial $g_2(x)$ of order $n - 2$ for a cyclic code of a longer unknown code length n, which we can write as:

$$g_2(x)q_2(x) = x^n + 1. \tag{4.7}$$

Define the polynomials $a(x)$ and $b(x)$ as:

$$\begin{aligned} g_2(x) &= g_1(x)a(x) \\ q_2(x) &= q_1(x)b(x) \end{aligned} \tag{4.8}$$

Combining (4.6) with (4.7) and using polynomial long division yields:

$$\begin{aligned} a(x)b(x) &= \frac{x^n + 1}{x^3 + 1} \\ &= x^{n-3} + x^{n-6} + x^{n-9} + \ ... \end{aligned} \tag{4.9}$$

This clearly only terminates without a remainder if n is a factor of 3. We introduce the restriction $\frac{n}{3} \in \mathbb{Z}$ and write:

$$a(x)b(x) = x^{n-3} + x^{n-6} + x^{n-9} + \ \ldots \ + 1, \ \frac{n}{3} \in \mathbb{Z}. \tag{4.10}$$

If the order of $a(x)$ is i, and the order of $b(x)$ is j, then:

$$i + j = n - 3. \tag{4.11}$$

Since the order of $g_1(x)$ is 1, we know from (4.8) we that the order of $g_2(x)$ is $i+1$. Then,

$$\begin{aligned} i + 1 &= n - 2 \\ \therefore i &= n - 3 \end{aligned} \tag{4.12}$$

From (4.11) this requires that $j = 0$ and therefore $b(x) = 1$. Therefore,

$$a(x) = 1 + \ \ldots \ + x^{n-9} + x^{n-6} + x^{n-3}, \ \frac{n}{3} \in \mathbb{Z}. \tag{4.13}$$

Therefore,

$$g_2(x) = a(x)g_1(x) = (1 + \ \ldots \ + x^{n-9} + x^{n-6} + x^{n-3})(1 + x), \tag{4.14}$$

generates a cyclic code if $\frac{n}{3} \in \mathbb{Z}$. Rewrite $g_1(x)$ as a FIR filter,

$$G_1(z) = 1 + z^{-1}. \tag{4.15}$$

The impulse response of length n represented by $a(x)$ is $\{1, 0, 0, 1, 0, 0, \ \ldots \ , 1, 0, 0\}$, which is simply a periodic repetition of a one every three terms. This can be written as an IIR filter which matches that impulse response up to length n,

$$A(z) = \frac{1}{1 + z^{-3}}. \tag{4.16}$$

Finally we write,

$$G_2(z) = A(z)G_1(z) = \frac{1 + z^{-1}}{1 + z^{-3}}, \tag{4.17}$$

where $G_2(z)$ then generates a cyclic code if $\frac{n}{3} \in \mathbb{Z}, \ n > k$. This is what we set out to prove. □

Chapter 5

Conclusion

This chapter draws together the calculations, observations, and results made or seen during the previous three chapters, to make a few general conclusions on the construction of FEC codes using FIR and IIR filters. Finally, based on the conclusions, a few recommendations are given for future research on the topic.

5.1 Conclusions

The conclusions arrived at in this work are shown below, grouped under a relevant subsection and supplemented with additional information where necessary.

5.1.1 On the performance of binary codes created by FIR and IIR filters

- Such codes are capable of reaching both the Hamming and Griesmer bounds
- The largest minimum Hamming distance achievable by such codes are, in the vast majority of cases, equivalent to the largest known minimum Hamming distance achievable by binary codes for the given code length n and dimension k

5.1.2 On the nature of binary codes created by FIR and IIR filters

- Block codes created by FIR and IIR filters are linear

  This was empirically observed and also proven in Section 3.3.

- The $(k \times n)$ Toeplitz matrix containing k non-circularly shifted versions of the filter impulse response as rows is a valid G matrix for such a (n, k) code

  Discussed in Section 3.4.

- If the codebook of a linear block code does not contain k non-circularly shifted versions of some vector, the code is not constructible with a causal FIR or IIR filter

  This follows from the previous conclusion and the fact that the rows of G are also codewords in the codebook of a linear block code. Another way to arrive at this same conclusion is by noting that the message book of a code with dimension k, contains k shifted impulses. E.g. for $k = 3$, the message book contains $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$. The encoded form of the first vector is then the impulse response and, from the time invariance property of an LTI system, we know that the other two are encoded as shifted versions of the same impulse response.

5.1.3 On the construction of binary codes with FIR and IIR filters

- Any finite length code constructible by an IIR filter is also constructible by some FIR filter.

  That FIR filter is obtained by windowing the impulse response of the IIR filter in the time domain to some length equal to or greater than the code length n.

- For any FIR filter that can construct a linear block code of length n, there exists exactly 2^{n-1} filters with numerator and denominator orders of $n-1$ or less, that construct the same code. The original FIR filter is also one of the 2^{n-1} solutions, but all the other solutions are IIR filters.

  These filters can be found with the algorithm developed in Section 3.8.

- IIR transfer functions exist that can construct optimal binary codes for $k = 2$ and any given n, but no analogous filters exist for any other binary codes with $k \leq 20$

  Found with the script developed in Section 3.9. An example is the transfer function $H(z) = \frac{1+z^{-1}}{1+z^{-3}}$. The codes constructed by this filter is cyclic if n is a factor of 3.

5.2 Future recommendations

This project concentrated only on binary codes. A future recommendation would be to generalise the search algorithm and FIR to IIR filter conversion algorithm to work in higher order Galois fields than just GF(2). The improved efficiency of the methods developed in this project would allow transfer functions for the construction of optimal non-binary codes to be found for larger values of n than was possible for similar attempts, such as [4].

A second possibility is further investigation of the FIR to IIR filter algorithm developed, to see if a method can be found to directly solve only for the IIR filter coefficients with the lowest order, i.e. the IIR filters that would require the least memory blocks to implement.

Another possible interesting research topic could be a deeper investigation of the very last conclusion made in this report: IIR filters that produce optimal codes for a wide range of parameters. The research could investigate if such IIR filters exist for larger values of k. Or perhaps the focus could shift to finding a more intricate pattern that this project did not investigate, for example looking for IIR filters that produce optimal codes for odd values of n and a given k, or only a certain range of n for a given k.

Finally, further investigation may be done to see if better codes can be constructed using puncturing of the codebooks constructed by the filters.

Bibliography

[1] W.A. Hamming. "Error Detecting and Error Correcting Codes". *The Bell System Technical Journal*, vol. 29(2), pp. 147-169, April 1950.

[2] J.L. Massey. "Deep-Space Communications and Coding: A Marriage Made in Heaven". *Proceedings of an International Seminar Organized by Deutsche Forschungsanstalt für Luft- und Raumfahrt (DLR) Bonn, Germany*, 1992, pp. 1-17.

[3] C.E. Shannon. "A Mathematical Theory of Communication". *The Bell System Technical Journal*, vol. 27, pp. 379-423, 623–656, July, October, 1948.

[4] A. Chandran. "Investigation of the use of Infinite impulse response filters to construct linear block codes". Masters Dissertation, University of The Witwatersrand, South Africa, 2016.

[5] J.P. Hespanha. *Linear Systems Theory.* Princeton, New Jersey: Princeton University Press, 2009, pp. 78.

[6] J.G. Proakis, D.K. Manolakis. *Digital Signal Processing.* Edinburgh Gate, Harlow, Essex CM20 2JE: Pearson Education Limited, 2014, pp. 151-228.

[7] D. Schlichthärle. *Digital Filters: Basics and Design.* Springer, 2000, pp. 83-98.

[8] "scipy.signal.filter". Internet: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.lfilter.html, 21 Jun, 2017 [2 Aug, 2017]

[9] "filter". Internet: https://www.mathworks.com/help/matlab/ref/filter.html, 2017 [2 Aug, 2017]

[10] A.V. Oppenheim. Class Lecture, Topic: "IIR, FIR Filter Structures". Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 2005.

[11] S. Lin, D.J. Costello. *Error Control Coding, 2nd Edition* Upper Saddle River, NJ, USA: Prentice Hall, 2004, pp. 15 - 121

[12] W.C. Huffman, V. Pless. *Fundamentals of Error-Correcting Codes* Cambridge, UK: Cambridge University Press, 2003, pp. 2-3.

[13] P. Sweeney. *Error Control Coding* West Sussex, England: John Wiley & Sons Ltd, 2002, pp. 69-71.

[14] B.P. Lathi, Z. Ding. *Modern Digital and Analog Communication Systems* 198 Madison Avenue, New York, New York 10016: Oxford University Press, 2010, pp. 802-846

[15] E. Berlekamp, R. McEliece, H. van Tilborg. "On the inherent intractability of certain coding problems". *IEEE Transactions on Information Theory*, Volume 24, Issue 3, pp. 384 - 386, May 1978

[16] A. Vardy. "The intractability of computing the minimum distance of a code". *IEEE Transactions on Information Theory*, Volume 43, Issue 6, pp. 1757 - 1766, Nov 1997

[17] M. Grassl. "Bounds on the minimum distance of linear codes over GF(2)." Internet: codetables.de/BKLC/Tables.php?q=2& n0=1& n1=20& k0=1& k1=20, 30 Dec 2000 [26 Aug 2017].

Appendix A: Project plan

Week	**Task**
Week 1	Research forward error correction codes, review digital filters and think about the link between them
Week 2	Research forward error correction codes, review digital filters and think about the link between them
Week 3	Write literature review chapter (Chapter 2)
Week 4	Develop and test code construction functions
Week 5	Develop and test search script
Week 6	Tabulate search script results and investigate the constructed codes
Week 7	Write methodology of work done so far (Chapter 3)
Week 8	Research and think about ways to optimize search script and further investigate code construction with filters
Week 9	Optimize search script
Week 10	Develop and test FIR to IIR filter conversion algorithm
Week 11	Develop and test cyclic code search script
Week 12	Finish writing methodology (Chapter 3)
Week 13	Write results and conclusion chapters (Chapter 4 and 5)
Week 14	Finalise report and include study leader feedback
Week 15	Finalise report and include study leader feedback

Appendix B: ECSA outcomes

ECSA outcome	How the outcome has been met in this project
1. Problem solving. Identify, assess, formulate and solve convergent and divergent engineering problems	Problem solving demonstrated by successful completion of the project which required finding creative solutions to problems. Clear identification of problems to be solved / objectives to be achieved in Chapter 1 and 3. Solution approach formulated throughout Chapter 3, with consideration of, and references to, alternative approaches. Solution results presented in Chapter 4 and analysed in Chapter 5.
2. Application of Scientific and Engineering Knowledge	Engineering knowledge from both digital signal processing and coding theory was used - working across disciplinary boundaries. Mathematical knowledge was fundamental to the analysis and modeling of all the designs and to proofs and was used throughout the project. Computer science skills were used in the design, optimisation and, finally, implementation of algorithms in MATLAB. Relevant sections: Chapters 3, 4, 5.
3. Engineering design. Procedural and nonprocedural design and synthesis of components, works, products and processes	Algorithm design. Consideration of computational complexity and attempts at optimisation. Consideration of alternative approaches and consideration of Python as alternative development platform. Use of programming and mathematical skills for the design of algorithms to construct codes and perform operations such as finding IIR filter structures that construct the same codes as a specified FIR filter. Design of functions to investigate the FEC codes produced. Thorough documentation of design steps and code operation (Chapter 3, Appendix D).
4. Investigations, experiments and data analysis. Design and conduct investigations and experiments	Thorough background research done with literature review (Chapter 2). Development of functions to perform investigations (e.g. finding of largest minimum Hamming distance achieved, determining code cyclicity etc) required to achieve project objectives (Chapter 3, 4, Appendix C). Empirical observations based on results / data produced to perform further experiments and draw conclusions. Conclusions drawn based on this investigation in Chapter 5. Many other investigations were conducted which turned out fruitless and hence wasn't included in this report.

5. Engineering Methods, Skills and Tools, including Information Technology. Methods, skills and tools, including those based on information technology	The use of software development in MATLAB as a tool to solve problems and achieve the project objectives. Consideration of the applicability of MATLAB by also considering Python as an alternative option. Overall demonstration of skills in programming (Chapter 3). Critical analysis and interpretation of the results produced (Chapter 4, 5).
6. Professional and Technical Communication. Effective oral and written communication	Written communication through a professional, clear and well-written technical report with appropriate style for the audience. Explanations augmented with graphics, flowcharts, pseudocode and mathematical formulas to ensure clarity. Oral communication through final oral and exchanging ideas with study leader.
9. Independent learning ability. Independent learning through well-developed learning skills	This outcome was achieved by independently learning about forward error correcting codes (a topic which the author had no prior knowledge about) and signal processing through research, and the application of this knowledge to the project. The successful completion of this independent learning is shown by the literature study and the successful completion of the project using the knowledge obtained. Relevant sections: Chapter 2.

Appendix C: The $(8, 4)$ code not constructible by a FIR or IIR filter

The only cases for $n, k \leq 20$ where FIR and IIR filters failed to construct a binary code that achieved the largest known minimum distance achievable are the $(8, 4)$ and $(18, 9)$ codes. The largest known achievable distance for an $(18, 9)$ binary code is 6, but only a code with minimum distance 5 was achieved in this project. Similarly, the largest known achievable distance for an $(8, 4)$ binary code is 4, but only a code with minimum distance 3 was achieved. The construction of this $(8, 4)$ code with minimum distance 4 provided by [17] is investigated to determine why it could not be constructed by a digital filter. If the generator matrix construction steps provided by [17] are followed one achieves a generator matrix that constructs the following non-systematic codebook:

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

Since $k = 4$, the message book contains four shifted impulses: $(1, 0, 0, 0)$, $(0, 1, 0, 0)$, $(0, 0, 1, 0)$ and $(0, 0, 0, 1)$. If C is constructed with a FIR or IIR filter these messages would be encoded in the codebook as four shifted versions of the impulse response to the right. Therefore a necessary condition for a code to be constructible with a causal filter is that C should contain k non-circularly shifted versions of some codeword to the right, which is the encoded versions of the impulses (we already made this conclusion in Section 3.4). If this does not exist in C the code can't be produced by a causal filter. This is a necessary condition. It is not known if it is also is a sufficient condition. In the case of this $(8, 4)$ code this condition is not fulfilled: we can determine this from inspection of the given codebook. It is therefore clear that the code can not be produced by a causal filter.

From inspection we see that there are three rows in C which are non-circular shifts of each other

$$\begin{aligned} \mathbf{c}_7 &= (1, 1, 0, 0, 1, 1, 0, 0), \\ \mathbf{c}_4 &= (0, 1, 1, 0, 0, 1, 1, 0), \\ \mathbf{c}_2 &= (0, 0, 1, 1, 0, 0, 1, 1). \end{aligned}$$

If C also contained a codeword such that:

$$\mathbf{c}_i = (0, 0, 0, 1, 1, 0, 0, 1), \tag{5.1}$$

the condition could be fulfilled by a filter with an impulse response $h[n] = \{1, 1, 0, 0, 1, 1\}$. However, such a row does not exist in C. C does, however, contain the codeword $\mathbf{c} = (1, 0, 0, 1, 1, 0, 0, 1)$. The reader may now ask if it could perhaps be possible to circumvent the restriction by using a non-causal filter which can encode the fourth shifted impulse to $(1, 0, 0, 1, 1, 0, 0, 1)$. This is a valid question. The non-causal filter with the following impulse response does in fact encode $(0, 0, 0, 1)$ as $(1, 0, 0, 1, 1, 0, 0, 1)$, while also encoding $(1, 0, 0, 0)$ as $\mathbf{c}_7$, $(0, 1, 0, 0)$ as $\mathbf{c}_4$ and $(0, 0, 1, 0)$ as $\mathbf{c}_2$:

$$h[n] = \{1, 0, 0, \underset{\uparrow}{1}, 1, 0, 0, 1, 1, 0, 0\}, \tag{5.2}$$

where the up arrow indicates the position of $h[0]$.

To investigate this possibility a custom MATLAB implementation of a non-causal filter was written, since the standard libraries only support causal filters. Code was also written to construct a codebook using a non-causal filter. This is all included in Appendix D. However, this was futile, as the codebook constructed by the non-causal filter with the impulse response given by (5.2) is not valid: it contains duplicate codewords and therefore also has a minimum Hamming distance of zero.

Appendix D: Software code

Direct GF(2) implementation of discrete convolution and digital filters in Python with Numpy

```
breaklines
'''
Initial version created on July 28, 2017

@author: Benjamin du Preez
'''

import numpy as np

'''
Has the same behaviour as numpy's function numpy.convolve(a, v, mode='full'), but adapted for GF(2)

Input parameters are:
    a : (N,) array_like
    First one-dimensional input array.
    v : (M,) array_like
    Second one-dimensional input array.

Returns the discrete, linear convolution of two one-dimensional sequences.
The result returns the full convolution, i.e. the result has length
(N + M - 1)
'''
def convolve(a, v):
    N = len(a)
    M = len(v)

    # if v is longer than a, the arrays are swopped before computation -> numpy does the same
    if(M > N):
        temp = a
        a = v
        v = temp

    length = N + M - 1

    # declare empty array for result
    result = np.zeros(length, dtype=np.int)

    for n in range(0, length):
        # for each value of n -> multiply the overlapped parts of a and v elementwise and add them together (this is discrete convolution)
        for m in range(0, n+1):
            # if a[m] is out of range, the end of the overlapped part of  the sequences a and v has been reached
            # for the current value of n -> don't waste time, break this loop and go to the next value of n
            a_element = 0
            try:
                a_element = a[m]
            except IndexError:
                break

            # if v[n-m] is out of range, we havent' necessarily reached the end of overlapped part of  the sequences a and v!!
            # Because if v[n-m] is after the last element of sequence v[n-m], a larger
            # therefore there is currenrly no overlap but there might still be -> don't waste time, continue to go to the next value of m
            v_element = 0
            try:
                v_element = v[n-m]
            except IndexError:
                continue

            # the key difference -> use XOR instead of normal addition
            result[n] = result[n] ^ (a_element * v_element)
```

```
59
60          # throw an error if the user is trying to use numbers outside of GF(2)
61          if (result[n] > 1 or result[n] < 0):
62             raise ValueError('Numbers other than 0 or 1 do not exist inside the finite
                   Galois field GF(2)')
63
64    return result
65
66 '''
67 Has the same behaviour as scipy's function scipy.signal.lfilter(b, a, x), but adapted for
       GF(2)
68
69 Input parameters are:
70    b : array_like
71    The numerator coefficient vector in a 1-D sequence.
72    a : array_like
73    The denominator coefficient vector in a 1-D sequence. If a[0] is not 1, then both a
          and b are normalized by a[0].
74    x : array_like
75    An N-dimensional input array.
76
77 Filter data along one-dimension with an IIR or FIR filter. The filter output is returned.
78
79 The filter function is implemented as a direct II transposed structure. This means that
       the filter implements:
80
81 a[0]*y[n] = b[0]*x[n] + b[1]*x[n-1] + ... + b[M]*x[n-M]
82                       - a[1]*y[n-1] - ... - a[N]*y[n-N]
83
84 Note: Initial rest conditions is assumed, i.e. y[n] = 0 for n < 0 and x[n] = 0 for n < 0
85 '''
86 def lfilter(b, a, x):
87
88    length = len(x)
89    M = len(b) # order of transfer function numerator
90    N = len(a) # order of transfer function denominator
91
92    # declare empty array to hold output
93    y = np.zeros(length, dtype=int)
94
95    if a[0] == 0:
96       raise ValueError('a[0] has to be a non-zero number')
97
98    for n in range(0, length):
99       for i in range(0, min((n + 1), M)):
100         y[n] = y[n] ^ (b[i] * x[n - i])
101      for i in range(1, min((n + 1), N)):
102         y[n] = y[n] ^ (a[i] * y[n - i])
103
104   return y
```

gf2.py

MATLAB functions

```
breaklines
1  function [ achieves_griesmer_bound ] = achieves_griesmer_bound( n, k, dmin )
2  % This function checks if the code specified by the given codebook achieves
3  % the Griesmer bound
4
5  % ARGUMENTS
6  % Inputs:   - n, the code length
7  %           - k, the message length
8  %           - dmin, the minimum Hamming distance of the code
9  % Outputs:  - a boolean, achieves_griesmer_bound, indicating if the code achieves
10 %             the Griesmer bound
11
12 % get lhs of Griesmer bound inequality
13 lhs = n;
14
15 % get rhs of Griesmer bound inequality
```

```
16  sum = 0;
17  for i = 0:(k-1)
18      sum = sum + ceil(dmin/(2^i));
19  end
20  rhs = sum;
21
22  % lhs <= rhs, if lhs == rhs -> the code reaches the Griesemer bound
23  if lhs == rhs
24      achieves_griesmer_bound = true;
25  else
26      achieves_griesmer_bound = false;
27  end
28
29  end
```

achieves_griesmer_bound.m

```
    breaklines
1   function [ are_equivalent ] = are_codes_equivalent( C1, C2 )
2   % This function checks if the two codes C1 and C2 are equivalent, i.e. if
3   % their codebooks contain the same set codewords, even if the codewords may
4   % be in a different order
5
6   % ARGUMENTS
7   % Inputs:    - code matrix C1
8   %            - codebook matrix C2
9   % Outputs:   - a boolean, are_equivalent, indicating if C1 and C2 are
10  %              equivalent codes
11
12  % assume false initially, change if otherwise
13  are_equivalent = false;
14
15  % throw error if dimensions are not the same
16  if size(C1) ~= size(C2)
17     error('Error. The dimensions of the two codebooks must be equivalent');
18  % check if codebook matrices are equivalent after their rows are sorted
19  elseif isequal(sortrows(C1), sortrows(C2))
20      are_equivalent = true;
21  end
22
23  end
```

are_codes_equivalent.m

```
    breaklines
1   function [ iir_filters ] = find_all_equivalent_iirs( b_fir, n )
2   % Given a certain FIR filter with b coefficients b_fir, this function
3   % returns all IIR filter structures with an impulse response matching the
4   % FIR filter's impulse response up to element n
5
6   % ARGUMENTS
7   % Inputs:    - b_fir, the FIR coefficients of the filter who's impulse
8   %              response needs to be matched
9   %            - n, the code length, i.e. the point to where the impulse
10  %              response represented by b_fir needs to be matched
11  % Outputs:   - irr_filters, a 2^n-1 matrix holding all matching IIR filters found
12
13  % each row in this matrix holds an a and b coefficients vector pair
14  iir_filters = zeros(1, 2*n);
15
16  if n >= size(b_fir, 2)
17      impulse_response = cat(2, b_fir, zeros(1, n - size(b_fir, 2)));
18  else
19      impulse_response = b_fir(1:n);
20  end
21
22  rev_impulse_response = fliplr(impulse_response);
23
24  a_iir = zeros(1, n);
25  b_iir = zeros(1, n);
26  % a_0 is always = 1
27  a_iir(1) = 1;
28  % b_0 is = the first impulse response element
```

```
29 b_iir(1) = impulse_response(1);
30
31 determine_new_row_b_and_a(b_iir, a_iir, 2);
32 % this function is used recursively to work the solutions matrix from top
33 % to bottom. each possible new answer is followed up in it's own recursive
34 % function call. in the end all the matching irr filter coefficients are
35 % stored in a matrix
36     function determine_new_row_b_and_a(b_iir, a_iir, i)
37         % if i = n+1 then recursion needs to stop and the result saved
38         if i == n + 1
39             iir_filters = cat(1, iir_filters, cat(2, b_iir, a_iir));
40         else
41
42             % first check if the equation is satisfied when a and b = 0
43              if mod(b_iir(i) + sum(rev_impulse_response(n-i+2:n).*a_iir(2:i)), 2) ==
                    impulse_response(i)
44                  % if true, check if h_0 is = 0
45                  if rev_impulse_response(n) == 0
46                      % we can now satisfy the equation by setting a = 0 or 1
47                      % and setting b = 0 (only b influences the result since a is
                            multiplied with 0)
48
49                      determine_new_row_b_and_a(b_iir, a_iir, i+1); % b = 0, a = 0
50
51                      a_iir(i) = 1;
52                      determine_new_row_b_and_a(b_iir, a_iir, i+1); % b = 0, a = 1
53                  else
54                      % we can now satisfy the equation by setting a = b
55                      %(i.e. a = b = 0 or a = b = 1)
56
57                      determine_new_row_b_and_a(b_iir, a_iir, i+1); % b = 0, a = 0
58
59                      b_iir(i) = 1;
60                      a_iir(i) = 1;
61                      determine_new_row_b_and_a(b_iir, a_iir, i+1); % b = 1, a = 1
62                  end
63              else
64                  % if false, check if h_0 is = 0
65                  if rev_impulse_response(n) == 0
66                      % we can now satisfy the equation by setting a = 0 or 1
67                      % and setting b = 1 (only b influences the result since a is
                            multiplied with 0)
68
69                     b_iir(i) = 1;
70
71                     determine_new_row_b_and_a(b_iir, a_iir, i+1); % b = 1, a = 0
72
73                     a_iir(i) = 1;
74                     determine_new_row_b_and_a(b_iir, a_iir, i+1); % b = 1, a = 1
75
76                  else
77                      % we can now satisfy the equation by setting a != b
78                      %(i.e. a = 0, b = 1 or a = 1, b = 0)
79
80                     b_iir(i) = 1;
81
82                     determine_new_row_b_and_a(b_iir, a_iir, i+1); % b = 1, a = 0
83
84                     b_iir(i) = 0;
85                     a_iir(i) = 1;
86                     determine_new_row_b_and_a(b_iir, a_iir, i+1); % b = 0, a = 1
87
88                  end
89              end
90
91         end
92     end
93 iir_filters = iir_filters(2:end,:); % remove zero row at the top
94 end
```

find_all_equivalent_iirs.m

```
    breaklines
```

```
function [ C ] = gen_codebook( b, a, MB, n )
% This function generates the codebook of a code by encoding the given
% message book with the FIR/IIR filter with transfer function coefficients
% a and b. The given input length should be = k, output length = n

% ARGUMENTS
% Inputs:   - row vectors for filter coefficients b and a
%           - (2^k x k) message book matrix message_book
%           - length of output codes n (integer)
% Outputs:  - (2^k x n) codebook matrix C

% get the value of k
k = size(MB, 2);

% throw error if k > n
if k>n
    error('Error. Codeword length, n, may not be shorter than message length, k');
end

% generate empty matrix for codebook
C = zeros(2^k, n);

% append zeros to messages to ensure filter output is of length n
MB = cat(2, MB, zeros(2^k, (n - k)));

% encode message book by filtering each row and save in codebook
for i = 1:(2^k)
    C(i,:) = gffilter(b, a, MB(i,:));
end

end
```

gen_codebook.m

```
    breaklines
function [ C ] = gen_codebook_non_causal( h, zero_pos, MB, n )
% This function generates the codebook of a code by encoding the given
% message book with the non causal filter defined by the given impulse response.
% The given input length should be = k, output length = n

% ARGUMENTS
% Inputs:   - row vectors for impulse response h
%           - zero_pos the position of t=0 in the impulse response
%           - (2^k x k) message book matrix message_book
%           - length of output codes n (integer)
% Outputs:  - (2^k x n) codebook matrix C

% get the value of k
k = size(MB, 2);

% throw error if k > n
if k>n
    error('Error. Codeword length, n, may not be shorter than message length, k');
end

% generate empty matrix for codebook
C = zeros(2^k, n);

% append zeros to messages to ensure filter output is of length n
MB = cat(2, MB, zeros(2^k, (n - k)));

% encode message book by filtering each row and save in codebook
for i = 1:(2^k)
    C(i,:) = non_causal_filter(h, MB(i,:), zero_pos);
end


end
```

gen_codebook_non_causal.m

```
    breaklines
function [ b_mat, a_mat ] = gen_coefficients( M, N )
```

```
% This function generates all the possible filter coeffient combinations
% with numerator order M and denominator order N

% ARGUMENTS
% Inputs:    - numerator order N (integer)
%            - denominator order M (integer)
% Outputs:   - (2^(M+1) - 1, M+1) matrix b_mat with different b coefficient pairs as rows
%            - (2^N, N+1) matrix a_mat with different a coefficient pairs as rows

num_of_b_coefs = M+1;
num_of_a_coefs = N+1;

b_mat = gen_msg_book(num_of_b_coefs);
% slice off the first zeros row
b_mat = b_mat(2:size(b_mat, 1),:);

% a_0 should always be 1 -> generate a_1 to a_N first
if (N-1) > 0
    a_mat = gen_msg_book((num_of_a_coefs-1));
    % now append a_0 = 1 to the front of all the rows
    a_mat = cat(2, ones((2^(num_of_a_coefs-1)), 1), a_mat);
else
    a_mat = ones((2^(num_of_a_coefs-1)), 1);
end

end
```

gen_coefficients.m

```
    breaklines
function [ M ] = gen_msg_book( k )
% This function generates a message book for a binary code with the
% given message length k. The full message book with all possible 2^k
% binary messages is returned.

% ARGUMENTS
% Inputs:    - length of input messages k (integer)
% Outputs:   - (2^k x k) message book matrix M

% generate integer range from 0 to 2^k
nums = 0:(2^k-1);

% convert integers to binary vectors in GF(2)
M = de2bi(nums, k, 'left-msb');

end
```

gen_msg_book.m

```
    breaklines
function [ C ] = gen_non_causal_codebook( h, zero_pos, M, n )
%GEN_NON_CAUSAL_CODEBOOK Summary of this function goes here
%   Detailed explanation goes here

k = size(M, 2);

C = zeros(2^k, n);

for i = 1:2^k
C(i, :) = non_causal_filter(h, cat(2, M(i, :), zeros(1, n-k)), zero_pos);
end



end
```

gen_non_causal_codebook.m

```
    breaklines
function [ G ] = get_g_matrix( b, a, n, k )
% This function determines the generator matrix of a code

```

```
% ARGUMENTS
% Inputs:   - row vectors for filter coefficients b and a
%           - n, the code length
%           - k, the message length
% Outputs:  - (k x n) generator matrix G

% generate empty matrix for generator matrix
G = zeros(k, n);

impulse = cat(2, [1], zeros(1, n-1));
impulse_response = gffilter(b, a, impulse);

% construct generator matrix by shifting impulse response
row_i = impulse_response;
for i = 1:k
    G(i, :) = row_i;
    row_i = circshift(row_i, [1 1]);
    row_i(1) = 0; % replace the circularly wrapped part of the shift with 0
end

end
```

get_g_matrix.m

```
    breaklines
function [ dist ] = hamming_dist( a, b )
% This function finds the Hamming distance between binary vectors a and b
% of equal length and returns the result as an integer

% ARGUMENTS
% Inputs:   - row vectors a, b of equal length
% Outputs:  - the minimum Hamming distance between a and b (integer)

% throw error if a and b is not of equal length
if size(a, 2) ~= size(b, 2)
   error('Error. Vectors a and b must be of equal length.');
end

% for binary codes d(a, b) = w(a + b) -> use this property
% a + b
diff = gfadd(a, b);
% count number of ones in result, i.e. w(a + b)
dist = hamming_weight(diff);

end
```

hamming_dist.m

```
    breaklines
function [ weight ] = hamming_weight( a )
% This function finds the Hamming weight of a binary vector a and returns
% the result as an integer

% ARGUMENTS
% Inputs:   - row vector a
% Outputs:  - the Hamming weight of a (integer)

weight = sum(a == 1);

end
```

hamming_weight.m

```
    breaklines
function [ is_cyclic ] = is_cyclic( C )
% This function checks if the code specified by the given codebook is a
% a cyclic code.

% ARGUMENTS
% Inputs:   - (2^k x n) codebook matrix C
% Outputs:  - a boolean, is_cyclic, indicating if the code is cyclic

```

```
% initialize to true -> will be set false if necessary
is_cyclic = true;

height = size(C, 1);

% loop through the codewords (assuming first row of codebook is a zero row)
for i = 2:height
    % for codeword i, test if the cyclic shift of that codeword exists
    % in the codebook (i.e. it is also a codeword)
    %
    shifted_codeword = circshift(C(i, :), [1 1]);
    shift_exists = false;

    % check shifter codeword i against all other codes in codebook
    for j = 1:height
        % check shifted version of codeword i against other codeword j
        if shifted_codeword == C(j, :)
            % shifted codeword has been found
            shift_exists = true;
            % end the innder loop -> shifted version of codeword i has
            % been found in C -> can now move to codeword i+1
            break;
        end
    end
    % if the shifted version of codeword i has not been found after the
    % the whole codebook has been searched -> the code can not be a cyclic
    % code -> set is_cyclic to false, break outer loop and return
    if shift_exists == false
        is_cyclic = false;
        break;
    end
end

end
```

is_cyclic.m

```
    breaklines
function [ is_cyclic ] = is_cyclic_2( b, a, n )
% This function checks if the codes of length n created by the given b and a
% coefficients are cyclic codes.

% ARGUMENTS
% Inputs:   - b, a the coefficients of the filter
% Outputs:  - a boolean, is_cyclic, indicating if the code is cyclic

impulse_resp = gffilter(b, a, cat(2, 1, zeros(1, n-1))); % impulse response (generator
    polynomial)
xn1 = cat(2, cat(2, 1, zeros(1, n-1)), 1); % x^n - 1
[~, remd] = gfdeconv(xn1, impulse_resp);

if remd == 0
    is_cyclic = true;
else
    is_cyclic = false;
end

end
```

is_cyclic_2.m

```
    breaklines
function [ is_perfect_code ] = is_perfect_code( n, k, dmin )
% This function checks if the code specified by the given codebook achieves
% the Hamming bound

% ARGUMENTS
% Inputs:   - n, the code length
%           - k, the message length
%           - dmin, the minimum Hamming distance of the code
% Outputs:  - a boolean, is_perfect_code, indicating if the code achieves
%             the Hamming bound

```

```
% get code dimensions and distance

% get lhs of Hamming bound inequality
lhs = 2^k;

% get rhs of Hamming bound inequality
t = floor((dmin - 1)/2);
sum = 0;
for i = 0:t
    sum = sum + nchoosek(n, i);
end
rhs = (2^n)/(sum);

% lhs <= rhs, if lhs == rhs -> the code reaches the Hamming bound
if lhs == rhs
    is_perfect_code = true;
else
    is_perfect_code = false;
end

end
```

is_perfect_code.m

```
    breaklines
function [ Gout ] = make_g_systematic( Gin )
% This function accepts a generator matrix and puts it in the standard
% sytematic form. Any linear block code has an equivalent G matrix in
% systematic form

% ARGUMENTS
% Inputs:   - (k x n) generator matrix G
% Outputs:  - (k x n) generator matrix G in systematic form

Gout = mod(rref(Gin), 2); % Gaussian elimination method

% Uncomment for the secondary method
%n = size(Gin, 2);
%k = size(Gin, 1);
%B = Gin(:,(n-k+1):n);
%Gout = mod(inv(B)*Gin, 2); % muliply inverse B with G

end
```

make_g_systematic.m

```
    breaklines
function [ min_dist ] = min_hamming_dist( C, is_lin_block )
% This function calculates the minimum Hamming distance of a given codebook
% C. If the optional parameter is_lin_block_code is set to true, the
% function uses the following property of linear block codes:
% d_min(C) = min_w(C) to speed up the function execution time

% ARGUMENTS
% Inputs:   - codebook matrix C
%           - optional boolean is_lin_block specifying if C is a linear
%             block code
% Outputs:  - the minimum Hamming weight of C (integer)

% check if the optional parameter was passed and set to true
if nargin == 2
    if is_lin_block == true
        min_dist = min_hamming_weight(C);
    end
else
    % get dimensions of the codebook C
    height = size(C, 1);
    width = size(C, 2);
    % initialise min to the largest possible value
    min_dist = width;

    % loop through codes
    for i = 1:height
```

```
        % for each code i, loop through all other codes j
        for j = 1:height
            % don't compare with self
            if i==j
                continue;
            end
            % get Hamming distance between code i and j
            curr_dist = hamming_dist(C(i,:), C(j,:));
            if curr_dist < min_dist
                min_dist = curr_dist;
            end
        end
    end
end


end
```

min_hamming_dist.m

```
    breaklines
function [ min_weight ] = min_hamming_weight( C )
% This function finds the minium Hamming weight of code C, i.e. the
% non-zero row in C with the smallest Hamming weight

% ARGUMENTS
% Inputs:   - codebook matrix C
% Outputs:  - the minimum Hamming weight of C (integer)

height = size(C, 1);
width = size(C, 2);
% initialise min_weight to the largest possible value it can be
min_weight = width;

% loop through rows
for i = 1:height
    % find hamming weight of current row
    curr_weight = hamming_weight(C(i, :));
    % if hamming weight of the current non-zero row is smaller than the
    % previous minimum record -> update minimum
    if curr_weight < min_weight && curr_weight ~= 0
        min_weight = hamming_weight(C(i, :));
    end
end

end
```

min_hamming_weight.m

```
    breaklines
function [ y ] = non_causal_filter( h, x, zero_pos )
% This function implements a non-causal filter with impulse response h
% The output of the filter is returned starting from time 0 and with the
% same length as the input

% ARGUMENTS
% Inputs:   - x, the data to be filtered
%           - h, the impulse response of the non-causal filter
%           - zero_pos is an integer that indicates the position in the
%           given impulse response sequence h[n] that represents n = 0
% Outputs:  - y, the filtered data of the same length as x

y = gffilter(h, 1, cat(2, x, zeros(1, size(x, 2))));
y = y((zero_pos):(zero_pos + size(x, 2) - 1));
y = mod(y, 2);

end
```

non_causal_filter.m

MATLAB scripts

breaklines

```
% Don't extract generator matrices or do cyclicity checks etc except for
% record codes. Only check for FIR filter coefficients (impulse responses)

% specify input parameters
n = 10; % code length
k = 6; % dimension

% generate message book with given parameters
MB = gen_msg_book(k);

largest_min_hamming_dist = 0;

% each row in this matrix holds a b coefficients vector
record_coeffs = zeros(1, n);

[b_mat, a_mat] = gen_coefficients(n-1, 1);

tic
% loop through coefficients
for i = 1:(2^n - 1)
    b_curr = b_mat(i, :);
    a_curr = 1;

    % generate code, codeword by codeword
    dmin = n;
    for j = 2:(2^k)
        % append zeros to messages to ensure filter output is of length n
        msg = cat(2, MB(j,:), zeros(1, (n - k)));
        % encode msg j in msg book and get codeword weight
        cw_weight = hamming_weight(gffilter(b_curr, a_curr, msg));
        % check hamming weight -> update minimum distance of code
        if cw_weight < dmin%(cw_weight ~= 0)&&(cw_weight < dmin)
            dmin = cw_weight;
        end
        % if dmin < largest min distance we already know this
        % code won't achieve record min distance -> don't have to
        % check rest of msg book
        if dmin < largest_min_hamming_dist
            break;
        end
    end

    % update largest minimum Hamming distance achieved
    if dmin > largest_min_hamming_dist
        record_coeffs = b_curr;
        largest_min_hamming_dist = dmin;
    % another code also achieved this Hamming distance
    elseif dmin == largest_min_hamming_dist
        record_coeffs = cat(1, record_coeffs, b_curr);%cat(2, b_curr, a_curr));
    end
end
toc
disp(['Largest minimum Hamming distance achieved for n = ', num2str(n), ' and k = ',
    num2str(k), ' is ', num2str(largest_min_hamming_dist)]);
disp(['The number of FIR filters that achieve this Hamming distance: ', num2str(size(
    record_coeffs, 1))]);
disp(' ');
disp('The first one found is shown below:');
disp(['Is perfect code / reaches Hamming bound (1 = true, 0 = false): ', num2str(
    is_perfect_code(n, k, largest_min_hamming_dist))]);
disp(['Reaches Griesmer bound (1 = true, 0 = false): ', num2str(achieves_griesmer_bound(n
    , k, largest_min_hamming_dist))]);
disp(' ');
disp('Transfer function coefficients (b and then a):');
disp(record_coeffs(1,:));
disp('Filter transfer function, H[z]:');
filt(record_coeffs(1,:), 1)
disp(' ');
disp('Generator matrix:');
record_G = get_g_matrix(record_coeffs(1,:), 1, n, k);
```

```
67 disp(record_G)
68 disp('Systematic generator matrix:');
69 disp(make_g_systematic(record_G));
```

search_script_fir.m

```
    breaklines
1  % The same as search_script_fir, but coefficients achieving the largest
2  % minimum distance aren't saved, in an effort for better speed
3
4  % specify input parameters
5  n = 8; % code length
6  k = 4; % dimension
7
8  % generate message book with given parameters
9  MB = gen_msg_book(k);
10
11 largest_min_hamming_dist = 0;
12
13 [b_mat, a_mat] = gen_coefficients(n-1, 1);
14
15 tic
16 % loop through coefficients
17 for i = 1:(2^n - 1)
18     b_curr = b_mat(i, :);
19     a_curr = 1;
20
21     % generate code, codeword by codeword
22     dmin = n;
23     for j = 2:(2^k)
24         % append zeros to messages to ensure filter output is of length n
25         msg = cat(2, MB(j,:), zeros(1, (n - k)));
26         % encode msg j in msg book and get codeword weight
27         cw_weight = hamming_weight(gffilter(b_curr, a_curr, msg));
28         % check hamming weight -> update minimum distance of code
29         if cw_weight < dmin
30             dmin = cw_weight;
31         end
32         % if dmin <= largest min distance we already know this
33         % code won't improve record min distance -> don't have to
34         % check rest of msg book
35         if dmin <= largest_min_hamming_dist
36             break;
37         end
38     end
39
40     % update largest minimum Hamming distance achieved
41     if dmin > largest_min_hamming_dist
42         largest_min_hamming_dist = dmin;
43     end
44 end
45 toc
46 disp(['Largest minimum Hamming distance achieved for n = ', num2str(n), ' and k = ',
       num2str(k), ' is ', num2str(largest_min_hamming_dist)]);
```

search_script_fir_dist_only.m

```
    breaklines
1  % Strategy: first find all the FIR filters achieving optimal codes using
2  % exactly the same method as in search_script_fir. Then use the function
3  % find_all_iir_filters to find all the corresponding IIR filters.
4  % Do this multiple times, keeping k fixed and varying n and checking if one
5  % one IIR filter is every present
6
7  % specify input parameters
8  n_max = 15; % maximum code length to consider before terminating the search
9  k = 3; % dimension
10
11 % generate message book with given parameters
12 MB = gen_msg_book(k);
13
14 % each row in this matrix holds a b coefficients vector
15 record_coeffs = zeros(1, k);
```

```
16
17  % \\\\\\\\\\\ execute the search algorithm for different code lengths \\\\\
18  for n = k:n_max
19
20      largest_min_hamming_dist = 0;
21
22      % save record_coeffs of previous search
23      prev_coeffs = record_coeffs;
24      % clear record_coeffs
25      record_coeffs = zeros(1, n);
26
27      [b_mat, a_mat] = gen_coefficients(n-1, 1);
28
29      % \\\\\\\\\\\\ find fir filters for k and current n \\\\\\\\\\\
30      % loop through coefficients
31      for i = 1:(2^n - 1)
32          b_curr = b_mat(i, :);
33          a_curr = 1;
34
35          % generate code, codeword by codeword
36          dmin = n;
37          for j = 2:(2^k)
38              % append zeros to messages to ensure filter output is of length n
39              msg = cat(2, MB(j,:), zeros(1, (n - k)));
40              % encode msg j in msg book and get codeword weight
41              cw_weight = hamming_weight(gffilter(b_curr, a_curr, msg));
42              % check hamming weight -> update minimum distance of code
43              if cw_weight < dmin%(cw_weight ~= 0)&&(cw_weight < dmin)
44                  dmin = cw_weight;
45              end
46              % if dmin < largest min distance we already know this
47              % code won't achieve record min distance -> don't have to
48              % check rest of msg book
49              if dmin < largest_min_hamming_dist
50                  break;
51              end
52          end
53
54          % update largest minimum Hamming distance achieved
55          if dmin > largest_min_hamming_dist
56              record_coeffs = b_curr;
57              largest_min_hamming_dist = dmin;
58          % another code also achieved this Hamming distance
59          elseif dmin == largest_min_hamming_dist
60              record_coeffs = cat(1, record_coeffs, b_curr);%cat(2, b_curr, a_curr));
61          end
62      end
63
64      % check if any of the new found optimal FIRs are extended versions of
65      % previous optimal FIR filters, i.e. they new filter matches the impulse
66      % respons of the old up to sample n-1
67      % keep these rows and discard all the others
68      if n ~= k % don't check the first row
69
70          % get the indices of the rows in the two matrices where they match up to
71          % sample n - 1
72          [ia, ib] = ismember(record_coeffs(:, 1:n-1), prev_coeffs, 'rows');
73          % get those rows out of record coeffs
74          record_coeffs = record_coeffs(ia, :);
75
76
77          % if no such rows are found terminate the search -> we already
78          % know no IIR filters exist common to all optimal codes over the given
79          % range
80          if size(record_coeffs, 1) == 0
81              break;
82          end
83      end
84
85  end
86
87  % if record_coeffs still contain something, the IIR filters that represent
88  % this FIR filter will produce an optimal code for over the entire range of n
```

```
% with k specified
if size(record_coeffs, 1) ~= 0
    disp([num2str(size(record_coeffs, 1)), ' repeating impulse response pattern has been found over the specified range. Its equivalent IIR filter structures are saved in super_iirs']);
    super_iir_filters = find_all_equivalent_iirs(record_coeffs(1,:), size(record_coeffs, 2));
else
    disp('No results sorry!');
end
```

search_for_reusable_iir_filters_script.m

```
    breaklines
% Given the order, n, of the polynomial x^n + 1, this function finds all
% the binary polynomials of order n-k that factorise it. All these polynomials
% can be used as generator polynomials to generate cyclic (n,k) codes and,
% in other words, any filter having their coefficients as its impulse response
% up to the n'th sample will also generate a cyclic code

% polynomials are represented by their coefficients in the form:
% 1 + x + x^2 + x^3 + . . . + x^n, where n is the order

% Finally, the script then also constructs codes from all these cyclic
% impulse responses using FIR filters and see what the best Hamming
% distance achieved is

n = 7; % the order of the polynomial x^n + 1
k = 4; %k, the message length
factor_order = n - k; % the order of the desired generator polynomials

% empty matrix which will contain the answers as rows
cyclic_impulse_responses = zeros(1, n-k+1);

xn = cat(2, 1, cat(2, zeros(1, n-1), 1)); % the polynomial x^n+1

% all the possible answers are polynomials of order n-k, i.e. the
% coefficients are a vector of length n-k+1, with the last entry always
% equal to 1. use gen_coefficients and fliplr to generate this
[b, a] = gen_coefficients(1, n-k);
possibilities = fliplr(a);

% check which of
% these polynomials divide x^n+1 and save them in answers
for j = 1:size(possibilities,1)

    [quot,remd] = gfdeconv(xn, possibilities(j,:));

    if remd == 0 % the polynomial divides x^n+1
        cyclic_impulse_responses = cat(1, cyclic_impulse_responses, possibilities(j,:));
    end
end
% delete zero row introduced by initialisation
cyclic_impulse_responses = cyclic_impulse_responses(2:end,:);

if(size(cyclic_impulse_responses,1) == 0)
    disp('No cyclic codes found.');
end

% find largest minimum Hamming distance achieved by one of the cyclic codes
% constructed from the impulse responses that were found
record_min_distance = 0;
msg_book = gen_msg_book(k);
record_coeffs = zeros(1, n); % each row in this matrix holds a b coefficients vector

for m = 1:size(cyclic_impulse_responses,1)
    C = gen_codebook(cyclic_impulse_responses(m,:), 1, msg_book, n);
    min_distance = min_hamming_dist(C, true);

    if min_distance > record_min_distance % record broken
        record_min_distance = min_distance;
        record_coeffs = cyclic_impulse_responses(m,:);
    elseif min_distance == record_min_distance % record tied
```

```
60          record_coeffs = cat(1, record_coeffs, cyclic_impulse_responses(m,:));
61      end
62  end
```

find_cyclic_codes_script.m