GENERALISED TYPE-II HYBRID AUTOMATIC REPEAT REQUEST SCHEMES and KM CODES

Isobel McFarlane

Submitted for the degree of M.Sc. at The University of Glasgow

in the Department of Mathematics,

October 1991.

ProQuest Number: 13834293

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 13834293

Published by ProQuest LLC (2019). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Thesis
9189
Copy 2

GLASGOW UNIVERSITY LIBRARY

CONTENTS

CONTENTS

CHAPTER 1 - Introduction

In this work an error control scheme, *Generalised Hybrid Type-II ARQ (GH-ARQ)* is discussed. We then study a recently introduced class of linear codes called *KM codes* discovered by Krishna and Morgera. Finally, we observe how KM codes may be employed in GH-ARQ schemes.

However, to begin we shall briefly mention linear codes and some important factors concerning error control in digital communications systems.

1.1 - Codes, Error-Detection & Error-Correction

1.1.1 - Why do we need Error-Correcting Codes ?

It is unavoidable that from time to time, interference in the transmission channel will produce errors in the transmitted signal.

Speech channels do not usually require correction since there is sufficient redundancy in speech to allow for these errors.

However, in digital communications systems, this interference will result in the receipt of erroneous data. Hence error-correcting codes are used. These encode the message so that after transmission and possible distortion, the original information can still be recovered.

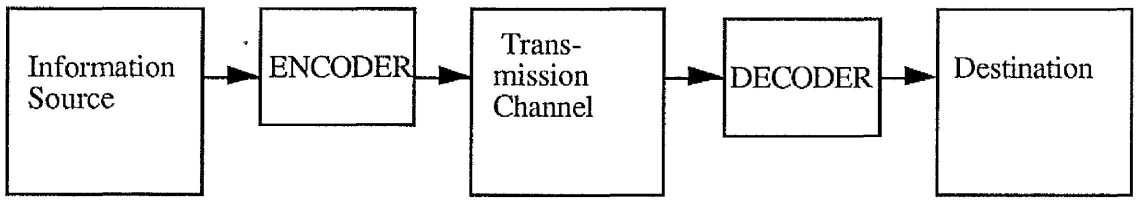


Figure 1.1- Basic Elements of Telecommunications System (using an error-correcting code).

Figure 1.1 above illustrares the main features of a communications system. The information to be delivered is encoded and then the transmitter sends it along the transmission channel. It is at this point that the message is in danger of being corrupted.

Upon arrival, the message is checked for errors. If errors are found to be present, the receiver will deal with these as specified by the protocol. In some schemes errors are corrected while in others the receiver requests that the message be retransmitted.

There are two fundamemtal techniques for error control in digital communications systems, namely *forward error control (FEC)* schemes and *automatic repeat request (ARQ)* schemes. These shall be discussed in Chapter 2.

1.1.2 - Block & Convolution Codes

Although, there are many different types of codes, in general, codes may be divided into two classifications :-

(a) *Block Codes* - The information is presented as k bits. This is encoded into an n-bit codeword. The $(n\text{-}k)$ additional bits, called *parity check digits* enable error-detection and correction. Each individual packet is independent of all other packets.

(b) *Convolution Codes* - Codewords of a convolution code are formed in the same way as for the block code. However, in addition, there is a dependency between successive incoming frames. The *memory order* of these codes is the number of previous frames having an influence on the way in which the present frame was encoded.

By adding parity-check bits, codes will provide the dedundancy required for error-detection and correction. The error-detecting capability of a code depends on the number of parity bits. Clearly these parity bits reduce the rate at which 'real' data is transmitted. Error-correction is a more difficult process and so additional bits are needed. Thus codes can correct fewer errors than they detect.

1.1.3 - Errors

If $\boldsymbol{c}$ is a codeword and $\boldsymbol{y}$ is the corresponding received codeword after transmission through a possibly 'noisy' channel, then $\boldsymbol{e} = \boldsymbol{y} - \boldsymbol{c}$ is called the *error vector*. When $\boldsymbol{e} = 0$, then no errors have occurred. Otherwise errors have taken place. Errors may be divided into two types :

(a) *Random Errors* - A random error is an isolated erroneous bit in a sequence of correct bits. The probabilty of error is the same for all bits regardless of whether the previous bits were correct or not.

(b) *Burst Errors* - Errors of this type occur in blocks. Usually a stream of good bits (low bit error rate) is followed by a stream in which the occurence of errors is great (high bit error rate), followed by a stream of good bits.

An error burst of length h will be defined to be a sequence of h error symbols; the first and the last of which are non-zero.

1.1.4 - Linear Codes

A very important class of codes is linear codes. We shall 'remind' the reader of some of the basic properties these codes pocess.

A *linear code* C is a linear subspace of F^n, some field F. If C has dimension k then C is called an (n, k) code. The information is presented in a block of k-bits which is encoded into a longer word of n-bits, a codeword. The $(n\text{-}k)$ parity bits enable error-detection and correction.

Let C be an (n, k) linear code, then we may define the following.

Definition 1.1

(a) G, the *generator matrix* of code C is a (k x n) matrix for which the rows are a basis of C. i.e. every codeword is a unique linear combination of the rows of G. If $\boldsymbol{u} = (u_1, u_2, \ldots, u_k)^T$ then the following simple rule maps messages $\boldsymbol{u}$ into codewords $\boldsymbol{c} = (c_1, c_2, \ldots, c_n)^T$:

$$\boldsymbol{c}^T = \boldsymbol{u}^T G.$$

(b)An (n-k) x n matrix H such that $\boldsymbol{c}^T H^T = 0$ iff $\boldsymbol{c} \in C$ is called a *parity-check matrix* of C. $\boldsymbol{c}^T H^T$ is called the *syndrome* of $\boldsymbol{c}$.

Note A received codeword is assumed error-free only if its syndrome is zero.

Definition 1.2

(a) The *rate* of C is k/n.

(b) C is *invertible* if knowing only the (n-k) parity bits the associated k information bits can be uniquely recovered.

Definition 1.3

(a) Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be codewords of C, then the *hamming distance* is defined to be the number of places in which $\boldsymbol{x}$ and $\boldsymbol{y}$ differ.

(b) The *minimum distance* of C is the least distance between any two distinct codewords.

(c) The *weight* of a non-zero codeword is the number of non-zero digits it has.

(d) The least weight of any non-zero codeword in C is equal to the minimum distance of C.

Lemma 1.1

Let d denote the minimum distance of C, it can be shown that :

(a) C can detect up to d-1 errors

and

(b) C can correct e errors if $d \geq 2e + 1$.

1.2 - Error Control in Digital Communications

We have already stated that FEC and ARQ are the 2 basic techniques for error control in digital communications. Both these schemes shall be studied in Chapter 2. However, first we must define some very important terms related to communications systems.

Definition 1.4

A *buffer* is used to store information which may be required later. In most error control systems, both the receiver and the transmitter have a buffer and systems are chosen to reduce the buffer size where possible.

Definition 1.5

We require to study and compare the performance of different error control schemes. Two important measures of system performance are :

(a)

$$\begin{array}{c}\textit{Throughput}\\ \textit{Efficiency}\end{array} = \frac{\text{average no. of data bits accepted}}{\text{total no. data bits transmitted}}$$
per unit of channel time

and

(b)

$$\textit{Reliability} = \frac{\text{probability of the occurrence of undetected errors}}{\text{probability that decoding succeeds}} .$$

1.3 - Shift Registers[1]

In Chapter 6, we shall see how shift registers may be used in the encoding of KM codes. Below, the basic elements of shift registers are mentioned and then we illustrate that shift registers may be used to multiply or divide one polynomial by another.

In these linear switching circuits, the information is assumed to be some representation of the elements of $GF(q)$. There are three types of devices commonly used in these circuits, namely

(a) An *adder* - This has two inputs and one output. The output is the sum of the two inputs modulo q,

(b) A *storage* or *delay unit* - This device has only one input and one output. It is a delay device in that the output after one unit of time is equal to the input of the previous unit of time

and

(c) A *multiplier* - Again, this has only one input and one output. The output in this case being simply the input multiplied by a constant a, where $a \in GF(q)$.

The above devices may be used to construct a shift register. In a shift register, there is a *shift signal* which causes the element stored in a delay unit to shift to the next delay unit. The entry of the final delay device is simply output.

When the input or the output is a polynomial, only the coefficients appear and the high-order coefficients are transmitted first.

e.g. If $h(u) = h_0 + h_1u + h_2u^2 + \ldots + h_{r-1}u^{r-1} + h_ru^r$, then h_r would be input first, followed by $h_{r-1}, \ldots$, followed by h_1 and the final input coefficient would be h_0.

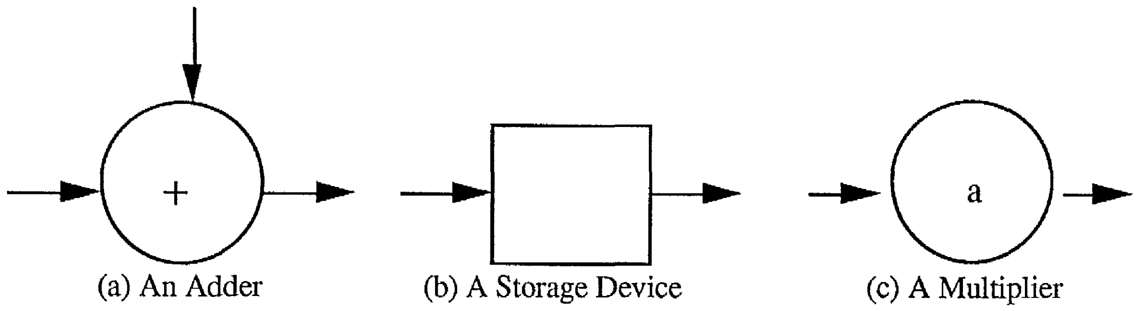


Figure 1.2 - 3 Common Devices Of Linear Switching Systems.

1.3.1 - Multiplier & Division Circuits

The circuit shown in the Figure 1.3, will multiply any polynomial $a(u) = a_ku^k + a_{k-1}u^{k-1} + \ldots + a_1u + a_0$ by the fixed polynomial $f(u) = f_ru^r + f_{r-1}u^{r-1} + \ldots + f_1u + f_0$.

Initially, the storage elements are assumed to contain zeros. The coefficients of $f(u)$ are assumed to enter high-order first followed by r zeros. The first input is the coefficient a_k resulting in an initial output of a_kf_r. At this stage all the storage devices contain zeros. After one shift signal, a_{k-1} is the new input while a_k is stored in the first device. This results in the output $a_{k-1}f_r + a_kf_{r-1}$. Clearly after the next time unit, a_{k-1} is stored in the first device, a_k in the second and a_{k-2} is the current input. The output will be $a_{k-2}f_r + a_{k-1}f_{r-1} + a_kf_{r-2}$. The shift register continues in this manner until $r + k$ shifts have occured. After this final shift, the shift register contains $0, 0, \ldots, 0, a_0$ and the final output is a_0f_0.

From the above, it is obvious that the first output a_kf_r is the first coefficient of $a(u)f(u)$. Similarly, the second output is the second coefficient of the product $a(u)f(u)$, . . . etc. Hence the shift register does indeed multiply any polynomial by $f(u)$.

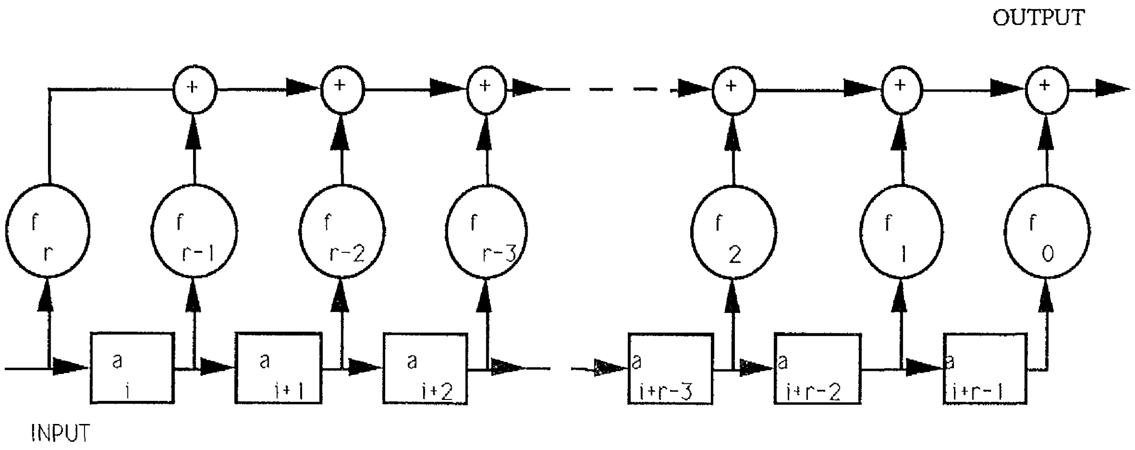


Figure 1.3 - A Multiplier Circuit.

We are also able to construct shift register circuits which perform division by a fixed polynomial. A circuit for dividing $d(u) = d_n u^n + \ldots. + d_0$ by $p(u) = p_r u^r + p_{r-1} u^{r-1} + \ldots. + p_1 u + p_0$ is shown below.

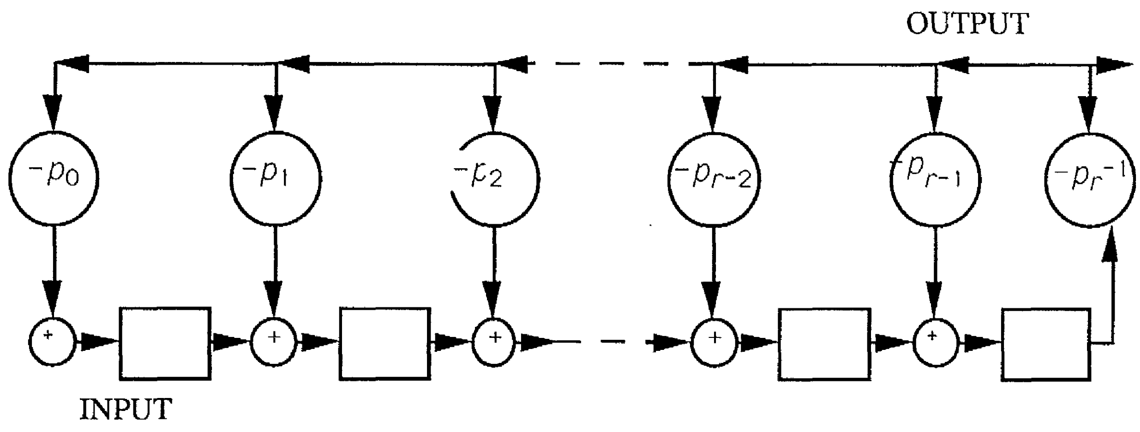


Figure 1.4 - A Division Circuit.

All storage devices are initially set to zero and consequently the output of each of the first r shifts will be zero.

The first non-zero output will be $d_n p_r^{-1}$, the first coefficient of the quotient.

For each quotient q_j, the polynomial $q_j p(u)$ must be subtracted from the dividend. This is dône using feedback connections.

After n shifts, the complete quotient has appeared as output and the remainder is stored in the shift register.

CHAPTER 2 - Digital Communications Systems

As stated in Chapter 1, there are two fundamental techniques for error control in digital communications - FEC and ARQ. In FEC schemes, one code is employed for both error-detection and correction. However, in some circumstances, when a message is found to contain errors, it is more reasonable, simply for the information to be retransmitted. This idea forms the basis of ARQ schemes. In this chapter both these schemes are described briefly. Their relative advantages and disadvantages are also discussed. As a measure of their performance, we consider their throughput efficiency and system reliability (see Definition 1.5).

However, sometimes neither of these schemes are suitable - perhaps the channel is too 'noisy' to guarantee the required throughput using ARQ while the desired system reliability cannot be achieved by FEC alone. One solution is to employ a combination of ARQ and FEC. Schemes of this type are known as *Hybrid ARQ* schemes. Section 2.2 discusses Hybrid ARQ in more detail.

Finally, a recently introduced error control procedure, *Generalised Hybrid Type-II ARQ* (*GH-ARQ*) is presented in Section 2.3 [2],[3]. In later chapters, codes particularly suited to GH-ARQ will be developed.

Throughout Chapter 2, let T denote the transmitter and R denote the receiver.

2.1 - Fundamental Error Control Techniques

2.1.1 - FEC - Forward Error Control

An error-correcting code is employed in this scheme and so the bit redundancy together with mathematically based encoding/decoding procedures enable all errors to be corrected provided that channel conditions are within some tolerance.

A k-bit message D is encoded into a n-bit codeword I using a (n, k) linear code. I is then transmitted. Let I^ denote the received codeword.

On receiving I^, R computes its syndrome. If this is found to be zero, then I^ is assumed to be error-free i.e. I = I^ and I^ is delivered to the data bank. Otherwise, the code attempts to correct the errors in I^ and the corrected block is sent to the data bank.

However, in some cases the code will be unable to correct the errors and so an erroneous block will be sent to the data bank.

FEC has large overheads since the code is used for both error-detection and correction. As there are no retransmissions, it has a high throughput efficiency. This is set by the code rate k/n and so is constant and independent of the channel conditions. However, since it is possible for incorrect data to be accepted it has low system reliability especially in very 'noisy' conditions. Clearly, the reliability is strongly dependent on the channel conditions. If a great deal is known about the channel, it is possible to choose a code with an appropriate error-correcting capability in order to maintain an acceptable level of reliability [4].

2.1.2 - ARQ - Automatic Repeat Request

This scheme requires a high rate error-detecting code. A message is encoded at T and the codeword is then transmitted. At R, the received codeword is tested for errors. However no error-correction is performed at R.

If the received codeword is found to be error-free then it is delivered to the data bank and R sends a *positive acknowledgement* (*ACK*) to T. When T gets this ACK, a new block is transmitted. If, however errors are detected in the received codeword, R discards it and awaits the retransmission of the codeword. This retransmission process continues until the block is successfully received.

Erroneous data is delivered to data bank only when R fails to detect the presence of errors and so is much more reliable than FEC schemes. The throughput efficiency depends largely on the number of requested retransmissions which in turn is controlled by the channel quality. Hence the efficiency falls rapidly with increasing channel error rate. There are many different types of ARQ schemes based on use of buffer storage and efficient use of the transmission channel. Below is a description of two of the most commonest forms.

(a) *Idle RQ* (*Send & Wait*) - This is by far the simplest ARQ scheme. T sends an information frame and initialises a timer. A copy of this frame is stored in T's buffer.

On receiving the *I* - frame, R checks for errors. If the frame is found to be error-free then it is delivered to the data bank and ACK is sent to T. When T gets this ACK, it then transmits the next frame. On the other hand, if the received *I* - frame is found to be erroneous, then R simply discards it.

If after the time-out period, T has not received ACK (either because the received *I* - frame contained errors or because ACK itself was corrupted) then T retransmits the same frame. To avoid duplicates, which could occur when ACK is corrupted, R must check that a new frame is distinct from previously received ones. Sequence numbers are used to distinguish frames - R keeps a record of the sequence number of last frame accepted and if it receives a duplicate *I* - frame it will discard it.

In this system, flow control is strict across the link (T can only have one frame awaiting ACK at a given moment). The main drawback with this protocol is that the time between the transmission of a block and the receipt of ACK is wasted time. For T, the idle time is at least equal to the round-trip delay time. For satelitte applications where the delay is approximately 1/4 second, such time waste is significant. This scheme could be improved by extending the protocol to include the transmission of *NACKs* (*negative acknowledgements*) when errors are found in a received block and so cutting the time delay. A clear advantage of Idle RQ is that minimal buffer storage is required at both T and R.

(b) *Continuous RQ* - With this scheme, datablocks are continuously sent off by T without waiting for individual responses from R. T retains a copy of each *I* - frame it sends in a retransmission list and sequence numbers are used to identify different frames.

At R, the blocks are tested for errors. For each correctly received *I* - frame, R returns ACK together with the corresponding sequence number. R retains an ordered list containing the sequence numbers of the last correctly received *I* - frames.

On receipt of ACK, T deletes the corresponding *I* - frame from the retransmit list. Suppose errors do occur. Then either T detects out of sequence ACKs or R detects the receipt of out of sequence *I* -frames. There are two different protocols which can be used :-

(i) *Selective Retransmission* - T must process the ACKs to determine if any frames in the retransmission list have not yet been acknowledged. If T receives ACK for the (*N*+1)th frame but the *N*th frame has not been acknowledged then T resends a copy of frame *N*. It is posssible that either the *N*th frame was corrupted during transmission or simply that ACK corresponding to frame *N* was corrupted on its way to T. So on getting a frame, R must check the receive list to determine whether or not this frame has already been correctly received. If a received frame is found to be a duplicate it is discarded. However, for every correctly received frame (whether it is an original or a duplicate), R must acknowledge its receipt to ensure that it is removed from the retransmission list at T.

This error control protocol ensures that exactly one copy of each *I* - frame is correctly received. However the ordering is not maintained. This is acceptable if each frame is a self contained packet. Often, a frame is part of a larger message and so we would have to buffer out of sequence frames at R before reassembling the message. Thus, there are substantial buffering overheads at R.

(ii) *Go-Back-N* - When R detects an out of sequence error-free *I* - frame, it requests T to retransmit all *I* - frames since the last correctly received one. i.e. if the (*N*+1)th frame is corrupted, then R will find the (*N*+2)th frame out of sequence. R then returns an NACK together with the sequence number of the last correctly received frame (frame *N* in this case). R discards frames (*N*+3), (*N*+4), ... and all successors until the correct receipt of frame (*N*+1). It then continues normally in sequence. If ACK for the *N*th frame is corrupted but ACK for the (*N*+1)th frame is correctly received by T, both frames *N* and (*N*+1) can be removed from the retransmission list - since R ACKs in strict order.

G-B-*N* maintains frame sequence and therefore reduces the buffering requirements at R. However, as the retransmission of correct frames is increased, the throughput efficiency is reduced.

Clearly continuous repeat request schemes improve link utilisation by allowing T to send multiple frames before receiving ACK/NACKs. This is at the expense of increased buffering. Link efficiency can be further improved by Piggybacking i.e. *I* - frames flow in both directions of the link simultaneously and each end maintains both a retransmission and a receive list. *I* - frames in the reverse direction carry ACK/NACKs for the forward direction and vice-versa, thus avoiding where possible separate ACK/NACK control frames.

2.1.2.1 - A Comparision of ARQ Schemes

The Send & Wait scheme is very simple but inefficient due to time delay.

The main drawback of the G-B-*N* ARQ scheme is that whenever a received block is found to contain errors, the next (*N*-1) blocks which arrive at R are rejected, regardless of whether they are error-free. Hence, these (*N*-1) blocks must also be retransmitted, reducing the efficiency of the overall system.

As the ordering of the blocks is not maintained in the Selective Repeat ARQ scheme, adequate buffer storage (infinite) must be available so that the received blocks may be reassembled in consecutive order. If the buffering is insufficient, an overflow could occur, resulting in the loss of some of the blocks.

2.1.3 - FEC versus ARQ

The occurrence of a decoding error during correction is much more likely than the occurrence of an undetected error. Hence, since ARQ schemes perform only error-detection, they are far more reliable than FEC schemes.

Error-detecting codes are not very sensitive to error patterns and so detect most error patterns. Hence, ARQ, unlike FEC, is effective in most channels. Further, error-detection with retransmission is adaptive, i.e. transmission of redundant information is increased when errors occur. Therefore in some circumstances we get better performance with ARQ sytems, than would be theoretically possible using FEC.

Further, ARQ schemes are a great deal cheaper than FEC schemes since error-detection is, by its nature, simpler to perform than error-correction.

So, ARQ schemes offer high system reliability fairly independently of the channel conditions. However as the channel conditions decline, more retransmissions are required and so the throughput is reduced.

On the other hand, FEC provides constant throughput regardless of the channel conditions but system reliability falls rapidly as channel degrades.

We have seen that ARQ schemes increase the reliability of the system at the expense of reduced throughput efficiency. When the channel error rate is too high to guarantee the desired throughput using ARQ and where the required system reliability is too high to be met by FEC, it is possible to combine the best attributes of these two schemes to give *hybrid ARQ*.

2.2 - Hybrid ARQ Schemes

Combining ARQ and FEC, *hybrid ARQ* schemes are able to offer higher throughput than could be achieved by ARQ and greater reliability than with FEC alone. Such schemes are based on ARQ with an underlying subsystem using FEC. The function of FEC is to reduce the frequency of retransmissions by correcting errors whenever possible. Hybrid ARQ schemes can de divided into two classes [5]:

(1) *Type-I hybrid ARQ* schemes.
(2) *Type-II hybrid ARQ* schemes.

We shall deal with each class separately.

2.2.1 - Type-I Hybrid ARQ schemes

This is the simpler of the two schemes where a code is used for simultaneous error-detection and correction.

When a received codeword is found to contain errors, R first attempts to correct the errors. If the number of errors is within the capability of the code, they will be corrected, the decoded message will be delivered to the user and ACK sent to T. However, if the error pattern is found to be uncorrectable, R rejects the received codeword and requests a retransmission.

On receiving the retransmitted word, R again tries to correct any errors. If decoding is still not possible, the word is rejected and a second retransmission is requested. This error-correction and retransmission process continues until the codeword is accepted at R.

The code in this scheme, being for both error-detection and correction requires more parity-check bits than a code used only for error-detection. So the overhead for each transmission is increased. If channel error rate is low, type-I has a lower throughput than its corresponding ARQ scheme (due to the extra parity bits). However, in poorer conditions, the correction (where possible) of erroneous codewords reduces the frequency of retransmissions. Hence, under these conditions, type-I ARQ has a higher throughput than a corresponding ARQ scheme.

2.2.2 - Type-II Hybrid ARQ Schemes

This scheme is based on the idea that the parity check bits for error-correction are sent to R only when they are required. Two linear codes, C0 and C1 are required. C0 is an (n, k) error-detecting code and C1 is a $(2n, n)$ half-rate invertible code (see Definition 1.2(a)).

A message is encoded with a number of parity check bits for error-detection only. When R detects errors, it stores the erroneous codeword in the receiver buffer and requests T to send a block of parity check bits based on the original message and the invertible code. When this block is received, it is used to try to correct the erroneous codeword (which is stored in the buffer). Either, the errors are successfully rectified and the corrected codeword is delivered to the data bank; or the errors are found to be outwith the capabilities of the code and R requests a second retransmission. Depending on the retransmission protocol, this second retransmission will be either the original codeword or again a parity block.

Provided that the code used for error-correction and the retransmission strategy are chosen well, type-II should provide better results than type-I.

2.2.3 - A Type-II Hybrid ARQ Scheme

We shall now detail a particular type-II hybrid ARQ scheme [2].

Two linear codes are used - a high rate (n, k) code C0 for error-detection and a half-rate invertible $(2n, n)$ code C1 for simultaneous error-correction and detection. Using the code C0, T encodes a k-bit message D into an n-bit codeword I. Block I is then transmitted. T also computes the codeword $(I,P(I))$ based on the block I and the code C1. Here $P(I)$ represents the n parity check bits. $P(I)$ is not transmitted but stored in the retransmission buffer for possible later use. Note CI is invertible and so from a parity block the original message may be obtained. i.e. $P(I)$ may be inverted to give $I(P)$ an estimate of I.

Let I^ denote the received codeword corresponding to I. At R, error-detection based on C0 is performed on I^. If the syndrome is zero, I^ is assumed error-free. It will be accepted by R and ACK sent to T. While if the syndrome is non-zero, then I^ contains errors. I^ will be saved in the buffer at R and NACK is sent to T.

On receiving this negative response, T sends $P(I)$ to R. Let $P(I)$^ denote the received codeword corresponding to $P(I)$. On receipt of $P(I)$^, R computes its inverse, $I(P)$^. $I(P)$^ is then tested for errors using C1. If it is found to be error-free, then R assumes that $I(P)$^ = I and ACK is sent to T. Otherwise, the blocks $P(I)$^ and I^ are used together for error-correction based on the $(2n, n)$ code C1. Let I^0 denote the decoded block after this error-correction process. I^0 (a codeword in C0) is now tested for errors using C0. If no errors are detected, it is assumed that $I^0 = I$ and ACK is sent to T. However, if errors are found to be present, R discards I^, stores $P(I)$^ in its place and a second NACK is sent to T.

The second retransmission will be the block I itself. So the retransmissions alternate between the block I and the parity block $P(I)$ until the block is successfully received.

The most important feature of this type-II hybrid ARQ is the parity retransmission based on the half-rate invertible code C1. Although this protocol can be used with any of the three basic types of ARQ, it is particularly effective with selective repeat ARQ. Since C1 is invertible, the message I can be uniquely determined from the parity block $P(I)$ and so I and $P(I)$ contain the same amount of information. If the channel error rate is low this scheme maintains the same throughput as the corresponding ARQ scheme. While, if the channel error rate is high, the error-correction capibility provided by the code C1 and the parity retransmission reduces the frequency of retransmissions and so the throughput remains good.

2.2.4 - A Comparision of Type-I & Type-II Hybrid ARQ Schemes

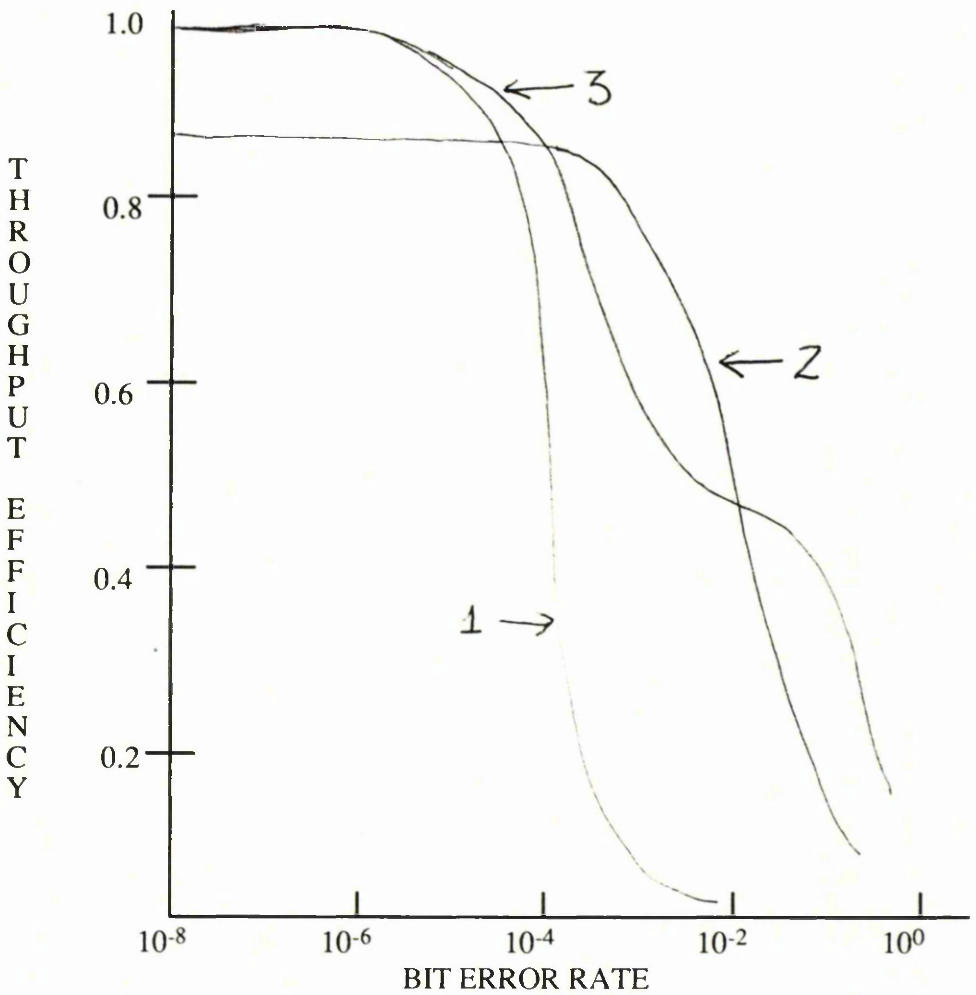
The main disadvantage of type-I is that the overhead due to the extra parity check bits for error-correction is included in each transmission regardless of whether or not it is required. When the channel is quiet this is very wasteful. It is best suited for channels whose characteristics are fairly constant. Type-II removes this drawback since error-detection with retransmission is adaptive. It is particularly attractive for error control over channels where the data rate is high, the round trip delay large and the error rate changeable.

The decoding complexity for type-II is only slightly greater than that for a corresponding type-I scheme designed to have the same error-correcting capability. The extra circuits needed in type-II are an inversion circuit based on C1 and an error-detection circuit based on C0.

In Figure 2.1, typical plots of throughput efficiency versus channel error rate for selective-repeat, type-I and type-II hybrid selective-repeat ARQ schemes are presented for comparision [5].

For bit error rate of up to about 10^{-5}, the throughput of both S-R ARQ and type-II hybrid ARQ are seen to be constant and close to one. Hence over channels, with these conditions, both these schemes perform well. Type-I hybrid ARQ remains constant, although less than one, for channel error rate up to 10^{-4}.

When the channel error rate is greater than 10^{-5}, the throughput efficiency of SR-ARQ declines to almost zero. Simarily, if the error rate is more than 10^{-4}, the throughput of type-I hybrid ARQ decrease rapidly. However, the throughput of type-II hybrid ARQ does not fall at such a rate under degrading conditions. In fact, the throughput of this scheme, has an inflection at 0.5 - since error-correction is performed upon the first retransmission, the probability of further retransmissions is reduced.



KEY
1. Selective Repeat ARQ
2. Type-I Hybrid Selective Repeat ARQ
3. Type-II Hybrid Selective Repeat ARQ

Figure 2.1 - Throughput efficiency of various ARQ schemes.

2.3 - GH-ARQ - Generalised Type-II Hybrid ARQ Schemes

In type-II hybrid ARQ schemes, the second retransmission is the same as the original codeword. Clearly the performance of the system could be improved if the second retransmission is another parity block which can be used to form a $(3n, n)$ error-correcting code. This scheme can be generalised to any number of retransmissions before T resends the blocks in repetition again. This is a *Generalised Type-II Hybrid ARQ* scheme (GH-ARQ) [2], [3].

GH-ARQ uses two codes - namely a high rate (n, k) code C0 for error-detection only and a (mn, n) code C1 which is used adaptively for error-correction. Code C1 is an (mn, n) error-correcting code having distance d and generator matrix G. G can be partitioned into m subblocks $G_1, \ldots, G_m$, each of dimension $(n \times n)$. m is referred to as the depth of the code.
Then

$$G = [G_1 \mid G_2 \mid \ldots\ldots \mid G_m].$$

For code C1 to perform as required, it is assumed that the subcode $C1^{(i)}$ with generator matrix $G^{(i)}$ where

$$G^{(i)} = [G_1 \mid G_2 \mid \ldots\ldots \mid G_i]$$

has minimum distance d_i such that $d_i < d_j$ for all $1 \le i < j \le m$. By definition, the depth of the subcode $C1^{(i)}$ is i and obviously $C1^{(m)}$ is just C1.

Let I denote the block obtained from the message D using the (n, k) code C0. A mn - bit codeword, $\boldsymbol{c}$, is formed using I and the (mn,n) code C1.
i.e.

$$\boldsymbol{c} = (\boldsymbol{c}_1,\ \boldsymbol{c}_2,\ \ldots\ldots,\ \boldsymbol{c}_{\mathrm{m}}) = IG \quad \text{and} \quad \text{so} \quad c_i = IG_i$$

The data block I can be uniquely determined from knowledge of $\boldsymbol{c}_i$ if and only if the corresponding $(n \times n)$ matrix G_i is invertible.
i.e.

$$I = c_i G^{-1}$$

Hence the matrix G_1 is assumed invertible so the data block can be recovered from $\boldsymbol{c}_1$ alone (first transmission). It is desirable although not essential that G_i, $i = 2, \ldots, m$ also be invertible. This is particularly important when a burst of errors may destroy one of the transmissions, yet leave the others relatively error-free.

For I to be transmitted, T sends the following sequence of blocks until a block is accepted by R : $\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_m, \boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_m, \boldsymbol{c}_1, \ldots$

Let R_i denote the received block corresponding to the transmitted block c_i. When R receives a block R_i , R will either :

(i) compute $E_i = R_i G_i^{-1}$, an estimate of I, provided G_i is invertible. Then, using C0, perform error-detection on E_i. If E_i is found to be error-free then it is assumed that $E_i = I$ and ACK is sent to T. On the otherhand, if E_i contains errors, then decode $[R_1 \ldots R_i]$ using the subcode $C1^{(i)}$. This provides I^0, an estimate of I. Using C0, I^0, is tested for errors. If I^0, is found to contain errors, NACK is sent to T and $R_1, \ldots, R_i$ are stored in the R buffer. Otherwise, it is assumed that $I^0 = I$ and ACK is sent to T.
or

(ii) If G_i^{-1} does not exist, then decode $[R_1 \ldots R_i]$, using the subcode $C1^{(i)}$ to obtain I^0, an estimate of I. Error-detection based on C0 is then performed on I^0. If no errors are found in I^0, ACK is sent to T while if I^0 contains errors $R_1, \ldots, R_i$ are stored in the R buffer and NACK sent to T.

For the above sequence of blocks transmitted, R performs error-correction based on the codes $C1^{(2)}, C1^{(3)}, C1^{(4)}, \ldots, C1^{(m-1)}, C1, C1, \ldots$ having minimum distance $d_2, d_3, d_4, \ldots, d_{m-1}, d, d, \ldots$. So, with each retransmission, a code with a larger distance, and hence a greater error-correcting capability is used for error correction until the code C1 is obtained.

Obviously, the type-II hybrid ARQ scheme, described in Section 2.2.3, is a special case of GH-ARQ where $m = 2$.

For a block I, T computes a codeword of length mn based on the code C1 and so a buffer of size mn is needed at both R and T for each block transmitted. T also needs an encoder for C0 and C1 and R requires decoders for each of the codes $C1^{(2)}, C1^{(3)}, \ldots, C1^{(m-1)}$, C1and an invertor circuit for each $\boldsymbol{c}_i$, $i = 1, \ldots, m$.

This scheme may seem rather complex and expensive to impliment. If the decoding process for each code $C1^{(2)}, \ldots, C1^{(m-1)}$, C1 were different, this could well be the case and in fact could offset any gain in system performance. Alternatively, if the decoding process for these codes were the same, then the GH-ARQ scheme could be a great deal better than any type-II hybrid ARQ schemes without overwhelming additional system complexity. There follows (Chapters 4-6) a study of a class of codes having these necessary properties.

CHAPTER 3 - Bilinear Forms & Linear Codes

In this chapter, the correspondence between bilinear forms and linear codes is presented. Firstly, the mathematics required to illustrate this relationship is given.

3.1 - Mathematical Background

Definition 3.1

A function $B(\boldsymbol{u}, \boldsymbol{v})$ of two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ in a given field F^n is called *bilinear* if the following conditions hold :

(i) $B(\boldsymbol{u}_1 + \boldsymbol{u}_2, \boldsymbol{v}) = B(\boldsymbol{u}_1, \boldsymbol{v}) + B(\boldsymbol{u}_2, \boldsymbol{v})$,
(ii) $B(r\boldsymbol{u}, \boldsymbol{v}) = rB(\boldsymbol{u}, \boldsymbol{v})$,
(iii) $B(\boldsymbol{u}, \boldsymbol{v}_1 + \boldsymbol{v}_2) = B(\boldsymbol{u}, \boldsymbol{v}_1) + B(\boldsymbol{u}, \boldsymbol{v}_2)$
and
(iv) $B(\boldsymbol{u}, r\boldsymbol{v}) = rB(\boldsymbol{u}, \boldsymbol{v})$

$\forall\ \boldsymbol{u}, \boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{v}, \boldsymbol{v}_1, \boldsymbol{v}_2 \in F^n$ and $r \in F$.

Definition 3.2

Let F be a given field and $x_1, x_2, \ldots, x_r$ be indeterminates over F. The *extension* of F, denoted $F[x_1, x_2, \ldots, x_r]$ is the smallest commutative ring R such that $F \cup \{x_1, x_2, \ldots, x_r\} \subseteq R$. To determine the multiplicative complexity of an algebraic function over F, we consider the function as an element of $F[x_1,x_2, \ldots, x_r]$. F is referred to as the field of constants and multiplications by fixed elements of F are not counted. The *multiplicative complexity* of an element $a \in F[x_1,x_2,\ldots, x_r]$ is defined to be the minimum number of multiplications involving a pair of elements from $F[x_1,x_2,\ldots, x_r]$ that are needed to compute a.

For example, if $a = x_3x_1 + x_3x_2 + x_3x_3$ then the multiplicative complexity of a is one since $a = x_3(x_1 + x_2 + x_3)$.

Let $M(B)$ denote the multiplicative complexity of a bilinear form B.

Theorem 3.1 - Chinese Remainder Theorem for Polynomials

Consider the following system of congruences,

$$Y(u) \equiv Y_1(u) \text{ modulo } P_1(u)$$
$$Y(u) \equiv Y_2(u) \text{ modulo } P_2(u)$$
$$\vdots$$
$$Y(u) \equiv Y_t(u) \text{ modulo } P_t(u)$$

where $P_1(u), P_2(u), \ldots, P_t(u)$ are relatively prime polynomials.

The *Chinese Remainder Theorem for polynomials* states that $Y(u)$ can be uniquely reconsructed from these congruences modulo P(u), where $P(u) = \prod_{i=1}^{t} P_i(u)$.

The method employed to solve such a system is described below:

$$Y(u) \equiv \sum_{i=1}^{t} S_i(u) Y_i(u) \text{ modulo } P(u),$$

where the polynomials $S_i(u)$ satisfy the following congruences,

$$S_i(u) \equiv \begin{cases} 0 & \text{modulo } P_j(u) \quad j = 1,2,..,t\, ; j \neq i \\ 1 & \text{modulo } P_i(u) \end{cases}$$

and the polynomials $S_i(u)$ are given by

$$S_i(u) = R_i(u) \prod_{\substack{j=1 \\ j \neq i}}^{t} P_j(u)$$

where the polynomials $R_i(u)$ are determined from the congruence,

$$R_i(u) \prod_{\substack{j=1 \\ j \neq i}}^{t} P_j(u) \equiv 1 \text{ modulo } P_i(u).$$

#

Definition 3.3

Let ϑ denote a set of k bilinear forms.

i.e.

$$\vartheta = \begin{bmatrix} \vartheta_1 \\ \vartheta_2 \\ . \\ . \\ \vartheta_k \end{bmatrix} = \begin{bmatrix} X_{1,1}y_1 + X_{1,2}y_2 + \, .. \, + X_{1,s}\, y_s \\ X_{2,1}y_1 + X_{2,2}y_2 + \, .. \, + X_{2,s}\, y_s \\ . \\ . \\ X_{k,1}y_1 + X_{k,2}y_2 + \, .. \, + X_{k,s}\, y_s \end{bmatrix}$$

where $X_{i,j}$ is of the form $\sum_{m=1}^{r} a_{m_{i,j}} x_m$, $i = 1, 2, ..,k, j = 1, 2, ..,s$.

Then, we may express ϑ as

$$\vartheta = \begin{bmatrix} X_{1,1} & X_{1,2} & . & . & . & X_{1,s} \\ X_{2,1} & X_{2,2} & . & . & . & X_{2,s} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ X_{k,1} & X_{k,2} & . & . & . & X_{k,s} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_s \end{bmatrix}$$

$$= Xy$$

Formally, a *computation* of ϑ is defined to be an expression of the form

$$\vartheta = C(Ax \text{ x } By)$$

where A, B and C are matrices of dimension (n x r), (n x s) and (k x n) respectively over F, $x = (x_1, x_2, \ldots, x_r)^T$ and x represents component-by-component multiplication of vectors [6].

Informally, a computation is simply deciding on how the terms of the bilinear form shall be grouped and on the order of the operations required to compute the bilinear form.

A computation of the form above is said to be *non-commutative* (NC),[7], since it involves component-by-component multiplication between elements of the form

$$\sum_{i=1}^{r} a_i \cdot x_i \quad \text{and} \quad \sum_{j=1}^{s} b_j \, y_j \, .$$

Since the straightforward method of evaluating $C(Ax \text{ x } By)$ involves n multiplications between elements of the form above, the computation is said to have multiplicative complexity n [6].

It should be noted that matrices A, B and C are not unique, however they are related. Let $D(x)$ denote the (n x n) diagonal matrix whose diagonal elements are the elements of the column vector Ax . Then, any computation $C(Ax \text{ x } By)$ of ϑ can be written as

$$\vartheta = C(Ax \text{ x } By) = CD(x)By.$$

i.e.

$$Xy = CD(x)By$$

and since this holds for every y, it follows that

$$X = CD(x)B. \tag{3.1}$$

Conversely, given a decomposition of X into $C'D'(x)B'$ where C' and B' are over F and $D'(x)$ consists of linear forms of x on its main diagonal and zeros everywhere else, then we have an algorithm, $C'(A'x \text{ x } B'y)$ for computing Xy where $(A'x \text{ x } B'y) = D'(x)B'y$. We may therefore conclude that there exists a one-to-one correspondence between NC algorithms $C(Ax \times By)$ for computing ϑ and the decomposition of X described above(3.1).

A decomposition of X of the form (3.1) is said to be *minimal* if n is the least integer for which such a decomposition exists. Clearly, every minimal decomposition defines an algorithm for computing ϑ with minimum multiplicative complexity and vice versa [7].

Example 3.1.1

Consider ϑ, a system of 3 bilinear forms given by :

$$\vartheta = \begin{bmatrix} x_1y_1 + x_2y_2 + x_1y_2 + x_2y_3 \\ x_2y_2 + x_3y_1 + x_2y_3 + x_3y_3 \\ x_3y_1 + x_3y_3 + x_2y_3 \end{bmatrix}.$$

We may express ϑ in the form $\vartheta = Xy$.

i.e.

$$\vartheta = \begin{bmatrix} x_1 & x_1+x_2 & x_2 \\ x_3 & x_2 & x_2+x_3 \\ x_3 & 0 & x_2+x_3 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

Rearranging the terms, we see

$$\vartheta = \begin{bmatrix} x_2(y_2+y_3) + x_1(y_1+y_2) \\ (x_2+x_3)y_3 + x_2y_2 + x_3y_1 \\ x_3(y_1+y_3) + x_2y_3 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_2(y_2+y_3) \\ x_1(y_1+y_2) \\ (x_2+x_3)y_3 \\ x_2y_2 \\ x_3y_1 \\ x_3(y_1+y_3) \\ x_2y_3 \end{bmatrix}.$$

From this, we obtain the following computation of ϑ,

$$\vartheta = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \left[\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \right].$$

This computation of ϑ has 7 multiplications. It was stated above that a computation of ϑ is not unique. There are many other computations, including the one below, which is minimal.

$$\vartheta = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \left[\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \right]$$

Definition 3.4

Let $z = (z_1, z_2, \ldots\ldots, z_k)^T$ and let $C(Ax \times By)$ be a computation of ϑ, a system of k bilinear forms. The *P-dual* of the computation is the algorithm $A^T(C^Tz \times By)$ and the *R-dual* is $B^T(Ax \times C^Tz)$ [2].

Example 3.1.2

The *P-dual* of the minimal computation of ϑ given in Example 3.1.1 is

$$\Phi = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \left[\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \right]$$

while the *R-dual* is

$$\Psi = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \left[\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} \right].$$

Theorem 3.2

The following statements are equivalent :-
(i) there is a computation having n multiplications for the system $C(Ax \times By)$,
(ii) there is a computation having n multiplications for the system $A^T(C^Tz \times By)$,
(iii) there is a computation having n multiplications for the system $B^T(Ax \times C^Tz)$,
(iv) there is a computation having n multiplications for the system $B^T(C^Tz \times Ax)$,

(v) there is a computation having n multiplications for the system $A^T(By \times C^Tz)$ and
(vi) there is a computation having n multiplications for the system $C(By \times Ax)$ [6].

Proof

Suppose $x = (x_1,x_2,..,x_r)$, $y = (y_1,y_2,..,y_s)$ and $z = (z_1,z_2,..,z_k)$.

(i) $\Rightarrow$ (ii) Suppose (i) holds, then matrices A, B and C are of dimension $(n \times r)$, $(n \times s)$ and $(k \times n)$ respectively. Thus C^Tz and By are column vectors of length n and $(C^Tz \times By)$ involves n component-by-component multiplications. It follows that $A^T(C^Tz \times By)$ has n multiplications and so condition(ii) is true.

(ii) $\Rightarrow$ (i) Since system (ii) can be computed using n component-by-component multiplications, the matrices A, B and C are of dimension $(n \times r)$, $(n \times s)$ and $(k \times n)$ respectively. Further, $(Ax \times By)$ involves n component-by-component multiplications, so that the computation $C(Ax \times By)$ has n multiplications. Hence result.

A similar argument can be applied repeatedly to show the equivalence of all six statements.

#

The following theorem establishes either a row or a column orientated lower bound on the number of multiplications necessary to compute a system ϑ of k bilinear forms, while Theorem 3.4 gives a lower bound which is both row and column orientated.

Theorem 3.3

Let Xy be a system of bilinear forms over F where X is a $(t \times n)$ matrix with entries $x_{i,j} \in F[x]$ and $y = (y_1, .., y_n)^T$. If the column rank of X is α, then any algorithm computing Xy requires at least α multiplications [8].

Dually, if the row rank of X is β, then any computation of the system requires at least β multiplications [2].

In order to prove Theorem3.3, we require the following definition.

Definition 3.5

Let $\mathbf{A} = (L, \Omega)$ where L is called the carrier of the algebra and Ω denotes the set of (possibly partial) operations. Let L^i be the cartesian product of i copies of L.

An i - *ary operation* on L is a map

$$\omega : L^i \rightarrow L$$

while a *partial i - ary operation* on L is a map

$$\omega : Y_i \rightarrow L$$

where Y_i is a non-empty subset of L^i.

In constructing an element $\alpha \in \mathbf{A}$, we start with $\mathbf{B}$, a certain subset of $\mathbf{A}$ and construct other elements of $\mathbf{A}$ from elements in $\mathbf{B}$ using the operations in Ω.

Then, an *N-step algorithm* Γ over $(\mathbf{A}, \mathbf{B})$ is a mapping

$$\Gamma : \{1,2,..,N\} \longrightarrow \mathbf{B} \cup \left(\bigcup_{i \in I} \{\omega_i\} \times \{1,2,..,N\}^{n_i} \right)$$

subject to the constraint that if $\Gamma(t)=(\omega_i, j_1, .., j_{ni})$ then $j_s < t$ for $s = 1,..,n_i$.

Associated with each algorithm Γ, there is a (partial) function $e_\Gamma : \{1,2,..,N\} \rightarrow \mathbf{A}$ given by

(a) $e_\Gamma(t) = \Gamma(t)$ if $\Gamma(t) \in \mathbf{B}$

(b) $e_\Gamma(t) = \omega_i(e_\Gamma(j_1), .., e_\Gamma(j_{ni}))$ if $\Gamma(t)=(\omega_i, j_1, .., j_{ni})$ and $e_\Gamma(j_s)$, $s = 1,..,n_i$ and $\omega_i(e_\Gamma(j_1),..,e_\Gamma(j_{ni}))$ are defined.

Further, the function e_Γ is *total* provided $e_\Gamma(i)$ is defined, $i = 1, 2,.., N$.

So, the function $\Gamma(i)$ is the sequence of operations which the algorithm executes while $e_\Gamma(i)$ is the sequnce of elements which Γ computes.

Example 3.1.3

Let $F = GF(2)$, $\mathbf{A} = F(x_1, x_2, x_3)$ and $\mathbf{B} = F \cup \{x_1, x_2, x_3\}$. Further let Γ be the following 8-step algorithm over $(\mathbf{A}, \mathbf{B})$ where ω_1 and ω_2 denote multiplication and addition respectively : $\Gamma(1) = x_1$, $\Gamma(2) = x_2$, $\Gamma(3) = x_3$, $\Gamma(4) = (\omega_1, 2, 3)$, $\Gamma(5) = (\omega_1, 1, 3)$, $\Gamma(6) = (\omega_1, 1, 1)$, $\Gamma(7) = (\omega_2, 5, 6)$ and $\Gamma(8) = (\omega_2, 4, 7)$.

It is easily verified that e_Γ is total and

$$e_\Gamma(8) = x_2x_3 + x_1x_3 + x_1x_1.$$

Proof of Theorem 3.3

To prove Theorem 3.3, we shall first consider the number of multiplications necessary to compute $Xy + \rho$ where matrix X and vector y are as stated in Theorem 3.3 while $\rho = (\rho_1, .., \rho_t)^T$; $\rho_i \in F$, $i = 1, 2, ..,t$. Within this proof, it will be assumed that all algorithms are over $(\mathbf{A}, \mathbf{B})$ where $\mathbf{A} = F(y_1, .., y_n)$ and $\mathbf{B} = F \cup \{y_1, .., y_n\}$.

Let Γ be an algorithm computing $Xy + \rho$. Recall, multiplications by fixed elements of F do not contribute to the multiplicative complexity of the algorithm. So if Γ has a step t, such that $\Gamma(t)=(*, j_1, j_2)$ where $e_\Gamma(j_1) \notin F$ and $e_\Gamma(j_2) \notin F$, then this 'multiplication is counted'. Otherwise the step t does not involve a multiplication which is counted.

Suppose that the first k steps of Γ do not involve a multiplication which is counted. Then, clearly step i, $i = 1, 2, .., k$ does not involve any multiplications between any y_i's. Hence, $e_\Gamma(i)$, $i = 1, 2, .., k$ must be of the form

$$e_\Gamma(i) = \sum_{j=1}^{n} f_{j,i}\, y_j + g_i \qquad (3.2)$$

for some $f_{i,j} \in F$, $i = 1, 2, \ldots, k$, $j = 1, 2, \ldots, n$ and some $g_i \in F$. Infact, these elements form a vector space over F.

To prove that if X has column rank α, then any algorithm Γ computing $Xy + \rho$, has at least α multiplications which are counted, we shall use induction on α.

$\alpha = 1$

Suppose Γ involves no multiplications which are counted.

Since the column rank of X is one, there must exist i^*, j^* $(1 \le i^* \le t, 1 \le j^* \le n)$ such that $x_{i^*,j^*} \notin F$. Now, Γ computes $Xy + \rho$ and so some step s must be of the form

$$e_\Gamma(s) = \sum_{j=1}^{n} x_{i^*,j}\, y_j + r_{i^*} \qquad (3.3)$$

for some $r_{i^*} \in F$.

The first s steps of Γ do not involve a multiplication which is counted and so it follows from (3.2) that there exist $f_j \in F$, $j = 1, 2, \ldots, n$ and $g \in F$ such that

$$e_\Gamma(s) = \sum_{j=1}^{n} f_j\, y_j + g\,. \qquad (3.4)$$

Comparing the expressions (3.3) and (3.4), we see that $x_{i^*,j^*} = f_{j^*}$ - contradicting $x_{i^*,j^*} \notin F$. Hence Γ must have at least one multiplication which is counted.

We shall assume that our assertion is true for α. Let Γ be an algorithmn minimising the multiplicative complexity of the system $Xy + \rho$ and let the column rank of X be $\alpha+1$.

Let k be the smallest integer such that a multiplication that is counted occurs at step k. Then $\Gamma(k)=(*, j_1, j_2)$ where $e_\Gamma(j_1) \notin F$ and $e_\Gamma(j_2) \notin F$. However, since no steps before step k have multiplications which are counted, we may express $e_\Gamma(j_1)$ and $e_\Gamma(j_2)$ in the form (3.2). Then,

$$e_\Gamma(k) = e_\Gamma(j_1) * e_\Gamma(j_2)$$

$$= \left(\sum_{i=1}^{n} f_i\, y_i + g\right) * \left(\sum_{i=1}^{n} f_i'\, y_i + g'\right)$$

for some $f_i, f_i' \in F$, $i = 1, 2, \ldots, n$ and some $g, g' \in F$.

Observe,(at the very least) either one of the f_i or one of the f_i' must be non-zero. For otherwise, $e_\Gamma(j_1) = g \in F$ and $e_\Gamma(j_2) = g' \in F$ and the multiplication at step k would not be counted. Without loss of generality, we assume that one of the f_i' is non-zero. Infact, we may assume that $f_n' \neq 0$. Moreover, since multiplications by f_n' are not counted, we may take $f_n' = 1$. Let $h \in F$ be such that if we substitute y_n with $h - g' - \sum_{i=1}^{n-1} f_i' y_i$,

the resulting algorithm Γ' is such that $e_{\Gamma'}$ is total. Consider this substitution, we have

$$\begin{bmatrix} x_{11} & x_{12} & \cdot & \cdot & x_{1n} \\ x_{21} & x_{22} & \cdot & \cdot & x_{2n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ x_{t1} & x_{t2} & \cdot & \cdot & x_{tn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ y_{n-1} \\ h - g' - \sum_{i=1}^{n-1} f_i' y_i \end{bmatrix} + \begin{bmatrix} \rho_1 \\ \rho_2 \\ \cdot \\ \cdot \\ \rho_t \end{bmatrix}$$

$$= \begin{bmatrix} x_{11}y_1 + x_{12}y_2 + \ldots + x_{1n-1}y_{n-1} + x_{1n}(h - g' - \sum_{i=1}^{n-1} f_i' y_i) \\ x_{21}y_1 + x_{22}y_2 + \ldots + x_{2n-1}y_{n-1} + x_{2n}(h - g' - \sum_{i=1}^{n-1} f_i' y_i) \\ \cdot \\ \cdot \\ x_{t1}y_1 + x_{t2}y_2 + \ldots + x_{tn-1}y_{n-1} + x_{tn}(h - g' - \sum_{i=1}^{n-1} f_i' y_i) \end{bmatrix} + \begin{bmatrix} \rho_1 \\ \rho_2 \\ \cdot \\ \cdot \\ \rho_t \end{bmatrix}$$

$$= \begin{bmatrix} (x_{11} - f_1'x_{1n})y_1 + (x_{12} - f_2'x_{1n})y_2 + \ldots + (x_{1n-1} - f_{n-1}'x_{1n})y_{n-1} \\ (x_{21} - f_1'x_{2n})y_1 + (x_{22} - f_2'x_{2n})y_2 + \ldots + (x_{2n-1} - f_{n-1}'x_{2n})y_{n-1} \\ \cdot \\ \cdot \\ (x_{t1} - f_1'x_{tn})y_1 + (x_{t2} - f_2'x_{tn})y_2 + \ldots + (x_{tn-1} - f_{n-1}'x_{tn})y_{n-1} \end{bmatrix}$$

$$+ \begin{bmatrix} \rho_1 + x_{1n}(h - g') \\ \rho_2 + x_{2n}(h - g') \\ \cdot \\ \cdot \\ \rho_t + x_{tn}(h - g') \end{bmatrix}$$

$$= \begin{bmatrix} x_{11} - f_1'x_{1n} & x_{12} - f_2'x_{1n} & \cdot & \cdot & x_{1n-1} - f_{n-1}'x_{1n} \\ x_{21} - f_1'x_{2n} & x_{22} - f_2'x_{2n} & \cdot & \cdot & x_{2n-1} - f_{n-1}'x_{2n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ x_{t1} - f_1'x_{tn} & x_{t2} - f_2'x_{tn} & \cdot & \cdot & x_{tn-1} - f_{n-1}'x_{tn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ y_{n-1} \end{bmatrix} + \begin{bmatrix} \rho_1 + x_{1n}(h - g') \\ \rho_2 + x_{2n}(h - g') \\ \cdot \\ \cdot \\ \rho_t + x_{tn}(h - g') \end{bmatrix} .$$

Let $X_1, X_2, \ldots, X_n$ denote the columns of matrix X. Then clearly after the above substitution, we have an algorithmn Γ' which computes $X'y' + \rho'$, where X' is a $t \times (n-1)$ matrix with columns X_j' given by $X_j' = X_j - f_j'X_n$, $j = 1, 2, \ldots, n-1$, $y' = (y_1, y_2, \ldots, y_{n-1})^T$ and $\rho' = \rho + (h - g')X_n$. Step k of Γ' is not a multiplication which is counted. Further, Ψ, the algorithm computing $h - g' - \sum_{i=1}^{n-1} f_i'y_i$ has no multiplications which are counted. Hence, the multiplicative complexity of Γ' is at least one less than that of Γ. But the column rank of matrix X' is at least α and so by our induction hypothesis Γ' has at least α multiplications which are counted. It follows that Γ involves at least $(\alpha+1)$ multiplications which are counted. Thus, we have proved the result for computing the system $Xy + \rho$.

To prove theorem 3.3 - simply take $\rho = 0$.

#

Theorem 3.4

Let Xy be a system of bilinear forms over a field F, where F contains the entries of X as indeterminates. If X has an $(\alpha \times \beta)$ submatrix S such that

$$u^TSv \in F \quad \text{iff} \quad u = 0 \text{ or } v = 0.$$
$$\forall\, u \in F^{\alpha} \text{ and } v \in F^{\beta}$$

Then, the multiplicative complexity of Xy is at least $\alpha + \beta - 1$ [9].

Proof

Suppose X is a $(m \times n)$ matrix and $y = (y_1, y_2, \ldots, y_n)^T$. Further, suppose that Xy may be computed in t multiplications.

Let X' be the $(\alpha \times n)$ submatrix of X, which contains the $(\alpha \times \beta)$ submatrix S. Then, since Xy can be computed with t multiplications, there exist matrices A, B and c of dimension $(\alpha \times t)$, $(\alpha \times n)$ and $(\alpha \times 1)$ respectively, with entries from F, such that

$$X'y = A\mu + By + c\,.$$

Recall,($\forall\ 0 \neq u \in F^{\alpha}$ and $0 \neq v \in F^{\beta}$), $u^TSv \notin F$ and so the rows of X' must be linearly independent over F. Then, by Theorem 3.3, $\alpha \geq t$.

Now partition matrix A into 2 matrices A_1 and A_2 such that A_2 has α rows and $\alpha-1$ columns. Then, there exists a, a $(1 \times \alpha)$ vector over F such that

$$aA_2 = 0.$$

Then, writing $\mu = (\mu_1, \mu_2)^T$, where μ_1 corresponds to the first $t - \alpha+1$ elements of μ, we have

$$aX'y = a\{[A_1 \mid A_2](\mu_1, \mu_2)^T + By + c\}$$

$$= \boldsymbol{a}A_1\mu_1 + \boldsymbol{a}B\boldsymbol{y} + \boldsymbol{a}\boldsymbol{c}$$
$$= \boldsymbol{a}'\mu_1 + \boldsymbol{b}\boldsymbol{y} + \boldsymbol{d}.$$

So, $\boldsymbol{a}X'$ is a non-trivial linear combination of the rows of X' such that $\boldsymbol{a}'X'\boldsymbol{y}$ can be computed in t-α+1 multiplications. However, $\boldsymbol{u}^TS\boldsymbol{v} \notin F$ and thus $\boldsymbol{a}X'$ must have at least β linearly independent columns over F. Then, by Theorem 3.3, $\boldsymbol{a}X'\boldsymbol{y}$ has a multiplicative complexity of no less than β.
i.e.

$$t - \alpha + 1 \geq \beta$$
$$t \geq \beta + \alpha - 1$$

Hence result.

#

3.2 - The Multiplicative Complexity of Bilinear Forms & Linear Codes

The following theorem establishes a correspondence between linear (n, k, d) codes and algorithms for computing a system ϑ of k bilinear forms [2].

Theorem 3.5

Consider a system $\vartheta = X\boldsymbol{y}$ of k linearly independent bilinear forms. The $(k \text{ x } n)$ matrix C in a computation $\vartheta = C(\mathrm{A}\boldsymbol{x} \text{ x } B\boldsymbol{y})$ is the generator matrix of a linear (n, k, d') code over F where

$$d' \geq d = \min \{\rho(\boldsymbol{u}^TX)\}$$
$$\forall\, \boldsymbol{u} \in F^k,\ \boldsymbol{u} \neq 0$$

and d = design distance and d'= actual minimum distance.

Proof

Recall, $M(\vartheta)$ denotes the multiplicative complexity of the system ϑ. We observed previously that for any integer $n \geq M(\vartheta)$, ϑ may be computed as $\vartheta = C(A\boldsymbol{x} \text{ x } B\boldsymbol{y})$ where matrices A, B and C are of dimension $(n \text{ x } r)$, $(n \text{ x } s)$, $(k \text{ x } n)$ respectively.

The row rank of X, $\rho(X)$, is k, then by Theorem 3.3, any computation of ϑ will require at least k multiplications. i.e. $n \geq M(\vartheta) \geq k$. Hence $n \geq k$.
Further, $(A\boldsymbol{x} \text{ x } B\boldsymbol{y})$ involves n component-by-component multiplications. So the above computation of ϑ has n muliplications provided all k rows of C are linearly independent. If C had any dependent rows, this computation will require fewer multiplications. It follows that $\rho(C)$, the rank of C must be k.

So we have shown that C is a $(k \text{ x } n)$ matrix where $n \geq k$ and all k rows of C are linearly independent. Thus, matrix C can be considered the generator matrix of a linear (n, k) code over F. A typical codeword is $\boldsymbol{c}^T = (c_1, c_2, \ldots\ldots, c_n)$ where

$$c^T = u^T C \quad \text{and} \quad u^T = (u_1, u_2, \ldots, u_k).$$

We have

$$\vartheta = Xy = C(Ax \times By)$$

which implies

$$u^T Xy = u^T C(Ax \times By). \tag{3.5}$$

i.e.

$$u^T Xy = c^T(Ax \times By).$$

Consider any entry c_i of a non-zero codeword c ($i = 1, 2, .., n$). Observe if

(i) $c_i \neq 0$ then ith component-by-component multiplication of $(Ax \times By)$ is necessary.

while if

(ii) $c_i = 0$ then the ith multiplication need not be done as it will disappear when multiplied by c_i.

It follows that the weight of any non-zero codeword c (i.e. the number of non-zero entries of c) cannot be less than the multiplicative complexity of $c^T(Ax \times By)$.
i.e.

$$w(c) \geq M(c^T(Ax \times By))$$

Further, by (3.5) the multiplicative complexity of $c^T(Ax \times By)$ is equal to the multiplicative complexity of $u^T Xy$.
i.e.

$$M(c^T(Ax \times By)) = M(u^T Xy)$$

Finally, by Theorem 3.3, the number of multiplications necessary to compute $u^T Xy$ is greater than or equal to the row rank of $u^T X$.
i.e.

$$M(u^T Xy) \geq \rho(u^T X)$$

Hence, we have

$$w(c) \geq M(c^T(Ax \times By)) = M(u^T Xy) \geq \rho(u^T X).$$

Hence result. #

3.3 - A Particular Bilinear Form

We shall now study ϑ, a particular system of k bilinear forms given below :

$$\vartheta = X\,\mathbf{y} = \begin{bmatrix} x_0 & x_1 & x_2 & \cdot & \cdot & \cdot & x_{d-2} & x_{d-1} \\ x_1 & x_2 & x_3 & \cdot & \cdot & \cdot & x_{d-1} & x_d \\ x_2 & x_3 & x_4 & \cdot & \cdot & \cdot & x_d & x_{d+1} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ x_{k-1} & x_k & x_{k+1} & \cdot & \cdot & \cdot & x_{k+d-3} & x_{k+d-2} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \cdot \\ \cdot \\ y_{d-1} \end{bmatrix}. \tag{3.6}$$

Consider $\boldsymbol{a}^T X \boldsymbol{b}$ for any non-zero $\boldsymbol{a} \in F^k$ and non-zero $\boldsymbol{b} \in F^d$.

$$\mathbf{a}^T X\ \mathbf{b} = \begin{bmatrix} a_0 & a_1 & \cdot & \cdot & a_{k-1} \end{bmatrix} \begin{bmatrix} x_0 & x_1 & \cdot & \cdot & \cdot & x_{d-1} \\ x_1 & x_2 & \cdot & \cdot & \cdot & x_d \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ x_{k-1} & x_k & \cdot & \cdot & \cdot & x_{k+d-2} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \cdot \\ \cdot \\ b_{d-1} \end{bmatrix}$$

$$= \sum_{j=0}^{d-1} \left(\sum_{i=0}^{k-1} a_i\, x_{i+j} \right) b_j$$

Let l be the largest integer such that $0 \le l \le k\text{-}1$ and $a_l \neq 0$. Similarily, let m be the largest integer such that $0 \le m \le d\text{-}1$ and $b_m \neq 0$. In otherwords, a_l, b_m are the last non-zero entries of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ respectively. Then,

$$\mathbf{a}^T X\ \mathbf{b} = \sum_{j=0}^{m} \left(\sum_{i=0}^{l} a_i\, x_{i+j} \right) b_j .$$

Clearly, $a_l b_m x_{l+m} \neq 0$ and since this is the only term, of the above summation, involving x_{l+m}, it follows that $\boldsymbol{a}^T X \boldsymbol{b} \neq 0$, for all non-zero $\boldsymbol{a} \in F^k$ and $\boldsymbol{b} \in F^d$.
Moreover,

$$\rho(\boldsymbol{a}^T X) = d \quad \forall \text{ non-zero } \boldsymbol{a} \in F^k.$$

For,

$$\mathbf{a}^T X = \left[\textstyle\sum_{i=0}^{k-1} a_i\, x_i \ , \ \sum_{i=0}^{k-1} a_i\, x_{i+1}, \quad \cdot \quad \cdot \quad , \quad \sum_{i=0}^{k-1} a_i\, x_{i+d-1} \right].$$

and if there exist $f_0, f_1, \ldots, f_{d\text{-}1} \in F$, not all zero, such that

$$f_0 \sum_{i=0}^{k-1} a_i\, x_i + f_1 \sum_{i=0}^{k-1} a_i\, x_{i+1} + \cdot\cdot + f_{d-1} \sum_{i=0}^{k-1} a_i\, x_{i+d-1} = 0 \, ,$$

then taking $\boldsymbol{b} = (f_0, f_1, \ldots, f_{d-1})$, we have

$$\mathbf{a}^T X \mathbf{b} = \sum_{j=0}^{d-1}\left(\sum_{i=0}^{k-1} a_i x_{i+j}\right) f_j = 0$$

-contradicting $\boldsymbol{a}^T X \boldsymbol{b} \neq 0$. Hence $f_j = 0$ $j = 0, 1, \ldots, d\text{-}1$ and $\rho(\boldsymbol{a}^T X) = d$.

Thus, if ϑ is computed in the form $C(A\boldsymbol{x} \text{ x } B\boldsymbol{y})$, it follows from Theorem 3.5 that C is the generator matrix of a (n, k, d') linear code over F where $d' \geq d = \rho(\boldsymbol{a}^T X)$. n denotes the multiplicative complexity, which by Theorem 3.4, is at least k+d-1. When n is actually equal to this lower bound, the corresponding code is known as maximum-distance-separable codes.

In an attempt to derive an algorithm for computing system (3.6), consider the *P-dual* Φ of the computation, $C(A\boldsymbol{x} \text{ x } B\boldsymbol{y})$. Recall, the *P-dual* Φ is $A^T(C^T\boldsymbol{z} \text{ x } B\boldsymbol{y})$ where $\boldsymbol{z} = (z_0, z_1, \ldots\ldots, z_{k-1})^T$. It can be shown that the *P-dual* computation corresponds to the lower triangular sytem of bilinear forms (3.7) given below (for details see Appendix A):

$$\Phi = \begin{bmatrix} \phi_0 \\ \phi_1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \phi_{N-1} \end{bmatrix} = Z\,\mathbf{y} = \begin{bmatrix} z_0 & 0 & 0 & \cdot & \cdot & \cdot & 0 & 0 \\ z_1 & z_0 & 0 & \cdot & \cdot & \cdot & 0 & 0 \\ z_2 & z_1 & z_0 & \cdot & \cdot & \cdot & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ z_{k-1} & z_{k-2} & z_{k-3} & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & z_{k-1} & z_{k-2} & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & z_{k-1} & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & z_0 & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & z_0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & z_{k-1} & z_{k-2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & z_{k-1} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \cdot \\ \cdot \\ y_{d-1} \end{bmatrix}$$

Consider the polynomials

$$Z(u) = \sum_{j=0}^{k-1} z_j u^j \quad \text{and} \quad Y(u) = \sum_{l=0}^{d-1} y_l u^l .$$

Looking at the form of matrix Z in (3.7), it is obvious that the $N = k$+d-1 bilinear forms $\phi_0, \ldots, \phi_{N-1}$ of Φ are simply the coefficients of the polynomial $\Phi(u)$ where

$$\Phi(u) = \sum_{i=0}^{N-1} \phi_i u^i = Z(u)Y(u)$$

and

$$\phi_i = \sum_{\substack{0 \le j \le k-1 \\ 0 \le l \le d-1 \\ j+l=i}} z_j \, y_l \; .$$

We say that the sequence $\phi_0, \ldots, \phi_{N-1}$ is the *simple aperiodic convolution* of the sequences z_j, $j = 0,1, \ldots, k-1$ and y_l, $l = 0, 1, \ldots, d-1$. Hence, the dual bilinear form (3.7) may be computed as the polynomial product

$$\Phi(u) = Z(u)Y(u) \qquad (3.8)$$

Two methods which can be used to find this product will be studied in Chapter 4.

Recall, Φ is the *P-dual* of the computation $\vartheta = C(Ax \times By)$. Thus, Φ may be expressed in terms of matrices A, B and C as $\Phi = A^T(C^T z \times By)$. Also, we stated previously that if ϑ is computed as $C(Ax \times By)$, this computation requires n multiplications. It follows from Theorem 3.2 that the *P-dual* computation Φ has n multiplications. So, by (3.8), n is the multiplicative complexity of the aperiodic convolution of length $N = k+d-1$. Finally, if this *P-dual* is computed as $\Phi = P(Qz \times Ry)$, then $C = Q^T$. Thus we have shown that by computing the aperiodic convolution of two sequences, we compute the *P-dual* Φ given by (3.7) . From this, we may obtain the matrix C - the generator matrix of a (n, k, d') linear code over F (where $d' \ge d$).

CHAPTER 4 - Efficient Algorithms for the Aperiodic Convolution of Two Sequences

We are now aware of the importance of the aperiodic convolution of two sequences in the generation of a special class of linear codes. The multiplicative complexity of the aperiodic convolution is equal to the length of the corresponding linear codes. Thus, we wish to develop efficient algorithms to compute the aperiodic convolution of two sequences. We shall look at two methods [2] :

(i) a large degree polynomial product is represented as a number of small degree polynomial products.

(ii) a one-dimensional polynomial product is converted into multidimensional polynomial products.

4.1 - Convolution Algorithms Based on the CRT

Consider the polynomial product

$$\Phi(u) = Z(u)Y(u)$$

where the polynomials $Z(u)$ and $Y(u)$, of degree k-1 and d-1 respectively, are as described in Chapter 3. Recall, $N = k+d-1$. $\Phi(u)$, being of degree N-1, is unchanged if it is defined modulo any polynomial $P(u)$, provided $\deg[P(u)] \geq N$.
i.e.

$$\Phi(u) \equiv Z(u)Y(u) \quad \text{modulo } P(u) \quad \text{and } \deg[P(u)] \geq N.$$

Suppose $P(u)$ is the product of t co-prime polynomials.
i.e.

$$P(u) = \prod_{i=1}^{t} P_i(u) \text{ where } \deg[P_i(u)] = \alpha_i \text{ and } \sum_{i=1}^{t} \alpha_i \geq N .$$

Then, $\Phi(u)$ may be found by first reducing the polynomials $Z(u)$ and $Y(u)$ modulo $P_i(u)$ $(i = 1, 2, . ., t)$.
i.e.

$$Z_i(u) \equiv Z(u) \text{ modulo } P_i(u)$$
$$Y_i(u) \equiv Y(u) \text{ modulo } P_i(u)$$
$$\text{for } i = 1, 2, . . , t .$$

These form t congruences $\Phi_i(u)$ given by,

$$\Phi_i(u) \equiv Z_i(u)Y_i(u) \text{ modulo } P_i(u)$$
$$\text{for } i = 1,2,. . ,t .$$

Using the CRT, $\Phi(u)$ may be uniquely reconstructed from these products. Let $M(\alpha_i)$ be the number of multiplications required to calculate the polynomial product

$\Phi_i(u) \equiv Z_i(u)Y_i(u)$ modulo $P_i(u)$ where $\deg[P_i(u)] = \alpha_i$. It should be noted that $M(\alpha_i)$ depends on both the degree of $P_i(u)$ and on the actual form of $P_i(u)$.

n, the length of the linear code generated is equal to the multiplicative complexity of the above computation and is given by

$$n = M(N) = \sum_{i=1}^{t} M(\alpha_i).$$

We noted earlier, the wish to develop algorithms with as low a multiplicative complexity as possible for desired values of k and d. If $P(u)$ has degree D, its factors $P_i(u)$ ($i = 1, 2, .., t$) are chosen so as to minimise the complexity of the computation $Z(u)Y(u)$ modulo $P(u)$. It has been found that a large number of small degree coprime polynomials tends to result in an efficient algorithm and hence the following conditions should be satisfied :

(i) $P_i(u)$ and $P_j(u)$ have no nontrivial common factors, $1 \leq i < j \leq t$, $i \neq j$,

(ii) $\sum_{i=1}^{t} \deg[P_i(u)] = \deg[P(u)] = D$ and

(iii) Each of the polynomials $P_i(u)$ is of the lowest possible degree.

Obviously the resulting $P(u)$ will depend on the field of constants F. For example, if $D = 3$ then over $GF(2)$, $P(u) = (u+1)(u^2+u+1)$ would be the 'best' choice while over $GF(3)$, $P(u) = u(u+1)(u+2)$ would minimise the complexity of computing $Z(u)Y(u)$ modulo $P(u)$.

The efficiency of the algorithm can be further improved by modifying the above procedure to permit intentional wraparound of the polynomial coefficients.

e.g. if $M(N\text{-}1) < M(N)$ then it is more efficient to calculate $\Phi(u)$ from the product $Z(u)Y(u)$ modulo $P'(u)$ where $\deg[P'(u)] = N\text{-}1$, with one extra multiplication $z_{k-1}.y_{d-1}$.

Similarly, where $M(N\text{-}2) < M(N\text{-}1)\text{-}2 < M(N)\text{-}3$, it may be desirable to compute $\Phi(u)$ from the product $Z(u)Y(u)$ modulo $P''(u)$, $\deg[P''(u)] = N\text{-}2$ with three additional multiplications.

Further, it should be noted that the computation $Z(u)Y(u)$ modulo $P_i(u)$ where $P_i(u) = (u - a_i)^{\alpha_i}$ requires fewer multiplications than a similar computation where $P_i(u) = (u - a_j)^{\alpha_j}..(u - a_k)^{\alpha_k}$ and $\deg[P_i(u)] = \alpha_i$.

The modified algorithm to compute the product $Z(u)Y(u)$ where $\deg[Z(u)] = k\text{-}1$ and $\deg[Y(u)] = d\text{-}1$ can be generalised by the following :

Method 1 Computation of $Z(u)Y(u)$

If $k = d = 1$, then clearly the result is obtained by one multiplication. Otherwise, an integer s is chosen so as to minimise the total number of multiplications needed to compute

(i) $Z(u)Y(u)$ modulo $P(u)$ where $\deg[P(u)] = N\text{-}s$,

and

(ii) $\overline{Z}(u)\overline{Y}(u)$ modulo u^s where $\overline{Z}(u) = Z(u^{-1})u^{k-1}$ and $\overline{Y}(u) = Y(u^{-1})u^{d-1}$.

Then, s denotes the number of wraparounds and $N = \deg[P(u)]+s$.

Method 2 To find $Z_i(u)Y_i(u)$ modulo $P_i(u)$

There are two cases:

2(i) $P_i(u) = (u - a_i)^{\alpha_i}$, $a_i \in F$ then use method 3

2(ii) $P_i(u) \neq (u - a_i)^{\alpha_i}$ then compute the ordinary polynomial product $Z_i(u)Y_i(u)$ and reduce modulo $P_i(u)$

Method 3 To find $Z_i(u)Y_i(u)$ modulo $P_i(u)$ where $P_i(u) = (u - a_i)^{\alpha_i}$, $a_i \in F$

Again there are two cases :

3(i) $a_i = 0$ then use which ever of the following 2 methods has lower multiplicative complexity

(a) Use method 2(ii)

or

(b) Let

$$Z_i(u) = z_0' + z_1'u \ldots + z_{\alpha_i - 1}'u^{\alpha_i - 1}$$
$$Y_i(u) = y_0' + y_1'u + \ldots + y_{\alpha_i - 1}'u^{\alpha_i - 1}$$

and define

$$m_{s_1 s_2} = \left(\sum_{e=s_1}^{s_2} z_e'\right)\left(\sum_{f=s_1}^{s_2} y_f'\right).$$

Then,

$$m_{s_1 s_2} + m_{s_1+1, s_2-1} - m_{s_1, s_2-1} - m_{s_1+1, s_2} = z_{s_1}'y_{s_2}' + z_{s_2}'y_{s_1}'$$

and [10]

$$M(Z_i(u)Y_i(u) \text{ modulo } P_i(u)) = \begin{cases} \beta(\beta+1)-1 & \text{for} \quad \alpha_i = 2\beta - 1 \\ \beta(\beta+2) & \text{for} \quad \alpha_i = 2\beta \end{cases}.$$

3(ii) $a_i \neq 0$,define

$$Z_i(u)^* = Z_i(u + a_i)$$
$$Y_i(u)^* = Y_i(u + a_i)$$

Now, compute $\Phi_i(u)^* = Z_i(u)^*Y_i(u)^*$ modulo u^{α_i} as described in 3(i). Then, $\Phi_i(u) = \Phi_i(u - a_i)^*$.

If all the computations $Z_i(u)Y_i(u)$ modulo $(u+a_i)^{\alpha_i}$ are performed using method 2(ii) above. Then, for $\alpha_i \leq 2$, this does not result in any increase in multiplicative complexity and so the length of the code remains unchanged. For $\alpha_i > 2$, although there is an increase in complexity , it is very small and dependent on the field of constants [2].

Using the basic procedure outlined above, it is possible to design bilinear algorithms for a given value N . Examples using this algorithm as presented in Chapter 5.

4.2 - Multidimensional Convolution Algorithms

We shall now describe briefly a second method which may be used for computing the polynomial product

$$\Phi(u) = Z(u)Y(u)$$

where, as before, $\Phi(u)$, $Z(u)$ and $Y(u)$ are polynomials of degree N-1, k-1 and d-1 with coefficients ϕ_i, $i = 0,1,. ., N-1$, $z_j, j = 0, 1, . ., k-1$ and y_l, $1 = 0, 1, . ., d-1$ respectively.

Let k and d be composite numbers sharing a common factor c.
i.e.

$$k = k_1 c \text{ and } d = d_1 c$$

Then define $j = cj_2+j_1, j_2 = 0, 1, . ., k_1-1, j_1= 0, 1, . ., c-1, 1 = cl_2+l_1, l_2 = 0, 1, . ., d_1-1, l_1 = 0, 1, . ., c-1$ and $u_1 = u^c$.

Let $Z(u, u_1)$, $Y(u, u_1)$ and $\Phi(u, u_1)$ be the two dimensional polynomials corresponding to the polynomials $Z(u)$, $Y(u)$ and $\Phi(u)$ respectively. So

$$Z(u) = Z(u,u_1) = \sum_{j_1=0}^{c-1} Z_{j_1}(u_1)u^{j_1}$$

$$Y(u) = Y(u,u_1) = \sum_{l_1=0}^{c-1} Y_{l_1}(u_1)u^{l_1}$$

where

$$Z_{j_1}(u_1) = \sum_{j_2=0}^{k_1-1} z_{cj_2+j_1} u_1^{j_2}$$

$$Y_{l_1}(u_1) = \sum_{l_2=0}^{d_1-1} y_{cl_2+l_1} u_1^{l_2}$$

and so

$$\Phi(u,u_1) = \sum_{l_1=0}^{c-1} \sum_{j_1=0}^{c-1} Z_{j_1}(u)Y_{l_1}(u)\, u^{j_1+l_1} . \qquad (4.1)$$

Then $Z(u, u_1)$ is a polynomial of degree c-1 in u where each coefficient is a polynomial of degree k_1-1 in u_1. An analogous statement may be made about $Y(u, u_1)$.

$\Phi(u, u_1)$ is computed as the product of two polynomials each of degree c-1. In this computation, every multiplication is substituted by the product of two polynomials of degrees k_1-1 and d_1-1. If

n_1 = no of multiplications required to compute the product of 2 polynomials of degrees k_1-1 and d_1-1

and

n_2 = no of multiplications required to compute the product of 2 polynomials each of degree c-1.

Then, using (4.1), $\Phi(u) = Z(u)Y(u)$ has multiplicative complexity n where,

$$n = n_1 n_2.$$

This method may be extended to obtain m-dimensional convolution algorithms (m>2). In the 2-D case, it should be noted that the first dimension corresponds to the generation of a code C_1 where code dimension = minimum distance = c. While, the second dimension corresponds to the generation of a code C_2 of code dimension k_1 and minimum distance d_1. The resulting code is simply the product of C_1 and C_2. Such codes are referred to as *product* codes. Since a great deal is already known about these codes, we will not develop this topic any further. An example of this multi-dimensional approach for computing $\Phi(u) = Z(u)Y(u)$ is given in reference[2].

The aperiodic convolution algorithm and the multi-dimensional convolution algorithm are compared by their associated multiplicative complexity. Since the multiplicative complexity is equal to the length of the associated linear code, usually the algorithm involving the least multiplications would be favoured. However, when k or d is prime or when k and d share no non-trivial common factor, we have no alternative but to use the aperiodic convolution algorithm.

CHAPTER 5 - KM CODES

We now wish to compute the aperiodic convolution of 2 sequences of length k and d respectively over a specified field F and hence obtain the generator matrix of the corresponding linear code. These codes shall be referred to as *KM* (*Krishna & Morgera*) codes. The convolution algorithm based on the CRT (see Section 4.1) will be used. The non-commutative algorithms for small degree polynomial multiplication of Appendix B are used in these examples.

5.1 Binary Codes

Firstly, we will work over $GF(2)$ in order to obtain binary KM codes.

Example 5.1.1 - Bilinear Convolution Algorithm of Length $N = 8$ & the Corresponding Code

$N = 8 = k+d-1$. Hence $k+d = 9$. Let $k = 6$ and $d = 3$.

Then $Z(u) = z_0 + z_1u + z_2u^2 + z_3u^3 + z_4u^4 + z_5u^5$, $Y(u) = y_0 + y_1u + y_2u^2$ and $\Phi(u) = \phi_0 + \phi_1u + \phi_2u^2 + \phi_3u^3 + \phi_4u^4 + \phi_5u^5 + \phi_6u^6 + \phi_7u^7$. Since $\deg[P(u)] + s = N$, and $P(u)$ has to be chosen so as to minimise the multiplicative complexity of the algorithm, we take $\deg[P(u)] = 6$ and allow for two wraparounds i.e. $s = 2$

Note This choice for D and s is not unique. $D = 7$ and $s = 1$ would also produce a code of 'minimal' length.

Then,

$$P(u) = u^2(u^2 + 1)(u^2 + u + 1) = P_1(u)P_2(u)P_3(u).$$

We begin by reducing $Z(u)$ and $Y(u)$ modulo $P_i(u)$, $i = 1, 2, 3$.

$i = 1$

$$Z_1(u) \equiv Z(u) \text{ modulo } u^2$$
$$= z_0 + z_1u$$

and

$$Y_1(u) \equiv Y(u) \text{ modulo } u^2$$
$$= y_0 + y_1u.$$

Let,

$$m_0 = z_0.y_0,$$
$$m_1 = z_1.y_1$$

and

$$m_2 = (z_0 + z_1).(y_0 + y_1).$$

Then,

$$\Phi_1(u) \equiv \Phi(u) \text{ modulo } u^2$$
$$= m_0 + (m_0 + m_1 + m_2)u .$$

$i = 2$

$$Z_2(u) \equiv Z(u) \text{ modulo } (u^2+1)$$
$$= (z_0 + z_2 + z_4) + (z_1 + z_3 + z_5)u$$

and

$$Y_2(u) \equiv Y(u) \text{ modulo } (u^2+1)$$
$$= (y_0 + y_2) + y_1 u.$$

Let,

$$m_3 = (z_0 + z_2 + z_4).(y_0 + y_2),$$
$$m_4 = (z_1 + z_3 + z_5).y_1$$

and

$$m_5 = (z_0 + z_1 + z_2 + z_3 + z_4 + z_5).(y_0 + y_1 + y_2) .$$

Then,

$$\Phi_2(u) \equiv \Phi(u) \text{ modulo } (u^2+1)$$
$$= (m_3 + m_4) + (m_3 + m_4 + m_5)u .$$

$i = 3$

$$Z_3(u) \equiv Z(u) \text{ modulo } (u^2+u+1)$$
$$= (z_0 + z_2 + z_3 + z_5) + (z_1 + z_2 + z_4 + z_5)u$$

and

$$Y_3(u) \equiv Y(u) \text{ modulo } (u^2+u+1)$$
$$= (y_0+y_2) + (y_1+y_2)u.$$

Let,

$$m_6 = (z_0 + z_2 + z_3 + z_5).(y_0 + y_2),$$
$$m_7 = (z_1 + z_2 + z_4 + z_5).(y_1 + y_2)$$

and

$$m_8 = (z_0 + z_1 + z_3 + z_4).(y_0 + y_1).$$

Then,

$$\Phi_3(u) \equiv \Phi(u) \text{ modulo } (u^2+u+1)$$
$$= (m_6 + m_7) + (m_6 + m_8)u .$$

Using the CRT, we are able to recover the polynomial $\Phi(u)$ modulo $P(u)$, from the polynomials $\Phi_i(u)$.
i.e.

$$\Phi(u) \equiv \sum_{i=1}^{3} S_i(u)\Phi_i(u) \text{ modulo } P(u)$$

The polynomials $S_i(u)$, i =1, 2, 3 are found as follows (using the same notation as in definition of the CRT(Theorem 3.1)) :

$i = 1$

$$R_1(u)P_2(u)P_3(u) \equiv 1 \text{ modulo } u^2$$
$$\text{i.e. } R_1(u)(u^2+1)(u^2+u+1) \equiv 1 \text{ modulo } u^2$$
$$\text{i.e. } R_1(u)(u^4+u^3+u+1) \equiv 1 \text{ modulo } u^2$$

Hence $R_1(u) = u+1$ and $S_1(u) = u^5+u^3+u^2+1$.

$i = 2$

$$R_2(u)P_1(u)P_3(u) \equiv 1 \text{ modulo } (u^2+1)$$
$$\text{i.e. } R_2(u)u^2(u^2+u+1) \equiv 1 \text{ modulo } (u^2+1)$$
$$\text{i.e. } R_2(u)(u^4+u^3+u^2) \equiv 1 \text{ modulo } (u^2+1)$$

Hence $R_2(u) = u$ and $S_2(u) = u^5+u^4+u^3$.

$i = 3$

$$R_3(u)P_1(u)P_2(u) \equiv 1 \text{ modulo } (u^2+u+1)$$
$$\text{i.e. } R_3(u)u^2(u^2+1) \equiv 1 \text{ modulo } (u^2+u+1)$$
$$\text{i.e. } R_3(u)(u^4+u^2) \equiv 1 \text{ modulo } (u^2+u+1)$$

Hence $R_3(u) = 1$ and $S_3(u) = u^4+u^2$.

Then,

$$\Phi(u) \equiv \sum_{i=1}^{3} S_i(u)\Phi_i(u) \text{ modulo } P(u)$$

$$\begin{aligned}\equiv\ &(1+u^2+u^3+u^5)(m_0+(m_0+m_1+m_2)u)\\ &+ (u^3+u^4+u^5)(m_3+m_4+(m_3+m_4+m_5)u)\\ &+(u^2+u^4)(m_6+m_7+(m_6+m_8)u) \text{ modulo } P(u)\end{aligned}$$

$$\begin{aligned}\equiv\ &m_0 + (m_0+m_1+m_2)u + (m_0+m_6+m_7)u^2\\ &+ (m_1+m_2+m_3+m_4+m_6+m_8)u^3 + (m_0+m_1+m_2+m_5+m_6+m_7)u^4\\ &+ (m_0+m_5+m_6+m_8)u^5 + (m_0+m_1+m_2+m_3+m_4+m_5)u^6\\ &\text{modulo } (u^6+u^5+u^3+u^2)\end{aligned}$$

$$\begin{aligned}=\ &m_0 + (m_0+m_1+m_2)u + (m_1+m_2+m_3+m_4+m_5+m_6+m_7)u^2\\ &+ (m_0+m_5+m_6+m_8)u^3 + (m_0+m_1+m_2+m_5+m_6+m_7)u^4\\ &+ (m_1+m_2+m_3+m_4+m_6+m_8)u^5\end{aligned}$$

Hence, if $\Phi\text{^}(u) \equiv \Phi(u)$ modulo $P(u)$ then,

$$\Phi\text{^}(u) = \phi_0\text{^}+ \phi_1\text{^}u+ \phi_2\text{^}u^2+ \phi_3\text{^}u^3+ \phi_4\text{^}u^4+ \phi_5\text{^}u^5$$

where,

$\phi_0\text{^} = m_0$,

$\phi_1\text{^} = m_0+m_1+m_2$,

$\phi_2\text{^} = m_1+m_2+m_3+m_4+m_5+m_6+m_7$,

$\phi_3\text{^} = m_0+m_5+m_6+m_8$,

$\phi_4\text{^} = m_0+m_1+m_2+m_5+m_6+m_7$

and

$\phi_5\wedge = m_1+m_2+m_3+m_4+m_6+m_8.$

Intentional Wraparound

Since two wraparounds are involved, we wish to calculate $\overline{Z}(u)\overline{Y}(u)$ modulo u^2.

Now,

$$\overline{Z}(u) = z_5+z_4u+z_3u^2+z_2u^3+z_1u^4+z_0u^5$$
$$\equiv z_5+z_4u \text{ modulo } u^2$$

and

$$\overline{Y}(u) = y_2+y_1u+y_0u^2$$
$$\equiv y_2+y_1u \text{ modulo } u^2.$$

Let,

$$m_9 = z_5.y_2,$$
$$m_{10} = z_4.y_1$$

and

$$m_{11} = (z_4+z_5).(y_1+y_2).$$

Then,

$$\overline{Z}(u)\ \overline{Y}(u) \equiv m_9+(m_9+m_{10}+m_{11})u \text{ modulo } u^2.$$

Recall, the ordinary polynomial product is $\Phi(u) = Z(u)Y(u) = \phi_0+\phi_1u+\phi_2u^2+\phi_3u^3+\phi_4u^4+\phi_5u^5+\phi_6u^6+\phi_7u^7$ and $\Phi\wedge(u) \equiv \Phi(u)$ modulo $P(u)$. So the coefficients ϕ_i, $i = 1, 2, . ., 7$ are given by :

$$\phi_0 = m_0,$$
$$\phi_1 = m_0+m_1+m_2,$$
$$\phi_2 = m_1+m_2+m_3+m_4+m_5+m_6+m_7+m_{10}+m_{11},$$
$$\phi_3 = m_0+m_5+m_6+m_8+m_9+m_{10}+m_{11},$$
$$\phi_4 = m_0+m_1+m_2+m_5+m_6+m_7+m_9,$$
$$\phi_5 = m_1+m_2+m_3+m_4+m_6+m_8+m_{10}+m_{11},$$
$$\phi_6 = \text{coeff. of } u^6 = z_4y_2+z_5y_1= m_9+m_{10}+m_{11}$$

and

$$\phi_7 = \text{coeff. of } u^7 = z_5y_2 = m_9.$$

It follows that the bilinear form for the computation of the aperiodic convolution of two sequences of length 6 and 3 respectively is $P(Qz \times Ry)$ where matrices P, Q and R are given by,

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix},$$

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \text{ and } R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

C, the generator matrix for the corresponding binary (12, 6, 3) KM code is given by $C = Q^T$.
i.e.

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

Example 5.1.2 - Bilinear Convolution Algorithm of Length $N = 9$ & the Corresponding KM Code

Since $N = k+d-1=9$, take $k = 7$ and $d = 3$. Therefore $Z(u) = z_0 + z_1u + z_2u^2 + z_3u^3 + z_4u^4 + z_5u^5 + z_6u^6$ and $Y(u) = y_0 + y_1u + y_2u^2$. We must have $N = \deg[P(u)]+s$. We shall take $P(u) = u(u^2+1)(u^2+u+1)(u^3+u^2+1)$ and $s = 1$.

Proceeding as in Example 5.1.1,

$i = 1$

$$Z_1(u) \equiv Z(u) \text{ modulo } u$$

$$= z_0$$

and

$$Y_1(u) \equiv Y(u) \text{ modulo } u$$
$$= y_0.$$

Let,

$$m_0 = z_0.y_0 .$$

Then,

$$\Phi_1(u) \equiv \Phi(u) \text{ modulo } u$$
$$= m_0 .$$

$i = 2$

$$Z_2(u) \equiv Z(u) \text{ modulo } (u^2+1)$$
$$= (z_0+z_2+z_4+z_6) + (z_1+z_3+z_5)u$$

and

$$Y_2(u) \equiv Y(u) \text{ modulo } (u^2+1)$$
$$= (y_0+y_2) + y_1u.$$

Let,

$$m_1 = (z_0+z_2+z_4+z_6).(y_0+y_2),$$
$$m_2 = (z_1+z_3+z_5).y_1$$

and

$$m_3 = (z_0+z_1+z_2+z_3+z_4+z_5+z_6).(y_0+y_1+y_2) .$$

Then,

$$\Phi_2(u) \equiv \Phi(u) \text{ modulo } (u^2+1)$$
$$= (m_1+m_2) + (m_1+m_2+m_3)u.$$

$i = 3$

$$Z_3(u) \equiv Z(u) \text{ modulo } (u^2+u+1)$$
$$= (z_0+z_2+z_3+z_5+z_6) + (z_1+z_2+z_4+z_5)u$$

and

$$Y_3(u) \equiv Y(u) \text{ modulo } (u^2+u+1)$$
$$= (y_0+y_2) +(y_1+y_2)u.$$

Let,

$$m_4 = (z_0+z_2+z_3+z_5+z_6).\ (y_0+y_2),$$
$$m_5 = (z_1+z_2+z_4+z_5).\ (y_1+y_2)$$

and

$$m_6 = (z_0+ z_1+z_3+ z_4 +z_6).\ (y_0+y_1).$$

Then,

$$\Phi_3(u) \equiv \Phi(u) \text{ modulo } (u^2+u+1)$$
$$= (m_4+m_5) + (m_4+m_6)u.$$

$i = 4$

$$Z_4(u) \equiv Z(u) \text{ modulo } (u^3+u^2+1)$$
$$= (z_0+z_3+z_4+z_5) + (z_1+z_4+z_5+z_6)u + (z_2+z_3+z_4+z_6)u^2$$

and

$$Y_4(u) \equiv Y(u) \text{ modulo } (u^3+u^2+1)$$
$$= y_0 + y_1u + y_2u^2.$$

Let,

$$m_7 = (z_0+z_3+z_4+z_5).\ y_0,$$
$$m_8 = (z_1+z_4+z_5+z_6).\ y_1,$$
$$m_9 = (z_2+z_3+z_4+z_6).\ y_2,$$
$$m_{10} = (z_0+z_1+z_3+z_6).\ (y_0+ y_1),$$
$$m_{11} = (z_1+z_2+z_3+z_5).(y_1+ y_2)$$

and

$$m_{12} = (z_0+z_2+z_5+z_6).\ (y_0+ y_2).$$

Then,

$$\Phi_4(u) \equiv \Phi(u) \text{ modulo } (u^3+u^2+1)$$
$$= (m_7+m_8+m_{11}) + (m_7+m_8+m_9+m_{10})u + (m_7+m_9+m_{11}+m_{12})u^2.$$

Let,

$$\Phi^\wedge(u) \equiv \Phi(u) \text{ modulo } P(u).$$

Then, the polynomial $\Phi^\wedge(u)$ may be recovered from the polynomials $\Phi_i(u)$, $i = 1, 2, 3, 4$ using the CRT.
i.e.

$$\Phi^\wedge(u) = \sum_{i=1}^{4} S_i(u)\Phi_i(u) \tag{5.1}$$

It is straightforward to establish that $S_1(u) = (u^7+u^5+u^3+u^2+u+1)$, $S_2(u) = (u^7+u^3+u^2)$, $S_3(u) = (u^7+u^4+u^2+u)$ and $S_4(u) = (u^7+u^5+u^4+u^2)$.
Let,

$$\Phi^\wedge(u) = \phi_0^\wedge+ \phi_1^\wedge u+ \phi_2^\wedge u^2+ \phi_3^\wedge u^3+ \phi_4^\wedge u^4+ \phi_5^\wedge u^5+ \phi_6^\wedge u^6+ \phi_7^\wedge u^7 .$$

Then, by (5.1), we find

$$\phi_0^\wedge = m_0,$$
$$\phi_1^\wedge = m_0+m_1+m_2+m_3+m_5+m_6+m_7+m_8+m_9+m_{10},$$
$$\phi_2^\wedge = m_0+m_3+m_4+m_5+m_7+m_{10}+m_{12},$$
$$\phi_3^\wedge = m_0+m_1+m_2+m_7+m_9+m_{11}+m_{12},$$
$$\phi_4^\wedge = m_5+m_6+m_9+m_{10}+m_{11},$$
$$\phi_5^\wedge = m_0+m_4+m_6+m_7+m_{10}+m_{12},$$
$$\phi_6^\wedge = m_1+m_2+m_3+m_4+m_6+m_7+m_9+m_{11}+m_{12}$$

and

$$\phi_7^\wedge = m_0+m_1+m_2+m_4+m_5+m_7+m_8+m_{11}.$$

Wraparound

Clearly, $\overline{Z}(u) = z_6 + z_5u + z_4u^2 + z_3u^3 + z_2u^4 + z_1u^5 + z_0u^6$ and $\overline{Y}(u) = y_2 + y_1u + y_0u^2$. Recall $s = 1$, thus we compute,

$$\overline{Z}(u) \equiv z_6 \text{ modulo } u$$

and

$$\overline{Y}(u) \equiv y_2 \text{ modulo } u\ .$$

Let,

$$m_{13} = z_6.y_2.$$

If, $\Phi(u) = Z(u)Y(u) = \phi_0 + \phi_1u + \phi_2u^2 + \phi_3u^3 + \phi_4u^4 + \phi_5u^5 + \phi_6u^6 + \phi_7u^7 + \phi_8u^8$. Then, since $\Phi^\wedge(u) \equiv \Phi(u)$ modulo $P(u)$, we find

$\phi_0 = m_0,$
$\phi_1 = m_0+m_1+m_2+m_3+m_5+m_6+m_7+m_8+m_9+m_{10}+m_{13},$
$\phi_2 = m_0+m_3+m_4+m_5+m_7+m_{10}+m_{12}+m_{13},$
$\phi_3 = m_0+m_1+m_2+m_7+m_9+m_{11}+m_{12}+m_{13},$
$\phi_4 = m_5+m_6+m_9+m_{10}+m_{11}+m_{13},$
$\phi_5 = m_0+m_4+m_6+m_7+m_{10}+m_{12},$
$\phi_6 = m_1+m_2+m_3+m_4+m_6+m_7+m_9+m_{11}+m_{12}+m_{13},$
$\phi_7 = m_0+m_1+m_2+m_4+m_5+m_7+m_8+m_{11}$

and

$\phi_8 = \text{coeff. of } u^8 = m_{13}.$

Hence, the polynomial product $\Phi(u) = Z(u)Y(u)$ may be computed using the bilinear convolution algorithm $P(Qz \times Ry)$ where the matrices P, Q and R are given by,

$$P = \begin{bmatrix} 1&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 1&1&1&1&0&1&1&1&1&1&1&0&0&1 \\ 1&0&0&1&1&1&0&1&0&0&1&0&1&1 \\ 1&1&1&0&0&0&1&0&1&0&1&1&1&1 \\ 0&0&0&0&0&1&1&0&0&1&1&1&0&1 \\ 1&0&0&0&1&0&1&1&0&0&1&0&1&0 \\ 0&1&1&1&1&0&1&1&0&1&0&1&1&1 \\ 1&1&1&0&1&1&0&1&1&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&1 \end{bmatrix},$$

$$Q = \begin{bmatrix} 1&0&0&0&0&0&0\\ 1&0&1&0&1&0&1\\ 0&1&0&1&0&1&0\\ 1&1&1&1&1&1&1\\ 1&0&1&1&0&1&1\\ 0&1&1&0&1&1&0\\ 1&1&0&1&1&0&1\\ 1&0&0&1&1&1&0\\ 0&1&0&0&1&1&1\\ 0&0&1&1&1&0&1\\ 1&1&0&1&0&0&1\\ 0&1&1&1&0&1&0\\ 1&0&1&0&0&1&1\\ 0&0&0&0&0&0&1 \end{bmatrix} \text{ and } R = \begin{bmatrix} 1&0&0\\ 1&0&1\\ 0&1&0\\ 1&1&1\\ 1&0&1\\ 0&1&1\\ 1&1&0\\ 1&0&0\\ 0&1&0\\ 0&0&1\\ 1&1&0\\ 0&1&1\\ 1&0&1\\ 0&0&1 \end{bmatrix}.$$

C, the generator matrix of the associated (14, 7, 3) binary code is given by $C = Q^T$.
i.e.

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

5.2 - Some Advantages & Important Features of KM Codes

Using the above algorithm, Krishna and Morgera have derived linear codes of this type of up to length $n \leq 100$ and distance $d \leq 41$ [2]. (With the KM Codes Program of Appendix D, it is possible to derive KM codes of length and distance greater than these values.)

The columns of the generator matrix C correspond to the multiplications required to compute :

(i) the product $Z(u)Y(u)$ modulo $P(u)$

or

(ii) the wraparound $\overline{Z}(u)\overline{Y}(u)$ modulo u^s .

Hence, by shortening the columns of C, it is possible to decrease k and so increase d (since $N = k+d-1$ is fixed). The columns corresponding to (i) are shortened from the bottom while those associated with (ii) are shortened from the top. This suggests that a change in k does not result in a significant change in the encoding/decoding procedure. This shall be illustrated in Section 6.6.2, when the decoding procedure is generalised. As $k+d-1$ is fixed during these alternations, we can group these codes by their length n.
i.e. (18,9,3), (18,7,5), (18,5,7) all belong to the same set.

For a given value of k, we are able also to

(a) decrease the minimum distance, d, of the code by deleting columns of the generator matrix and so reducing the length n of the code

or alternatively

(b) increase d and thus n by adding columns to the generator matrix.

Properties (a) and (b) will be proved later (see Lemma 6.2 and Corollary 6.1). Further, the decoding of these families of codes where k is fixed is discussed in Section 6.7.

We have observed how versatile it is to alter the minimum distance d. Hence the error-correction capability of these codes can be easily changed, which is especially important in channels where the error rate is unstable.

Example 5.2.1

In example 5.1.1, we found that the generator matrix C of the binary (12, 6, 3) KM code was:

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

where the first nine columns correspond to (i) above and the final three to (ii).

Hence by reducing the length of the columns as described above,we may obtain C', the generator matrix of a (12,4,5) KM code. In fact, C' is given by :

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

Example 5.2.2

In example 5.1.2, we obtained the generator matrix of the (14, 7, 3) binary KM code :

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

The first thirteen columns of C correspond to the computation of $Z(u)Y(u)$ modulo $P(u)$ and only the final column corresponds to the wraparound. By reducing the lengths of the columns, as described above, we obtain C', the generator matrix of a (14,5,5) binary KM code.

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

Example 5.3 - A Comparision of the CRT-Based Convolution of Length 7 & the Related Code Over $GF(2)$ & $GF(3)$

It was stated earlier that the choice of $P(u)$ depends on the field of constants. To illustrate this, we shall derive a bilinear convolution algorithm of length $N = 7$ ($k = 5$, $d = 3$) and hence obtain the corresponding code over

(a) $GF(2)$

and

(b) $GF(3)$.

(a)

Over $GF(2)$, the 'best' choice of $P(u)$ is $P(u) = u^2(u^2+1)(u^2+u+1) = P_1(u)P_2(u)P_3(u)$ with one wraparound. Now, $Z(u) = z_0+z_1u+z_2u^2+z_3u^3+z_4u^4$ and $Y(u) = y_0+y_1u+y_2u^2$. Reducing the polynomials $Z(u)$ and $Y(u)$ modulo each of $P_i(u)$, we obtain,

$$Z_1(u) \equiv Z(u) \text{ modulo } u^2 \\ = z_0+z_1u$$

and

$$Y_1(u) \equiv Y(u) \text{ modulo } u^2 \\ = y_0+y_1u.$$

Let,

$$m_0 = z_0.y_0, \\ m_1 = z_1.y_1$$

and

$$m_2 = (z_0+z_1).(y_0+y_1).$$

Similarly,

$$Z_2(u) \equiv Z(u) \text{ modulo } (u^2+1) \\ = (z_0+z_2+z_4)+(z_1+z_3)u$$

and

$$Y_2(u) \equiv Y(u) \text{ modulo } (u^2+1) \\ = (y_0+y_2)+y_1u.$$

Let,

$$m_3 = (z_0+z_2+z_4).(y_0+y_2), \\ m_4 = (z_1+z_3).y_1$$

and

$$m_5 = (z_0+z_1+z_2+z_3+z_4).(y_0+y_1+y_2).$$

Also,

$$Z_3(u) \equiv Z(u) \text{ modulo } (u^2+u+1)$$
$$= (z_0+z_2+z_3)+(z_1+z_2+z_4)u$$

and

$$Y_3(u) \equiv Y(u) \text{ modulo } (u^2+u+1)$$
$$= (y_0+y_2)+(y_1+y_2)u.$$

Let,

$$m_6 = (z_0+z_2+z_3).(y_0+y_2),$$
$$m_7 = (z_1+z_2+z_4).(y_1+y_2)$$

and

$$m_8 = (z_0+z_1+z_3+z_4).(y_0+y_1).$$

Wraparound

We need to compute $\overline{Z}(u)\overline{Y}(u)$ modulo u where $\overline{Z}(u) = z_4+z_3u+z_2u^2+z_1u^3+z_0u^4$ and $\overline{Y}(u) = y_2+y_1u+y_0u^2$.

Let,

$$m_9 = z_4.y_2.$$

Then,

$$\overline{Z}(u)\ \overline{Y}(u) \equiv m_9 \text{ modulo } u.$$

The multiplications $m_0, \ldots, m_9$ are sufficient to compute the product $\Phi(u) = Z(u)Y(u)$. It follows that, C_1, the generator matrix for the associated (10, 5, 3) KM code over $GF(2)$ is

$$C_1 = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

(b)

Over $GF(3)$, the multiplicative complexity is minimum when $\deg[P(u)] = 6$ and one wraparound is allowed. Hence we take $P(u) = u^2(u+1)(u+2)(u^2+u+2) = P_1(u)P_2(u)P_3(u)P_4(u)$. Reducing $Z(u)$ and $Y(u)$ modulo each $P_i(u)$, we find

$$Z_1(u) \equiv Z(u) \text{ modulo } u^2$$
$$= z_0+z_1u$$

and

$$Y_1(u) \equiv Y(u) \text{ modulo } u^2$$
$$= y_0+y_1u\ .$$

Let,

$$m_0 = z_0.y_0,$$

$$m_1 = z_1.y_1$$

and

$$m_2 = (z_0+z_1).(y_0+y_1).$$

Similarly,

$$Z_2(u) \equiv Z(u) \text{ modulo } (u+1)$$
$$= z_0+2z_1+z_2+2z_3+z_4$$

and

$$Y_2(u) \equiv Y(u) \text{ modulo } (u+1)$$
$$= y_0+2y_1+y_2.$$

Let,

$$m_3 = (z_0+2z_1+z_2+2z_3+z_4).(y_0+2y_1+y_2).$$

Also,

$$Z_3(u) \equiv Z(u) \text{ modulo } (u+2)$$
$$= z_0+z_1+z_2+z_3+z_4$$

and

$$Y_3(u) \equiv Y(u) \text{ modulo } (u+2)$$
$$= y_0+y_1+y_2.$$

Let,

$$m_4 = (z_0+z_1+z_2+z_3+z_4).(y_0+y_1+y_2).$$

Also,

$$Z_4(u) \equiv Z(u) \text{ modulo } (u^2+u+2)$$
$$= (z_0+z_2+2z_3+2z_4)+(z_1+2z_2+2z_3)u$$

and

$$Y_4(u) \equiv Y(u) \text{ modulo } (u^2+u+2)$$
$$= (y_0+y_2)+(y_1+2y_2)u.$$

Let,

$$m_5 = (z_0+z_2+2z_3+2z_4).\ (y_0+y_2),$$
$$m_6 = (z_1+2z_2+2z_3).\ (y_1+2y_2)$$

and

$$m_7 = (z_0+z_1+z_3+2z_4).(y_0+y_1).$$

Wraparound

As in (a), we need to compute $\overline{Z}(u)\overline{Y}(u)$ modulo u .

Let,

$$m_8 = z_4.y_2.$$

Then,

$$\overline{Z}(u)\ \overline{Y}(u) \equiv m_8 \text{ modulo } u\ .$$

C_2, the generator matrix for the associated (9, 5, 3) KM code over $GF(3)$ is

$$C_2 = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 2 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 1 & 2 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 2 & 0 & 2 & 1 \end{bmatrix}.$$

Conclusion

For $k = 5$ and $d = 3$, the associated binary KM code has length 10 while the ternary KM code has length 9.

As the field of constants is extended, the number of polynomials of any degree over this field is also increased. Hence working over $GF(3)$, rather than $GF(2)$ results in a larger number of small degree polynomials, which we noted earlier usually leads to a more efficient algorithm.

CHAPTER 6 - A More Detailed Look at the Properties Of KM Codes

In this chapter, we shall discuss the process of encoding and decoding the linear codes generated by an aperiodic convolution algorithm based on the CRT (KM Codes).

The problem of error-detection and error-correction of these codes will also be studied.

Throughout, let the polynomial $Z(u) = z_0 + z_1u + \ldots + z_{k-1}u^{k-1}$, of degree $(k-1)$ represent the information vector.

6.1- Complexity of Encoding

Recall, $P(u) = \prod_{i=1}^{t} P_i(u)$.

Encoding corresponds to the following three steps :

(i) Compute $Z_i(u) \equiv Z(u)$ modulo $P_i(u)$ $i = 1, 2, \ldots, t$

(ii) Form the required linear combinations of $Z_i(u)$ so that the product $Z_i(u)Y_i(u)$ modulo $P_i(u)$ may be computed $i = 1, 2, \ldots, t$

(iii) Form the required linear combinations of $\overline{Z}(u)$ so that the product $\overline{Z}(u)\overline{Y}(u)$ modulo u^s may be computed.

As step (iii) is a special case of step (ii), we shall discuss only steps (i) and (ii).

Step (i) - $Z_i(u)$ is the remainder obtained when $Z(u)$ is divided by $P_i(u)$. This operation may be implemented by a division circuit, where the multipliers and adders are over the appropriate field. Hence, to implement step(i), a total of t such circuits are required, one for each $P_i(u)$ $i = 1, 2, \ldots, t$.

Step(ii) - When $P_i(u)$ is irreducible, the product $Z_i(u)Y_i(u)$ modulo $P_i(u)$ is obtained by computing the ordinary polynomial product $Z_i(u)Y_i(u)$ and then reducing it modulo $P_i(u)$. In such a case, the equations for the linear combinations of the coefficients of $Z_i(u)$ are independent of $P_i(u)$.

Shift registers which perform division by certain polynomials are given in Figure 6.1 while Figure 6.2 shows the linear combinations of the coefficients of $Z_i(u)$ for some polynomials over $GF(2)$. Figure 6.3 shows the shift register which may be used as an encoder for the (12, 6, 3) binary KM code derived in Example 5.1.1. Recall, $P(u)=u^2(u^2+1)(u^2+u+1)$ and $s = 2$. Hence the encoder has in total 4 division circuits; 3 corresponding to the computation $Z_i(u) \equiv Z(u)$ modulo $P_i(u)$ $(i = 1, 2, 3)$ and one for the wraparound. The circuit has 8 storage units and 7 adders.

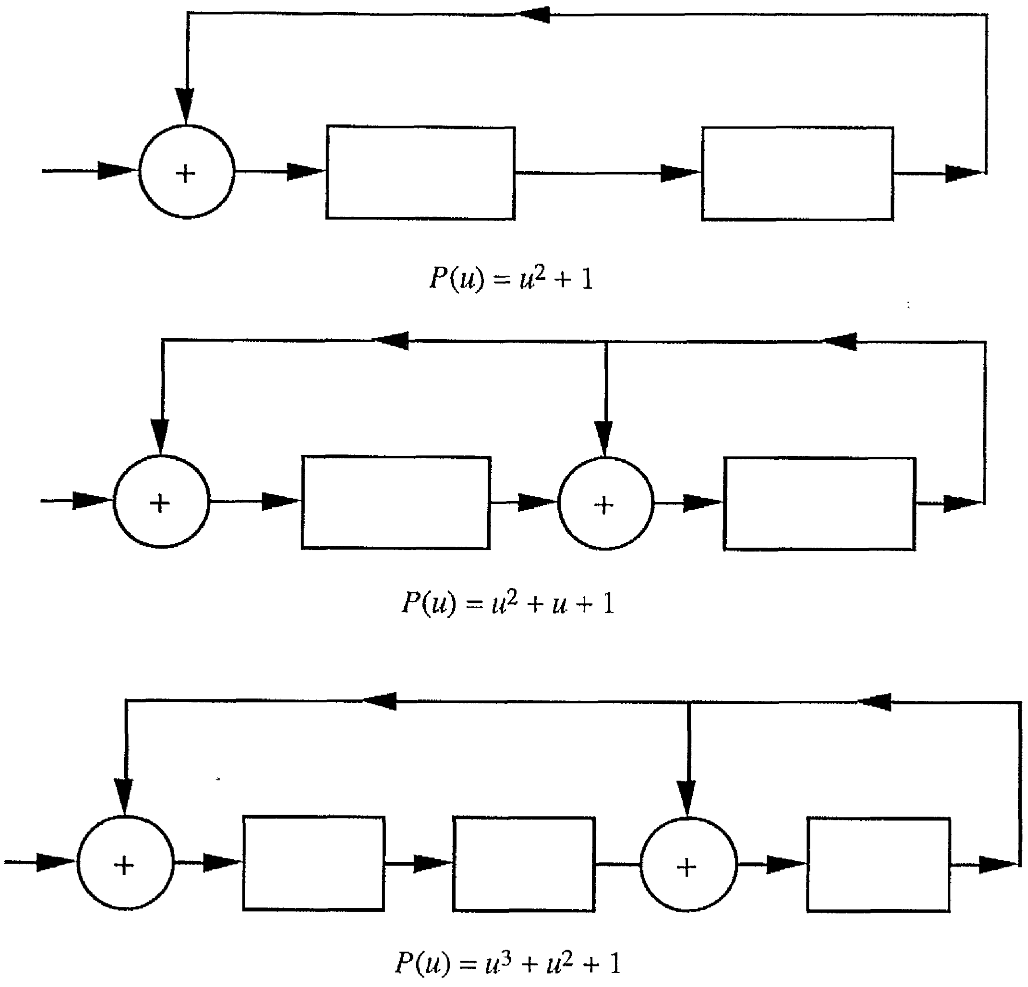


Fig 6.1 - Shift Register cicuits for division by polynomial P(u).

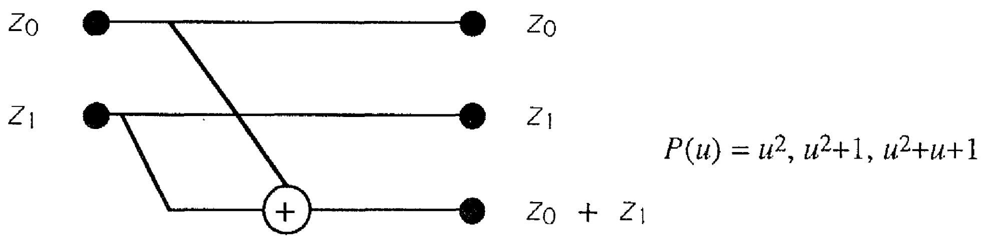


Figure 6.2 - Linear combinations of the coefficients of $Z(u)$ required for the computation $Z(u)Y(u)$ modulo $P(u)$.

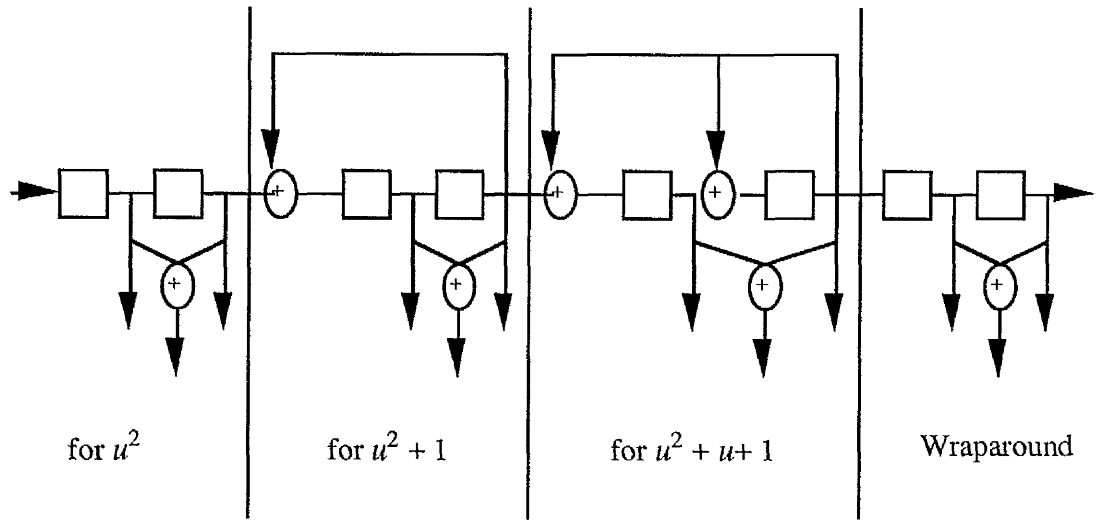


Figure 6.3- A shift register implementation of the encoder of (12, 6, 3) binary code.

6.2 - The Block Structure of the Generator Matrix of KM Codes

The block structure of the generator matrix C of these codes has been observed previously. Since this structure plays such an important role in the decoding procedures, we shall now discuss it in more detail.

There are $(t+1)$ blocks in the generator matrix C of an (n, k, d) KM code. We shall denote each of these blocks as C_i, $i = 1, 2, \ldots, t+1$.

The first t blocks arise from computations of the form

$$\Phi_i(u) \equiv Z_i(u)Y_i(u) \text{ modulo } P_i(u),$$

one for each of the t relatively prime polynomials $P_i(u)$ of degree α_i $(i = 1, 2, \ldots, t)$ where $P(u) = \prod_{i=1}^{t} P_i(u)$ and $\sum_{i=1}^{t} \alpha_i = D = \deg[P(u)]$.

The final block corresponds to the computation $\overline{Z(u)}\,\overline{Y}(u)$ modulo u^s (where $D+s = N$), the wraparound coefficients.

Each block has $M(\alpha_i)$ columns, $i = 1, 2, \ldots, t+1$ $(\alpha_{t+1} = s)$. As each block corresponds to a computation of the type described above, it is clear that in block C_i, α_i columns are linearly independent and the remaining $(M(\alpha_i)-\alpha_i)$ columns are dependent on them.

i.e. each block corresponds to the computation $\Phi_i(u) \equiv Z_i(u)Y_i(u)$ modulo $P_i(u)$ where $\deg[P_i(u)] = \alpha_i$. Then $\Phi_i(u)$ has degree α_{i-1}. If $\phi_{i,1}u + \ldots + \phi_{i,\alpha i-1}u^{\alpha_i-1}$, then the α_i coefficients $\phi_{i,0}, \phi_{i,1}, \ldots, \phi_{i,\alpha i-1}$ are linearly independent.

Further, the columns of C_i $(i = 1, 2, \ldots, t)$ can always be re-arranged so that the first α_i columns correspond to the polynomial

$$Z_i(u) \equiv Z(u) \text{ modulo } P_i(u) \quad i = 1, 2, \ldots, t$$

while, the remaining $[M(\alpha_i)-\alpha_i]$ columns are due to the multiplicative complexity of the block C_i.

Lemma 6.1

If the polynomial $P_i(u)$ is irreducible, then the block C_i corresponding to the computation $\Phi_i(u) \equiv Z_i(u)Y_i(u)$ modulo $P_i(u)$ is an $(n_i, \alpha_i, \alpha_i)$ code where $n_i = M(\alpha_i)$ and $\alpha_i = \deg[P_i(u)]$ [2].

Proof

When $P_i(u)$ is irreducible, the above computation is performed in two stages :

(i) The ordinary polynomial product $\Phi_i'(u) = Z_i(u)Y_i(u)$ is found

(ii) $\Phi_i'(u)$ is reduced modulo $P_i(u)$ to obtain $\Phi_i(u)$

i.e. $\Phi_i(u) \equiv \Phi_i'(u)$ modulo $P_i(u)$

The multiplicative complexity of step(ii) above is $n_i = M(\alpha_i)$ and so the corresponding block C_i will have n_i columns. Consider C_i to be the generator matrix of a code. Then the rows of C_i are codewords each of length n_i.

We observed above that in each block C_i there are exactly α_i linearly independent columns. Hence since the number of linearly independent columns is equal to the number of linearly independent rows, it follows that C_i has α_i linearly independent rows. Hence there are α_i linearly independent codewords within the matrix C_i. Clearly the dimension of the code C_i generates must be α_i. By a similar argument, it follows that the minimum distance of this code is α_i.

In conclusion, C_i is a $(n_i, \alpha_i, \alpha_i)$ code $(i = 1, 2, \ldots, t)$.

#

Note

Although Lemma 6.1 refers to an irreducible polynomial $P_i(u)$, the result holds for any polynomial $P_i(u)$ provided $\Phi_i(u) \equiv Z_i(u)Y_i(u)$ modulo $P_i(u)$ is calculated via the above 2 stages.

Lemma 6.2

If the block C_j corresponding to the computation $\Phi_j(u) \equiv Z_j(u)Y_j(u)$ modulo $P_j(u)$, is removed from C, the generator matrix of a (n, k, d) code, then the resulting matrix is the generator matrix of a $(n-n_j, k, d-\alpha_j)$ code where $\deg[P_j(u)] = \alpha_j$ and $n_j = M(\alpha_j)$ [2].

Proof

Recall,

$$C = [C_1 \mid C_2 \mid \ldots . \mid C_{t+1}]$$

where block C_j $(i = 1, 2, . . .,t)$ corresponds to a computation of the form

$$\Phi_j(u) \equiv Z_j(u)Y_j(u) \text{ modulo } P_j(u) \quad \deg[P_j(u)]=\alpha_j$$

while block C_{t+1} corresponds to the wraparound at s points.
Also,

$$P(u) = \prod_{j=1}^{t} P_j(u)$$

and

$$N = \sum_{j=1}^{t} \alpha_j + s$$

Then, letting $\alpha_{t+1}=s$, we obtain

$$N = \sum_{j=1}^{t+1} \alpha_j = k + d - 1 \tag{6.1}$$

Suppose, blocks C_{j1}, C_{j2}, ... are deleted from C , giving a reduced matrix C'. Recall that associated with each block C_i $(i = 1, 2, . . ,t+1)$ of C there is a $(n_i, \alpha_i, \alpha_i)$ code. Rearranging (6.1), we have,

$$k + \left(d - \sum_{j=j1,j2,..} \alpha_j\right) - 1 = \sum_{\substack{i=1 \\ i \neq j1,j2,..}}^{t+1} \alpha_i$$

i.e.

$$k + d' - 1 = \sum_{\substack{i=1 \\ i \neq j1,j2,..}}^{t+1} \alpha_i \tag{6.2}$$

where

$$d' = d - \sum_{j=j1,j2,..} \alpha_j$$

The length of the code with generator matrix C is given by

$$n = \sum_{i=1}^{t+1} M(\alpha_i) \tag{6.3}$$

Rearranging (6.3), we see

$$n - \sum_{j=j1,j2,..} M(\alpha_j) = \sum_{\substack{i=1 \\ i \neq j1,j2,..}}^{t+1} M(\alpha_i) \tag{6.4}$$

From (6.2), it is clear that we may obtain a code of dimension k and distance d', simply by excluding the computations $\Phi_j(\mathrm{u}) \equiv Z_j(u)Y_j(u)$ modulo $P_j(u), j = j1, j2,$ . . .

Moreover, from (6.4), it may be observed that the generator matrix C' of the reduced code is obtained by removing the blocks C_j ($j = j$ 1, $j2$,..) from C. (6.4) also gives the length n' of the new code.
i.e.

$$n' = n - \sum_{j=j\,1,j\,2,..} M\,(\alpha_j\,)$$

For a given value of d', the blocks C_j ($j = j1, j2,$ . . . .) are selected to minimise n', the length of the reduced code.

#

Corollary 6.1

Suppose $C = [C_1\, |C_2\, |. . . |C_{t+1}]$ is the generator matrix of a (n, k, d) code where each C_j corresponds to a computation of the form, $\Phi_j(u) \equiv Z_j(u)Y_j(u)$ modulo $P_j(u), j = 1, 2, . . ,t+1$. By adding new blocks C_i to C, where C_i corresponds to the computation $\Phi_i(u) \equiv Z_i(u)Y_i(u)$ modulo $P_i(u)$ (performed as described in the proof of Lemma 6.1), provided $\gcd(P_i(u), P_j(u)) = 1$, $j = 1, 2, . ., t$, we may obtain the generator matrix of an extended (n', k, d') code ($n' > n$ and $d' > d$).

Proof

Trivial - extension of Lemma 6.2.

#

Example 6.2.1

Consider the (14, 5, 5) code corresponding to the computation $\Phi(u) = Z(u)Y(u)$ where the polynomials are $P_1(u) = u$, $P_2(u) = (u+1)$, $P_3(u) = (u^2+u+1)$ and $P_4(u) = (u^3+u^2+1\,)$ and $s = 2$.

It can be shown that the generator matrix of this code is :

$$C = \begin{array}{c|c|ccc|cccccc|ccc}
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
\Uparrow & \Uparrow & & \Uparrow & & & & \Uparrow & & & & & \Uparrow & \\
C_1 & C_2 & & C_3 & & & & C_4 & & & & & C_5 &
\end{array}$$

where each block C_i corresponds to the computation $\Phi_i(u) \equiv Z_i(u)Y_i(u)$modulo $P_i(u)$ $i = 1, 2, 3, 4$. C_5 corresponds to the wraparound.

By dropping block C_3 from C, we obtain the generator matrix of the (11, 5, 3) code. Or, we can obtain the generator matrix of the (13, 5, 4) code by dropping either block C_1 or C_2. It is possible to obtain the generator matrices of a number of other codes by deleting other blocks or groups of blocks.

Example 6.2.2

Recall, in Example 5.1.1, we obtained the generator matrix of the (12, 6, 3) code by computing the aperiodic convolution $\Phi(u) = Z(u)Y(u)$. $P(u)$ was taken to be $P(u) = u^2(u^2+1)(u^2+u+1)$ and the computation involved two wraparounds.

$$C = \begin{array}{c} \left[\begin{array}{ccc|ccc|ccc|ccc} 1&0&1&1&0&1&1&0&1&0&0&0\\ 0&1&1&0&1&1&0&1&1&0&0&0\\ 0&0&0&1&0&1&1&1&0&0&0&0\\ 0&0&0&0&1&1&1&0&1&0&0&0\\ 0&0&0&1&0&1&0&1&1&0&1&1\\ 0&0&0&0&1&1&1&1&0&1&0&1 \end{array}\right] \\ \begin{array}{cccc} \Uparrow\quad\quad & \Uparrow\quad\quad & \Uparrow\quad\quad & \Uparrow \\ C_1\quad\quad & C_2\quad\quad & C_3\quad\quad & C_4 \end{array} \end{array}$$

We wish to illustrate Corallary 6.1. Let $P_4(u) = u^3+u+1$. Then $P_4(u)$ is coprime to $P_1(u)$, $P_2(u)$ and $P_3(u)$. Reducing $Z(u)$ and $Y(u)$ modulo $P_4(u)$ we obtain :

$$\begin{aligned} Z_4(u) &\equiv Z(u) \text{ modulo } (u^3+u+1) \\ &= (z_0+z_3+z_5) + (z_1+z_3+z_4+z_5)u \\ &\qquad + (z_2+z_4+z_5)u^2 \end{aligned}$$

and

$$\begin{aligned} Y_4 (u) &\equiv Y(u) \text{ modulo } (u^3+u+1) \\ &= y_0 + y_1u + y_2u^2. \end{aligned}$$

Let,

$$\begin{aligned} m_9 &= (z_0+z_3+z_5).\, y_0, \\ m_{10} &= (z_1+z_3+z_4+z_5).y_1, \\ m_{11} &= (z_2+z_4+z_5).y_2, \\ m_{12} &= (z_0+z_1+z_4).(y_0+y_1), \\ m_{13} &= (z_1+z_2+z_3).(y_1+ y_2) \end{aligned}$$

and

$$m_{14} = (z_0+z_2+z_3).(y_0+y_2).$$

It follows that C_5, the block corresponding to the computation $\Phi_4(u) \equiv Z_4(u)Y_4(u)$ modulo $P_4(u)$ is :

$$C_5 = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

By adding this block to the generator matrix of the (12, 6, 3) code , we obtain the generator matrix of (18, 6, 9), given by :

$$C' = \left[\begin{array}{ccc|ccc|ccc|ccc|cccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{array}\right]$$
$$\begin{array}{ccccc} \Uparrow & \Uparrow & \Uparrow & \Uparrow & \Uparrow \\ C_1 & C_2 & C_3 & C_4 & C_5 \end{array}$$

6.3- Error-Detection

An (n, k, d) linear code forms a vector space of dimension k, with a corresponding null-space of dimension $(n\text{-}k)$. This null-space is spanned by $(n\text{-}k)$ linearly independent equations.

Every codeword received is tested to determine whether or not it is a valid codeword. A received block is assumed error-free if it satisfies the $(n\text{-}k)$ parity check equations spanned by the null space. The only time errors in a received vector will remain undetected is when this erroneous codeword is identical to one of the non-zero codewords.

Below is a procedure which may be used to obtain these parity check equations for the KM (n, k) codes generated by the aperiodic convolution algorithm based on the CRT.

Consider $C = [C_1 \mid C_2 \mid ...... \mid C_{t+1}]$, the generator matrix of such a code. It has been noted that each C_i $(i = 1, 2, \ldots, t+1)$ has α_i columns which are linearly independent while the $[M(\alpha_i)\text{-}\alpha_i]$ remaining blocks are dependent on these. So each of the $(t+1)$ blocks gives rise to $[M(\alpha_i)\text{-}\alpha_i]$ parity check equations. For simplification, it is assumed that $\alpha_i < k$ $(i = 1, 2, \ldots, t+1)$. So, on receiving a vector $\boldsymbol{r}$, it can be divided into $t+1$ segments i.e. $\boldsymbol{r} = (r_1, r_2, \ldots, r_{t+1})$ where the segment r_i corresponds to the block C_i. If r_i is error-free, it should satisfy the $[M(\alpha_i)\text{-}\alpha_i]$ parity check equations associated with C_i. We shall illustrate how to obtain these parity check equations shortly (See Example 6.3.1).

For each block C_i, we have $[M(\alpha_i)-\alpha_i]$ corresponding parity check equations. Further, each set of $[M(\alpha_i)-\alpha_i]$ equations is linearly independent of the t other sets of $[M(\alpha_j)-\alpha_j]$ equations, $j = 1, \ldots, t+1, j \neq i$. Hence the total number of linearly independent parity check equations obtained so far is given by ,

$$\begin{aligned}\sum_{i=1}^{t+1}(M(\alpha_i)-\alpha_i) &= \sum_{i=1}^{t+1} M(\alpha_i) - \sum_{i=1}^{t+1}\alpha_i\\ &= n - N\\ &= n - (k+d-1)\\ &= (n-k)-(d-1)\end{aligned}$$

and so we require a further $(d\text{-}1)$ equations which must be linearly independent of the $(n\text{-}N)$ equations already obtained.

Consider blocks $C_1, C_2, \ldots, C_t$. Assume that the columns of these blocks are arranged so that the first α_i columns correspond to the polynomial

$$Z_i(u) \equiv Z(u) \text{ modulo } P_i(u) \quad i = 1, 2, \ldots, t.$$

To derive a further $(D\text{-}k)$ parity check equations we are interested only in these columns. Using the CRT, the polynomial $Z(u)$ may be recovered from the residue polynomials $Z_i(u)$ $(i = 1, 2, \ldots, t)$. If the polynomial $P(u) = \prod_{i=1}^{t} P_i(u)$ is of degree D, then the reconstructed polynomial $Zr(u)$ will have degree $(D\text{-}1)$.
i.e.

$$Zr(u) = z_0+z_1u+\ldots\ldots+z_{D-1}u^{D-1}.$$

However, the information polynomial is always of degree $(k\text{-}1)$. In order that the information polynomial and the reconstructed polynomial match, the last $[(D\text{-}1)\text{-}(k\text{-}1)] = D\text{-}k$ coefficients of the reconstructed polynomial must be zero.
i.e.

$$z_i \equiv 0 \quad i = k, k+1, \ldots, D\text{-}1. \tag{6.5}$$

This provides $(D\text{-}k)$ more parity check equations.

In the final block C_{t+1}, the first $\alpha_{t+1} = s$ columns are associated with the polynomial $\overline{Z}(u)$ modulo u^s. Provided no errors have occured, the corresponding coefficients of $\overline{Z}(u)$ and $Z(u)$ should be the same, thus leading to another s parity check equations.

$$\overline{z}_i \equiv z_i \qquad i = k-1, k-2, \ldots, k-s \tag{6.6}$$

Together (6.5) and (6.6) give $(D\text{-}k)+s = (D+s)\text{-}k$ equations. However, $(D+s) = N = (k+d\text{-}1)$ and so $(D+s)\text{-}k = (k+d\text{-}1)\text{-}k = (d\text{-}1)$. Thus we have found the remaining $(d\text{-}1)$ parity check equations. It should be noted that these equations are linearly independent since no coefficient of a polynomial can be expressed as a linear sum of the other coefficients.

Example 6.3.1

Consider the (10, 5, 3) code. We have $k = 5$ and $d = 3$ and so $N = k+d-1 = 7$. In example 5.3(a), we found the polynomials are $Z(u) = z_0+z_1u+z_2u^2+z_3u^3+z_4u^4$ and $Y(u) = y_0+y_1u+y_2u^2$. Also $P(u) = u^2(u^2+1)(u^2+u+1) = P_1(u)P_2(u)P_3(u)$ and s, the number of wraparound points equals 1. By reducing $Z(u)$ and $Y(u)$ modulo $P_i(u)$ $i = 1, 2, 3$ and then computing $\Phi_i(u) \equiv Z_i(u)Y_i(u)$ modulo $P_i(u)$, we found the multiplicative complexity to be 10, where

$$\begin{aligned}
m_0 &= z_0.y_0,\\
m_1 &= z_1.y_1,\\
m_2 &= (z_0+z_1).(y_0+y_1),\\
m_3 &= (z_0+z_2+z_4).(y_0+y_2),\\
m_4 &= (z_1+z_3).y_1,\\
m_5 &= (z_0+z_1+z_2+z_3+z_4).(y_0+y_1+y_2),\\
m_6 &= (z_0+z_2+z_3).(y_0+y_2),\\
m_7 &= (z_1+z_2+z_4).(y_1+y_2),\\
m_8 &= (z_0+z_1+z_3+z_4).(y_0+y_1)
\end{aligned}$$

and

$$m_9 = z_4.y_2.$$

Further, the generator matrix C of the corresponding (10, 5, 3) code is

$$C = \begin{array}{c} \left[\begin{array}{ccc|ccc|ccc|c}
1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0\\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0\\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0\\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0\\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1
\end{array}\right] \\
\begin{array}{cccc} \Uparrow \quad\quad & \Uparrow \quad\quad & \Uparrow \quad & \Uparrow \\ C_1 \quad\quad & C_2 \quad\quad & C_3 \quad & C_4 \end{array}
\end{array}$$

We stated above that each block C_i will contribute $[M(\alpha_i)-\alpha_i]$ parity check equations. So blocks C_1, C_2, C_3 each provide one equation ((6.7) below) while the fourth block provides no equations. Let c_i denote the i th digit of the code vector $c = (c_0, c_1, . . ., c_9)$. Then the equations are :

$$\begin{array}{ll}
\text{For } P_1(u) & c_0+c_1+c_2 = 0\\
\text{For } P_2(u) & c_3+c_4+c_5 = 0\\
\text{For } P_3(u) & c_6+c_7+c_8 = 0
\end{array} \qquad (6.7)$$

Further, the first α_i columns of each block C_i, correspond to $Z_i(u) \equiv Z(u)$ modulo $P_i(u)$. Then, in terms of $c_0, \ldots, c_9$, we have $Z_1(u) \equiv c_0+c_1u$ modulo u^2, $Z_2(u) \equiv c_3+c_4u$ modulo (u^2+1) and $Z_3(u) \equiv c_6+c_7u$ modulo (u^2+u+1). The polynomial $Zr(u) = z_0+z_1u+z_2u^2+z_3u^3+z_4u^4+z_5u^5$ may be reconstructed from these congruences $Z_i(u) \equiv Z(u)$ modulo $P_i(u)$ using the CRT.
i.e.

$$Zr(u) \equiv \sum_{i=1}^{3} S_i(u) Z_i(u) \text{ modulo } P(u)$$

The polynomials $S_i(u)$ $i=1, 2, 3$ are found as follows.

$i = 1$

$$R_1(u)P_2(u)P_3(u) \equiv 1 \text{ modulo } P_1(u)$$
$$\text{i.e. } R_1(u)(u^2+1)(u^2+u+1) \equiv 1 \text{ modulo } u^2$$
$$\text{i.e. } R_1(u)(u^4+u^3+u+1) \equiv 1 \text{ modulo } u^2$$

Hence $R_1(u) = (u+1)$ and $S_1(u) = 1+u^2+u^3+u^5$.

$i = 2$

$$R_2(u)P_1(u)P_3(u) \equiv 1 \text{ modulo } P_2(u)$$
$$\text{i.e. } R_2(u)u^2(u^2+u+1) \equiv 1 \text{ modulo } (u^2+1)$$
$$\text{i.e. } R_2(u)(u^4+u^3+u^2) \equiv 1 \text{ modulo } (u^2+1)$$

Hence $R_2(u) = u$ and $S_2(u) = u^3+u^4+u^5$.

$i = 3$

$$R_3(u)P_1(u)P_2(u) \equiv 1 \text{ modulo } P_3(u)$$
$$\text{i.e. } R_3(u)u^2(u^2+1) \equiv 1 \text{ modulo } (u^2+u+1)$$
$$\text{i.e. } R_3(u)(u^4+u^2) \equiv 1 \text{ modulo } (u^2+u+1)$$

Hence $R_3(u) = 1$ and $S_3(u) = u^4+u^2$.

It follows that,

$$Zr(u) \equiv (1+u^2+u^3+u^5)(c_0+c_1u) + (u^3+u^4+u^5)(c_3+c_4u) + (u^2+u^4)(c_6+c_7u) \text{ modulo } P(u)$$

$$\equiv c_0 + c_1u + (c_0+c_6)u^2 + (c_0+c_1+c_3+c_7)u^3 + (c_1+c_3+c_4+c_6)u^4 + (c_0+c_3+c_4+c_7)u^5 + (c_1+c_4)u^6 \text{ modulo } (u^2+u^3+u^5+u^6)$$

$$= c_0 + c_1u + (c_0+c_1+c_4+c_6)u^2 + (c_0+c_3+c_4+c_7)u^3 + (c_1+c_3+c_4+c_6)u^4 + (c_0+c_1+c_3+c_7)u^5.$$

Then $Zr(u) = z_0+z_1u+z_2u^2+z_3u^3+z_4u^4+z_5u^5$
where

$$z_0 = c_0,$$
$$z_1 = c_1,$$
$$z_2 = c_0+c_1+c_4+c_6,$$
$$z_3 = c_0+c_3+c_4+c_7,$$
$$z_4 = c_1+c_3+c_4+c_6$$

and

$$z_5 = c_0+c_1+c_3+c_7.$$

For this reconstructed polynomial of degree 5, to correspond to the original polynomial, of only degree 4 we must have $z_5 = 0$.
i.e.

$$c_0+c_1+c_3+c_7 = 0 \tag{6.8}$$

Further, if $Z(u) = z_0+z_1u+z_2u^2+z_3u^3+z_4u^4$, then $\overline{Z}(u) = z_4+z_3u+z_2u^2+z_1u^3+z_0u^4$. The wraparound corresponds to the coefficient z_4 and therefore $z_4+c_9 = 0$.
i.e.

$$c_1+c_3+c_4+c_6+c_9 = 0 \tag{6.9}$$

In total, we have found 5 parity check equations, (6.7), (6.8), (6.9). Since n-k = 10-5 = 5, this is the complete set of parity check equations.

6.4 - Parity Check Equations of Families of Codes where $N = k + d - 1$ = constant

Suppose C, the generator matrix of an (n, k, d) code is altered to C', the generator matrix of an (n, k', d') as described in Section 5.2, keeping $k+d-1 = N$ and thus n (the code length) constant. The parity check equations obtained from (6.6) are altered and if

(i) $k < k'$ then $(k'-k)$ equations are dropped from (6.5)

while if

(ii) $k > k'$ then $k-k'$ equations are added to (6.5).

The remaining $(n-N)$ equations remain unchanged [2].

Example 6.4.1

From, C, the generator matrix of the (10, 5, 3) code obtained in example 5.3(a), we may obtain C', the generator matrix of a (10, 3, 5) code by appropriately shortening the columns of C.

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

Since $N = k+d-1 = 7$, for both of these codes, n is fixed and so the parity check equations (6.7) obtained in Example 6.3.1, also hold for this example. However, we require $(n-k)=10-3=7$ check equations in total and so we must now seek 4 more equations.

Now, since $k = 3$, $Z(u) = z_0+z_1u+z_2u^2$. However, the reconstructed polynomial $Zr(u) = z_0+z_1u+z_2u^2+z_3u^3+z_4u^4+z_5u^5$ is of degree $(D-1) = 5$ and so $z_3= z_4= z_5 = 0$.
i.e.

$$\begin{aligned} c_0+c_3+c_4+c_7&= 0\\ c_1+c_3+c_4+c_6&= 0\\ c_0+c_1+c_3+c_7&= 0 \end{aligned} \tag{6.10}$$

Finally,

$$Z(u) = z_0+z_1u+z_2u^2 \text{ and } \overline{Z}(u\ ) = z_2+z_1u+z_0u^2 .$$

The wraparound corresponds to the coefficient z_2 and therefore $z_2+c_9 = 0$.
i.e.

$$c_0+c_1+c_4+c_6+c_9 = 0 \tag{6.11}$$

Hence, (6.10) and (6.11) provide the 4 remaining parity check equations. Any codeword must satisfy equations (6.7), (6.10) and (6.11).

6.5 - Burst Error-Detection Capability

We now wish to establish the burst error-detection capability of these codes.

Lemma 6.3

For any information vector, the sum of the degrees α_i of the polynomials $P_i(u)$ such that the associated blocks of the codevector are nonzero cannot be less than d [2].

Proof

The information vector is expressed as $Z(u)$, a polynomial of degree $(k-1)$.
Consider

$$Z_i(u) \equiv Z(u) \text{ modulo } P_i(u)$$

Clearly,

$$Z_i(u) = 0 \text{ iff } Z(u) = hP_i(u) \quad \text{for some integer } h.$$

Hence, if $P_i(u)$ divides $Z(u)$, then the block C_i corresponding to the computation $\Phi_i(u) \equiv Z_i(u)Y_i(u)$ modulo $P_i(u)$ is zero.

Suppose that the blocks $C_{j1}, C_{j2}, \ldots, C_{jm}$ are all zero. Then the blocks of a codevector corresponding to $C_{j1}, C_{j2}, \ldots, C_{jm}$ will be zero.
Further, we have

$$P_{j1}(u) \mid Z(u) \qquad \deg[P_{j1}(u)] = \alpha_{j1}$$

and

$$P_{j2}(u) \mid Z(u) \qquad \deg[P_{j2}(u)] = \alpha_{j2} .$$

Since $\gcd(P_{j1}(u), P_{j2}(u)) = 1$, then it follows that

$$(P_{j1}(u)\, P_{j2}(u)) \mid Z(u) \quad \text{and} \quad \deg[\, P_{j1}(u)\, P_{j2}(u)] = \alpha_{j1}+\alpha_{j2}.$$

Hence the degree of $(P_{j1}(u)\, P_{j2}(u))$ cannot be greater than the degree of $Z(u)$.
i.e.

$$\alpha_{j1} + \alpha_{j2} \le (k-1)$$

Continuing the above argument, it is clear that since $(P_{j1}(u)P_{j2}(u).\,.P_{jm}(u))$ divides $Z(u)$, we must have

$$\sum_{j=j1,j2,\ldots,jm} \alpha_j \le (k-1) \tag{6.12}$$

Recall,

$$\sum_{i=1}^{t+1} \alpha_i = k + d - 1$$

So that,

$$\sum_{\substack{i=1 \\ i \ne j1,j2,..jm}}^{t+1} \alpha_i + \sum_{j=j1,j2,\ldots,jm} \alpha_j = d + (k-1)$$

i.e.

$$\sum_{\substack{i=1 \\ i \ne j1,j2,..jm}}^{t+1} \alpha_i = d + (k-1) - \sum_{j=j1,j2,\ldots,jm} \alpha_j$$

By (6.12), it follows that

$$\sum_{\substack{i=1 \\ i \ne j1,j2,..jm}}^{t+1} \alpha_i \ge d + (k-1) - (k-1)$$

i.e.

$$\sum_{\substack{i=1 \\ i \ne j1,j2,..jm}}^{t+1} \alpha_i \ge d$$

Hence result.

#

This lemma, together with the assumption that the computations $\Phi_i(u) \equiv Z_i(u)Y_i(u)$ modulo $P_i(u)$ are performed in such a way that each block C_i is an $(n_i, \alpha_i, \alpha_i)$ code may be used to determine the burst error-detection capability of the code [2] .

6.6 - Error-Correction

We saw previously that if the polynomial $P_i(u)$ is irreducible then the computation $\Phi_i(u) \equiv Z_i(u)Y_i(u)$ modulo $P_i(u)$ is performed in 2 steps

(i) $\Phi_i'(u) = Z_i(u)Y_i(u)$

(ii) $\Phi_i(u) \equiv \Phi_i'(u)$ modulo $P_i(u)$.

However, if the polynomial $P_i(u)$ takes the form $P_i(u) = u^{\alpha_i}$ or $P_i(u) = (u+a_i)^{\alpha_i}$, the above procedure is not usually employed. As is true for the wraparound computation.

For simplicity, we shall assume that all computations $Z_i(u)Y_i(u)$ modulo $P_i(u)$ are performed using steps (i) and (ii) above. (As noted earlier, for $\alpha_i \leq 2$, this does not result in an increased multiplicative complexity, while for $\alpha_i > 2$ there is an increase but it is small and dependent on the field of computation.)

This assumption, together with Lemma 6.1 means that each block C_i is an $(n_i, \alpha_i, \alpha_i)$ code where $\alpha_i = \deg[P_i(u)]$, $\alpha_{t+1} = s$ and $n_i = M(\alpha_i)$. Each such code is capable of correcting up to $[(\alpha_i\text{-}1)/2]$ errors ($i = 1, 2, \ldots, t+1$).

Below, is an error-correcting procedure which Krishna and Morgera devised for KM codes. However, we do not believe it to be correct.

To decode a received vector $\boldsymbol{r}$, the following was suggested [2]:

Partition vector $\boldsymbol{r}$ into $(t+1)$ sub-vectors i.e. $\boldsymbol{r} = (\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_{t+1})$. The sub-vector $\boldsymbol{r}_i$ corresponding to C_i is then decoded independently. Let $ZD_i(u)$ denote the decoded vector corresponding to the block C_i. There are two possibilities of this decoding :

(a) When up to $[(\alpha_i\text{-}1)/2]$ errors are present in $\boldsymbol{r}_i$, this is within the error-correcting capability of the $(n_i, \alpha_i, \alpha_i)$ code associated with block C_i. Hence decoding is successful.
i.e.

$$ZD_i(u) = Z_i(u)$$

(b) If more than $[(\alpha_i\text{-}1)/2]$ errors occur in $\boldsymbol{r}_i$ during transmission, then the code associated with C_i is unable to recover $Z_i(u)$. A decoding failure takes place.
i.e.

$$ZD_i(u) \neq Z_i(u)$$

If $Z_i(u)$ is decoded erroneously, the corresponding block C_i is eliminated from any further analysis. So block C_i is removed from C to give a modified matrix C'.

i.e.

$$C' = [C_1 \mid C_2 \mid \ldots \mid C_{i-1} \mid C_{i+1} \mid \ldots \mid C_{t+1}]$$

By Lemma 6.2, C' is the generator matrix of a $(n\text{-}M(\alpha_i), k, d\text{-}\alpha_i)$ code which may correct up to $[((d\text{-}\alpha_i)\text{-}1)/2]$ errors. However, as block C_i has been excluded, we have effectively removed at least $[(\alpha_i\text{-}1)/2]+1$ errors. Therefore, if $Z(u)$ is recovered using C', the maximum number of errors that can be corrected in the received vector $\boldsymbol{r}$ is $[(d\text{-}1)/2]$.

So, by eliminating the block C_i, we have not restricted the error-correcting capability of our decoding procedure.

What they failed to observe is that when a decoding failure (condition (b)) occurs, the code is not aware of it. For example, suppose x and y are code vectors of an (n, k, d) code differing from each other in d places. Further, suppose x^, the received word corresponding to x contains d-1 errors and x^+(1,0,. . ,0)=y. Then since x^ and y differ in only one digit, x^ will be decoded to y. The code would assume that correct decoding had taken place when in fact it had not. Therefore, it is not possible to eliminate erroneous blocks.

The error-correcting procedure Krishna and Morgera describe is based on Theorem 6.1, below [2].

We are able to recover $Z(u)$ from the received vector $\boldsymbol{r}$, provided no more than $[(d\text{-}1)/2]$ errors have occurred. The information polynomial $Z(u)$, being a polynomial of degree $(k\text{-}1)$, can be recovered, using the CRT, from any set of residues of the type $Z_h(u) \equiv Z(u)$ modulo $P_h(u)$, $h=h_1, h_2, ..$ provided that $\sum\ \alpha_h \geq k$, $h = h_1, h_2, ..$

Let σ_i = no. of errors in $\boldsymbol{r}_i$ (- the received vector corresponding to C_i.) Then it is possible to recover $Z(u)$ from $\boldsymbol{r}$ provided :

$$\sum_{i=1}^{t+1} \sigma_i \leq \left[\frac{(d-1)}{2}\right].$$

So, the two possible outcomes of the decoding of $\boldsymbol{r}_i$ may be rewritten as :

(a) $\sigma_i \leq [(\alpha_i\text{-}1)/2]$.
(b) $\sigma_i > [(\alpha_i\text{-}1)/2]$.

Theorem 6.1

Let d be the minimum distance of a KM code with generator matrix $C = [C_1 | C_2 | . . | C_{t+1}]$. Provided no more than $[(d\text{-}1)/2]$ errors are present in a code vector $\boldsymbol{r} = (\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_{t+1})$, then after each subvector $\boldsymbol{r}_i$ is decoded according to its minimum distance α_i, there exists at least one set of error-free subvectors such that the sum of the degrees of the polynomials $P_j(u)$ corresponding to these blocks is at least k [2] .

Proof

Recall, σ_i is the number of errors in the received sub-vector r_i. Suppose that condition (b) above holds for $\sigma_{i1}, \sigma_{i2}, \ldots, \sigma_{if}$. Then the corresponding blocks $C_{i1}, C_{i2}, \ldots, C_{if}$ all suffer decoding failure and the smallest value each such σ_i may take is $[\alpha_i - 1/2]+1 (i = i1, i2, \ldots, if)$. Hence, in these circumstances, the least number of errors in received vector $\boldsymbol{r}$ (assuming all the other subvectors to be completely error-free) is :

$$\min\left\{\sum_{i=i1,i2,\ldots,if} \sigma_i\right\} = \sum_{i=i1,i2,\ldots,if}\left\{\left[\frac{\alpha_i - 1}{2}\right] + 1\right\}$$

The decoder is able to correct these errors provided it is within the error-correcting capability of the reduced code which we have already shown to be [(d-1)/2].
i.e. we require the following condition to hold :

$$\left[\frac{(d-1)}{2}\right] \geq \sum_{i=i_1,i_2,\ldots,i_f} \left\{\left[\frac{\alpha_i - 1}{2}\right] + 1\right\}$$

i.e.

$$\frac{d}{2} > \sum_{i=i_1,i_2,\ldots,i_f} \frac{\alpha_i}{2}$$

i.e.

$$d > \sum_{i=i_1,i_2,\ldots,i_f} \alpha_i \qquad (6.13)$$

Recall,

$$N = \deg\left[P\ (u\ )\right] + s = \sum_{j=1}^{t+1} \alpha_j$$

It follows

$$\sum_{i=i_1,i_2,\ldots,i_f} \alpha_i + \sum_{\substack{j=1 \\ j \neq i_1,i_2,\ldots,i_f}}^{t+1} \alpha_j = N$$

and

$$\sum_{\substack{j=1 \\ j \neq i_1,i_2,\ldots,i_f}}^{t+1} \alpha_j = N - \sum_{i=i_1,i_2,\ldots,i_f} \alpha_i$$

$$= (k + d - 1) - \sum_{i=i_1,i_2,\ldots,i_f} \alpha_i$$

$$> (k + d - 1) - d \qquad \text{by (6.13)}$$

$$= k - 1.$$

i.e.

$$\sum_{\substack{j=1 \\ j \neq i_1,i_2,\ldots,i_f}}^{t+1} \alpha_j \geq k$$

We have shown that the sum of the degrees of all the polynomials $P_j(u)$ corresponding to correctly decoded blocks C_j ($j = 1, 2, . . ., t+1, j \neq i1, i2, . . .,if$) is at least k.

#

Upon decoding, using a KM code of minimum distance d, we will not be aware of the actual number of errors which were present nor will we be aware which blocks have been decoded erroneously. We will look at the validity of Theorem 6.1 when it is guaranteed that no more than $[(d-1)/2]$ errors are present in the received vector $\boldsymbol{r}$ and when the erroneous blocks remain. Since the error-correcting capabilty of the full KM code is $[(d-1)/2]$, the basis of the proof of Theorem 6.1 still holds under these new conditions. Hence, it is true that after decoding, there is always at least one set of error-free residues of the form $Z_h(u) \equiv Z(u)$ modulo $P_h(u)$, $h = h_1, h_2, ...$ where $\sum \alpha_h \geq k$, $h = h_1, h_2, . . .$ However it is not possible to determine which residues are error-free.

For the remainder of Section 6.6 and Section 6.7 , let us assume that there is some way of determing which residues are error-free.

Let $I = \{1, 2, ..., t+1\}$. The integer i in the set I corresponds to the polynomial $P_i(u)$ of degree α_i for $i = 1, 2, . . ., t$, while the integer $(t+1)$ corresponds to the wraparound. From I, we form subsets $I_1, I_2, ...$ such that each subset is the minimal set with respect to the property that the sum of the powers of the polynomials corresponding to the integers in each subset is at least k.

Let k_i be the sum associated with subset I_i ($i = 1, 2, ...$). Then associated with each subset I_i, we have the set of residues

$$Z_h(u) \equiv Z(u) \text{ modulo } P_h(u) \qquad h \in I_i.$$

Since $k_i \geq k$, we may use the CRT to reconstruct a polynomial $Z_i^*(u) = z^*_{i,0} + z^*_{i,1}u + ... + z^*_{i,k_i-1}u^{k_i-1}$ of degree (k_i-1) from the residue polynomials $Z_h(u)$ where $h \in I_i$. However, the original information polynomial $Z(u) = z_0+z_1u+. . . . +z_{k-1}u^{k-1}$ is only of degree $(k-1)$. For $Z_i^*(u)$ to be considered a candidate for the information polynomial $Z(u)$, $Z_i^*(u)$ must satisfy the following (k_i-k) equations :

$$z^*_{i,h} = 0 \qquad h = k, k+1, ..., k_i-1$$

Suppose $Z_1^*(u), Z_2^*(u),. .$ are candidates for the information polynomial. Since the code is of minimum distance d, the code vector corresponding to a valid information polynomial will differ from the received vector in at most $[(d-1)/2]$ places. Let $CV_{(i)}$ denote the code vector corresponding to the candidate polynomial $Z_i^*(u)$. Thus if $CV_{(i)}$ differs from the received vector $\boldsymbol{r}$, in no more than $[(d-1)/2]$ places, then $Z_i^*(u)$ is accepted as a valid information polynomial. Theorem 6.1 guarantees the existence of at least one candidate polynomial which is a valid information polynomial.

To summarise, the complete decoding algorithm for these codes is [2] :

(1) Partition the received vector $\boldsymbol{r}$ into $(t+1)$ segments,

$$\boldsymbol{r} = (\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_{t+1})$$

where sub-vector $\boldsymbol{r}_i$ corresponds to block C_i of the generator matrix C , i =1, 2, ... , t+ 1

(2) Perform decoding on $\boldsymbol{r}_i$ using the $(n_i, \alpha_i, \alpha_i)$ code associated with block C_i , i = 1, 2, ..., t+1.

(3) Eliminate the blocks for which the decoding fails.

(4) Use the CRT, to construct the candidates $Z_1*(u)$, $Z_2*(u)$,. . . . from the residue polynomials obtained from the blocks known to be error free by the previous step.

(5) Find the candidate code vectors $CV_{(i)}$ for each of the candidate information polynomials $Z_i*(u)$.

(6) Accept $CV_{(i)}$ as a valid code vector if it differs from the received vector $\boldsymbol{r}$ in no more than $[(d\text{-}1)/2]$ places.

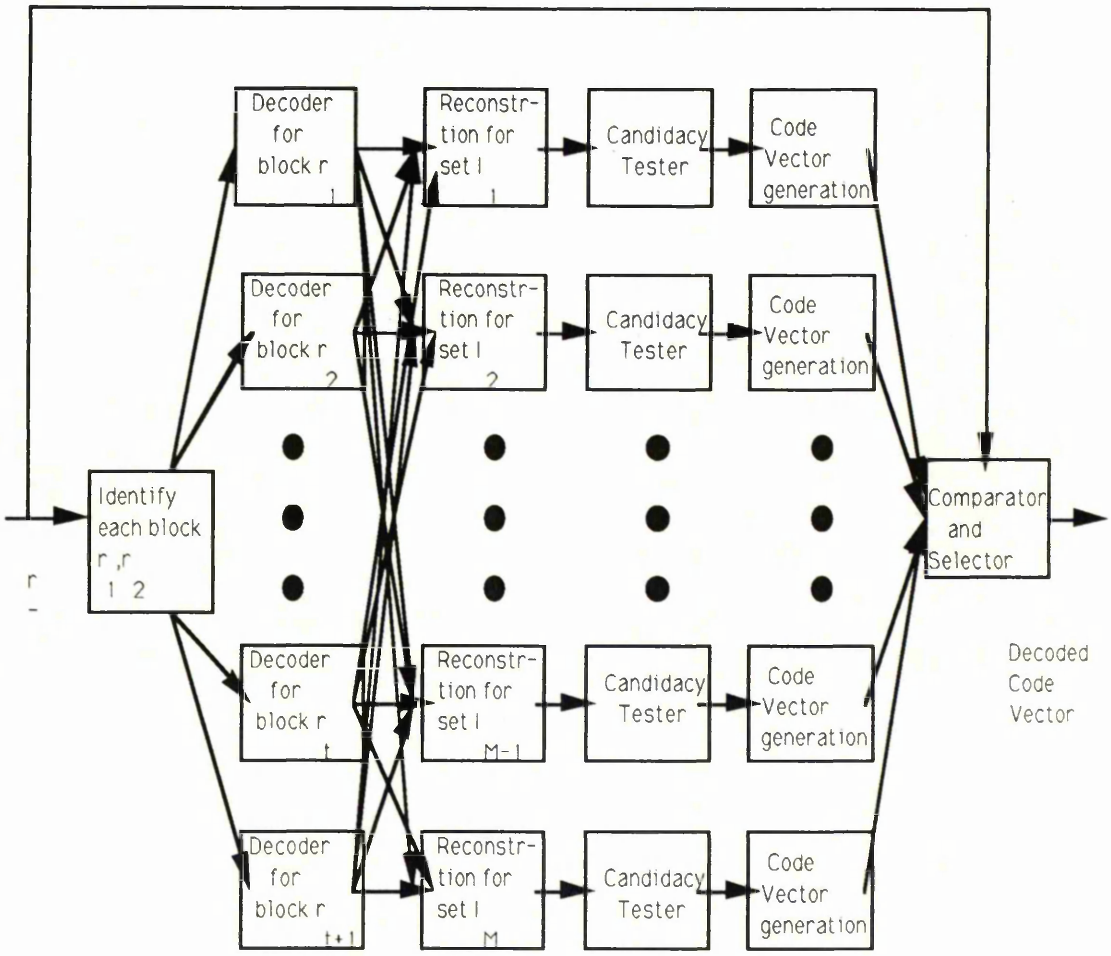


Figure 6.4 - Block Diagram of decoding algorithm.

Example 6.6.1

Again, consider the (10, 5, 3) code. $P(u)$ is taken as $u^2(u^2+1)(u^2+u+1)$ and $s = 1$. The received vector $\boldsymbol{r}$ is partitioned into 4 subvectors $\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4$ corresponding to $P_1(u)$, $P_2(u)$, $P_3(u)$ and the wraparound respectively.

Recall, the generator matrix of this code is given by :

$$C=\begin{array}{c}\left[\begin{array}{ccc|ccc|ccc|c} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0\\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0\\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0\\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \end{array}\right]\\ \begin{array}{cccccccccc} & \Uparrow & & & \Uparrow & & & \Uparrow & & \Uparrow\\ & C_1 & & & C_2 & & & C_3 & & C_4 \end{array}\end{array}$$

By Lemma 6.1, each block C_i is a $(n_i, \alpha_i, \alpha_i)$ code where $n_i = M(\alpha_i)$. Hence block C_4 is a (1, 1, 1) code for which no decoder is required. While block C_1, C_2, C_3 being (3, 2, 2) codes require decoders. The set I is given by :

$$I = \{1, 2, 3, 4\}$$

and the possible subsets are $I_1 = \{1, 2, 4\}$, $I_2 = \{1, 3, 4\}$, $I_3 = \{2, 3, 4\}$ and $I_4 = \{1, 2, 3\}$.

6.6.2 - Complexity of Decoding

We wish to know just how easy it is to implement the decoding algorithm described above.

We shall begin by considering the implementation of step (4) of the decoding algorithm - the reconstruction for each set I_i using the CRT. Associated with I_i, we have the residues :

$$Z_h(u) \equiv Z(u) \text{ modulo } P_h(u) \qquad h \in I_i$$

and using the CRT, these residues are combined to reconstruct the polynomial $Z_i^*(u)$. i.e.

$$Z_i{}^*(u) \equiv \sum_{h \in I_i} S_h(u) Z_h(u) \text{ modulo } P_i{}^*(u)$$

where

$$P_i{}^*(u) = \prod_{h \in I_i} P_h(u)$$

Knowing the polynomials $P_h(u)$, $h \in I_i$, the polynomial $S_h(u)$ may be determined in advance and so the above expression for $Z_i{}^*(u)$ may be obtained by :

(i) Compute $S_h(u)Z_h(u)$, $h \in I_i$

(ii) Find the sum $\sum_{h \in I_i} S_h(u)Z_h(u)$

(iii) Reduce the polynomial obtained in (ii) modulo $P_i{}^*(u)$

Steps (i) and (iii) can be implemented by a multiplicative circuit and a division circuit respectively. Step (ii) may be implemented using a series of adders, one for each coefficient of the sum. Hence, step(4) of the decoding algorithm is implemented relatively easily.

Step(2) requires $(t+1)$ decoders, one for each subcode $(n_i, \alpha_i, \alpha_i)$, $i = 1, 2, \ldots, t+1$. Step(5), forming candidate codewords from candidate information polynomials, may be performed using a multiplication circuit. Step(6) requires a comparator followed by a counter and a threshold detector set at $[(d-1)/2]+1$.

Suppose k is altered to k', keeping n, the length of the code constant. This will result in d being changed to d'. Clearly steps (1), (2), (3) of the decoding algorithm will remain unaffected. The circuits described above to implement step (4) will have to be suitably adjusted. Since a change in k does not mean a great change in the encoding circuit, alterations to step (5) will be minimal. Finally, the threshold of the comparator used in step (6) will have to be reset at $[(d'-1)/2]$, the error correcting capability of the new (n, k', d') code.

Hence, as expected, the decoder is not significantly altered by a change in k.

Example 6.6.2

Consider the (14, 5, 5) KM code where $P(u) = u(u^2+1)(u^2+u+1)(u^3+u^2+1)$ $P_1(u)P_2(u)P_3(u)P_4(u)$ and $s = 1$. The generator matrix C has 5 subblocks, C_1, C_2, C_3, C_4 and C_5 corresponding to $P_1(u)$, $P_2(u)$, $P_3(u)$, $P_4(u)$ and the wraparound respectively. Subblocks C_2, C_3 are (3, 2, 2) codes, C_1, C_5 are (1, 1, 1) codes and C_4 is a (6,3,3) code. Decoders are required for the codes corresponding to C_2, C_3 and C_4. The set I is given by

$$I = \{1, 2, 3, 4, 5\}$$

and there are 10 possible subsets, namely, $I_1 = \{1, 2, 3\}$, $I_2 = \{2, 3, 5\}$, $I_3 = \{3, 4\}$, $I_4 = \{2, 4\}$, $I_5 = \{1, 4, 5\}$, $I_6 = \{1, 2, 4\}$, $I_7 = \{1, 3, 4\}$, $I_8 = \{2, 4, 5\}$, $I_9 = \{3, 4, 5\}$, $I_{10} = \{2, 3, 4\}$.

Since the sum of the degrees of the polynomials associated with I_i, $i = 1, \ldots, 5$ is 5, the reconstructed polynomials $Z_i{}^*(u)$, $i = 1, \ldots, 5$ being of degree 4, are candidate information polynomials. However, $Z_j{}^*(u)$, $j = 6, \ldots, 9$ have degree 5 and will only be accepted as candidate information polynomials if

$$z_{j,5}{}^* = 0.$$

Further, $Z_{10}^*(u)$ has degree 6 and will only be considered as a candidate if

$$z_{10,5}^* = z_{10,6}^* = 0.$$

Hence a candidacy tester is required for $I_6, \ldots, I_{10}$. A code generator is required to obtain the candidate codevectors corresponding to the candidate information polynomials. Finally, a comparator accepts a candidate code vector as a valid code vector if it differs from the received vector in at most 2 places. The comparator is a set of 14 exclusive-OR gates followed by a counter and a threshold detector set at 3.

From the (14, 5, 5) code we may obtain (14, 3, 7) code. We shall see how its decoding requirements differ from that of the (14, 5, 5) code. The generator matrix of the (14, 3, 7) code also has 5 subblocks; C_2, C_3 are (3, 2, 2) codes, C_1 and C_5 are (1, 1, 1) codes and C_4 is a (6, 3, 3) code. Thus the decoders for these subblocks are essentially the same as those above. In this case there are 9 possible subsets given by, $I_1 = \{4\}$, $I_2 = \{1, 2\}$, $I_3 = \{1, 3\}$, $I_4 = \{2, 5\}$, $I_5 = \{3, 5\}$, $I_6 = \{2, 3\}$, $I_7 = \{4, 5\}$, $I_8 = \{2, 4\}$ and $I_9 = \{3, 4\}$. Only subsets I_6, I_7, I_8 and I_9 require a candidacy test. The design of the code vector generator and the comparator is as before. However, the threshold is now set to 4.

6.7 - Decoding of Families of Codes where k is Fixed

By Lemma 6.2, the blocks $\{C_j\} = \{C_j, j = i_1, i_2, \ldots\}$ may be deleted from C, the generator matrix of an (n, k, d) code to obtain C', the generator matrix of the reduced code (n', k, d'). We wish to see how the decoder of an (n, k, d) code may be modified to obtain the decoder of an (n', k, d') code. Let $\{P_j(u)\}$ denote the set of polynomials corresponding to the blocks $\{C_j\}$.

Since all the blocks in C' were present in C, steps (1), (2) and (3) of the decoding algorithm remain essentially unaltered - the decoders corresponding to the blocks $\{C_j\}$ are simply disabled.

In step (4), the reconstruction corresponding to the subset I_i is not performed if this subset involves any residue associated with $\{P_j(u)\}$. The other subsets undergo reconstruction as expected.

Outputs of the shift register used in step (5), corresponding to the blocks $\{C_j\}$, are disabled. Hence codewords of length n' are obtained.

Finally, in step (6), the threshold of the comparator is adjusted to $[(d'-1)/2]$, the error-correcting capability of the new code.

So, in summary, the decoder of the (n', k, d') code is obtained from the decoder of the (n, k, d) code simply by disabling any part of the original decoder associated with $\{C_j\}$. Clearly, as you would expect, the decoder structure of the (n', k, d') code is simpler than that of the (n, k, d) code.

We can obtain a (n', k, d') code from an (n, k, d) code where $n' > n$ and $d' > d$ by adding groups of columns to C, the generator matrix of the (n, k, d) code. Each group of columns will correspond to a computation $\Phi_j(u) \equiv Z_j(u)Y_j(u)$ modulo $P_j(u)$ where $P_j(u)$ is relatively prime to $P_i(u)$ $i = 1, 2, \ldots, t$. The decoder structure for this (n', k, d') code will be more complex.

Example 6.7.1

Suppose block C_2 is deleted from the generator matrix of the (14, 5, 5) KM code of Example 6.6.2, to give the generator matrix of a (11, 5, 3) KM code.

The decoder of the (11, 5, 3) code may be obtained from the decoder of the (14,5,5) code simply by disabling any part of the original decoder associated with C_2. Hence in step(4) of the decoding algorithm reconstruction of subsets I_1, I_2, I_4, I_6, I_8 and I_{10} is not performed. Also, the threshold of the comparator is reduced to 2.

6.8 - A Discussion of the Results

We have seen that the multiplicative complexity of these aperiodic algorithms depends strongly on the choice of the modulo polynomial $P(u)$ and the wraparound coefficient s. Hence there exists $P(u)$ and s such that the multiplicative complexity of the corresponding algorithm is minimal for given values of k and d. Our aim was to find these in order to obtain the associated KM codes with length n as small as possible. The examples presented throughout this work, although varied, illustrate only a tiny part of the wide range of KM Codes which may be derived.

However, the complexity of the decoding is proportional to t, the number of relatively prime factors , $P_i(u)$, of $P(u)$. Further t division circuits are required for encoding , one for each $P_i(u)$. So large number of small degree factors will result in increased complexity but reduced length n for a given value of k and d. Thus a balance point must be found where both the complexity and the code length are reasonable.

We had to question the validity of the decoding algorithm Krishna and Morgera [2] suggested.

Definition 6.1

The *companion matrix* of a monic polynomial $f(u) = a_0+a_1u+. . +a_{n-1}u^{n-1}+u^n$ of positive degree n over a field F is defined to be the (n x n) matrix :

$$C_f = \begin{bmatrix} 0 & 0 & 0 & . & . & . & 0 & -a_0 \\ 1 & 0 & 0 & . & . & . & 0 & -a_1 \\ 0 & 1 & 0 & . & . & . & 0 & -a_2 \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & . & . & . & 1 & -a_{n-1} \end{bmatrix}$$

Further, $f(C_f) = 0$.

Example 6.8.1

Let $f(u) = 1+u^2+u^3+u^4$ be a polynomial over $GF(2)$. Then

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

We state (without proof) Lemma 6.4 which is necessary to prove Lemma 6.5.

Lemma 6.4

Let Xy be a system of bilinear forms where X is a matrix with entries of the form $\sum a_i x_i$, $a_i \in F$ and let t be the minimum number of multiplications needed to compute Xy.

Then, there exists $2t$ linear forms $L_1, \ldots, L_t, L_1', \ldots, L_t'$ of x_i's and y_i's with coefficients in F such that $Xy = Um$ where U is a matrix with entries in F, $\boldsymbol{m} = (m_1, \ldots, m_t)^T$ and $m_i = L_i.L_i'$ $(i = 1, \ldots, t)$ [11].

Lemma 6.5

Let

$$Z(u) = \sum_{i=0}^{\alpha-1} z_i u^i \quad \text{and} \quad Y(u) = \sum_{i=0}^{\alpha-1} y_i u^i$$

be two polynomials with indeterminates z_i and y_i respectively as coefficients and let,

$$P(u) = u^\alpha + \sum_{i=0}^{\alpha-1} a_i u^i$$

be a polynomial of degree α where $a_i \in F$.

Further, suppose $P(u) = Q(u)^\beta$ where $Q(u)$ is irreducible over F. Then the minimum number of multiplications required to compute

$$\Phi(u) \equiv Z(u)Y(u) \text{ modulo } P(u)$$

is $2\alpha - 1$ [11].

Proof

The coefficients of the polynomial $\Phi(u) \equiv Z(u)Y(u)$ modulo $P(u)$ are a system of bilinear forms, which we shall denote by T_p. We will prove the lemma, by showing that the minimum number of multiplications needed to compute T_p is $2\alpha - 1$.

Let C_p be the companion matrix of the polynomial P and let

$$V_p = \{ v \in F^{\alpha} \mid \exists \, polynomial \; r \neq 0, \deg[r] < \alpha \text{ and } vr(C_p) = 0\}$$

Since Q is irreducible, $r(C_p)$ is non-singular whenever Q does not divide r. Let $v \in V_p$, then by definition $vr(C_p) = 0$, for some polynomial r. As a non-trivial linear combination of the rows of $r(C_p)$ is zero, the rank of $r(C_p)$ is less than α. It follows that $r(C_p)$ is singular and so $r = Q^s r'$ where $\beta > s > 0$ and $\gcd(Q, r') = 1$. We have,

$$0 = vr(C_p) = vQ^s(C_p)r'(C_p)$$

Since Q and r' are coprime, $r'(C_p)$ is non-singular. Thus

$$0 = vQ^s(C_p)$$

Consequently ,

$$0 = vQ^{\beta-1}(C_p).$$

Hence, V_p can be defined as,

$$V_p = \left\{ v \in F^{\alpha} \mid vQ^{\beta-1}(C_p) = 0 \right\}.$$

Clearly, V_p is a subspace of F^{α} and $\dim(V_p) < \alpha$.

Let $T_p = Zy$ (for details see Appendix C), then

$$Z = \left(\mathbf{z} \mid C_p \mathbf{z} \mid C_p{}^2 \mathbf{z} \mid \ldots \mid C_p{}^{\alpha-1} \mathbf{z} \right) \text{ where } z = (z_0, z_1, \ldots, z_{\alpha-1})^T$$

Let t be the minimum number of multiplications required to compute T_p. By Lemma 6.4, $Zy = U\boldsymbol{m}$ where U is a $(\alpha \times t)$ matrix with entries in F and $\boldsymbol{m} = (m_1, m_2, \ldots, m_t)^T$. For all non-zero $w \in F^{\alpha}$, $wZ \neq 0$ and thus the rank of matrix Z is α. Since $Zy = U\boldsymbol{m}$, it follows that the rank of U is also α. So U has α linearly independent columns. Without loss of generality, assume that the first α columns of U are linearly independent (where necessary can permute the columns of U and m_i's to achieve this). Hence there exists a non-singular $(\alpha \times \alpha)$ matrix W such that

$$WZy = WU\boldsymbol{m} = (I \mid U')\boldsymbol{m} \qquad (6.14)$$

where I is the identity matrix of dimension α and U' is a $(\alpha \times t-\alpha)$ matrix.

W, being non-singular, spans F^{α} and consequently there exists a row,w, of W which is not in V_p. Suppose w is the jth row of matrix W . Then, by (6.14)

$$wZy = (0, \ldots, \underset{\substack{\uparrow \\ j\text{th position}}}{1}, 0, \ldots, 0 \mid u_1', u_2', \ldots, u_{t-\alpha}')\boldsymbol{m}$$

illustrating that wZy may be computed in t-α+1 multiplications.

We claim that the rank of wZ is α. For, if there exist $\gamma_0, \ldots, \gamma_{\alpha-1} \in F$ such that

$$0 = wZ \, . \, (\gamma_0, \ldots, \gamma_{\alpha-1})$$

i.e.

$$0 = \sum_{i=0}^{\alpha-1} wC_p{}^i \, \mathbf{z}.\gamma_i = w\left(\sum_{i=0}^{\alpha-1} \gamma_i \, C_p{}^i\right)\mathbf{z}$$

Then,

$$w \sum_{i=0}^{\alpha-1} \gamma_i \, C_p{}^i = 0$$

and $w \in V_p$ - contradiction. Hence, $\gamma_i = 0$, $i = 0, 1, \ldots, \alpha-1$ and indeed $\rho(wZ) = \alpha$. Thus, by Lemma 3.3, any computation of $wZ\mathbf{y}$ has at least α multiplications. We observed above that $wZ\mathbf{y}$ may be computed in t-α+1 multiplications, therefore

$$t - \alpha + 1 \geq \alpha$$

i.e.

$$t \geq 2\alpha - 1,$$

which proves the lemma.

#

Lemma 6.5 may be expanded to obtain a lower bound on the number of multiplications required to compute

$$\Phi(u) = Z(u)Y(u)$$

using the convolution algorithm based on the CRT.

Suppose, a polynomial $P(u)$ is chosen to have t coprime factors $P_i(u)$ and the computation is to involve s wraparound coefficients.

Recall, we begin by reducing both $Z(u)$ and $Y(u)$ modulo $P_i(u)$ to give $Z_i(u)$ and $Y_i(u)$ respectively. $\Phi_i(u)$ is then computed as

$$\Phi_i(u) \equiv Z_i(u)\, Y_i(u) \text{ modulo } P_i(u)$$

which by Lemma 6.5 will involve at least $(2\alpha_i$-1) multiplications. Similarily, the wraparound computation will require at least $(2t$-1) multiplications.

Finally, the CRT is used to reconstruct $\Phi(u)$. However, this involves only additions and multiplications by elements of the field F and so does not contribute to the multiplicative complexity. Hence, the minimum number of multiplications, n_{min}, required to compute $\Phi(u) = Z(u)Y(u)$, using this convolution algorithm, is given by :

$$n_{min} = \sum_{i=1}^{t} (2\alpha_i - 1) + (2s - 1)$$

$$= 2\left(\sum_{i=1}^{t} \alpha_i\right) + 2s - \left(\sum_{i=1}^{t} 1\right) - 1$$

$$= 2\left(\sum_{i=1}^{t+1} \alpha_i\right) - \sum_{i=1}^{t+1} 1 \qquad \text{where } \alpha_{t+1} = s$$

$$= 2N - (t + 1).$$

This bound is very useful in determining the efficiency the (n, k, d) code obtained from the algorithm for computing an aperiodic convolution of length N $(=k+d-1)$.

There is no 'useful' upperbound on the multiplicative complexity of the aperiodic convolution. However, trivially, the product of two polynomials of degrees $(k-1)$ and $(d-1)$ can be computed in kd multiplications.

Example 6.8.2

Consider the aperiodic convolution of length $N = 9$, where $P(u) = u(u^2+1)(u^2+u+1)(u^3+u^2+1) = P_1(u)P_2(u)P_3(u)P_4(u)$ and $s = 1$. Then $t = 4$ and (by above), $n_{min} = 2(9)-(4+1) = 13$. The convolution algorithm based on the CRT, results in a corresponding KM code of length 14. Thus the code obtained is of length very close to the theoretical lower bound of this algorithm.

Lemma 6.6

A bilinear algorithm with field of constants $GF(p)$ which is valid for input data over $GF(p)$ remains valid for input data over $GF(p^m)$ [12].

Proof

Consider ϑ a system of bilinear forms. Suppose

$$\vartheta = x * y$$

where x and y are vectors over $GF(p)$.

Then, as observed previously, an algorithm to compute ϑ may be of the form

$$\vartheta = C(Ax \times By) \qquad (6.15)$$

where A, B and C are matrices of appropriate dimension over $GF(p)$ and x denotes component-by-component multiplication of vectors. We wish to show that this computation is valid when x and y are vectors over $GF(p^m)$.

Let α be an element of $GF(p^m)$ and of no smaller field. Recall, that if u is a vector over $GF(p^m)$, then it can be expressed as

$$\mathbf{u} = \sum_{i=1}^{m-1} \mathbf{u}_i \, \alpha^{i}$$

where u_i is a vector over *GF(p)*.

Then if x and y are vectors over $GF(p^m)$, we have

$$\mathbf{x} = \sum_{i=1}^{m-1} \mathbf{x}_i \, \alpha^{i} \quad \text{and} \quad \mathbf{y} = \sum_{j=1}^{m-1} \mathbf{y}_j \, \alpha^{j}$$

where x_i and y_j are vectors over *GF(p)*.

Then

$$\vartheta = \sum_{i=1}^{m-1} \mathbf{x}_i \, \alpha^{i} * \sum_{j=1}^{m-1} \mathbf{y}_j \, \alpha^{j}$$

$$= \sum_{i=1}^{m-1} \sum_{j=1}^{m-1} \alpha^{i+j} \, (\mathbf{x}_i * \mathbf{y}_j)$$

Also, since x_i and y_j are over *GF(p)* , we may use algorithm (6.15) to compute $x_i * y_j$. So,

$$\vartheta = \sum_{i=1}^{m-1} \sum_{j=1}^{m-1} \alpha^{i+j} \, C \, (A \, \mathbf{x}_i \times B \, \mathbf{y}_j)$$

$$= C \left(\sum_{i=1}^{m-1} \sum_{j=1}^{m-1} \left(A \, \mathbf{x}_i \, \alpha^{i} \times B \, \mathbf{y}_j \, \alpha^{j} \right) \right)$$

$$= C \left(A \sum_{i=1}^{m-1} \mathbf{x}_i \, \alpha^{i} \times B \sum_{j=1}^{m-1} \mathbf{y}_j \, \alpha^{j} \right)$$

$$= C(A\,x \times By)$$

Hence, the same algorithm can be used to compute ϑ over *GF(p)* and over $GF(p^m)$.

#

By Lemma 6.6, it follows that the aperiodic convolution algorithms described in detail earlier will remain valid over $GF(2^m)$, $GF(3^m)$, etc. Working over such a field, increases the field of constants. This reduces the multiplicative complexity of the algorithm which we saw earlier results in codes of shorter length for given values of *k* and *d*.

CHAPTER 7 - GH-ARQ Schemes Based on KM Codes

We shall now illustrate how KM Codes may be incorporated into GH-ARQ Schemes. The performance of such schemes will then be investigated.

7.1 - A GH-ARQ Scheme Based on KM Codes

We shall now describe how a KM code may be employed for error-correction in a GH-ARQ scheme. Recall, two codes must be chosen; C0 - an (n, k) code for error-detection and C1 - an (mn, n) code used for adaptive error-correction.

If a KM code is to be used in such a scheme, it must be an (ml',l') KM code having generator matix $C = [C_1|C_2| \ldots |C_m]$ where C_i is an $(l' \times l')$ invertible matrix. Further, the (n, k) code C0, must be chosen such that l' divides n.

Using C0, the k - bit message D is encoded into the n - bit datablock I. Let $I = (i_1, i_2, \ldots, i_n)$. The encoding and decoding procedures of such a scheme may be described in two equivalent ways.

First consider encoding, the two methods are

(a) From matrices C_i, $(i = 1, 2, \ldots, m)$, m $(n \times n)$ matrices $G_1, G_2, \ldots, G_m$ are defined by

$$G_i = C_i \copyright I_{n/l'}$$

where $I_{n/l'}$ denotes an identity matrix of order n/l' and © denotes the Kronecker product of two matrices.
i.e.

$$G_i = \begin{bmatrix} C_i & 0 & 0 & . & . & 0 \\ 0 & C_i & 0 & . & . & 0 \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ 0 & 0 & 0 & 0 & 0 & C_i \end{bmatrix}$$

Then, $G = [G_1|G_2| \ldots |G_m]$ is the generator matrix of the (mn, n) code which shall be used for adaptive error-correction. Using I, the transmitter computes m $(1 \times n)$ vectors c_i $(i = 1, 2, \ldots, m)$ where

$$c_i = IG_i .$$

i.e.

$$c_i = [(i_1 i_2 .. i_{l'})C_i \mid (i_{l'+1} .. i_{2l'})C_i \mid \ldots \mid (i_{n-l'+1} .. i_n)C_i]$$

Since C_i is invertible, it follows that G_i is also invertible $(i = 1, 2, \ldots, m)$. In fact,

$$G_i^{-1} = C_i^{-1} \copyright I_{n/l'}.$$

Further, the data block I may be uniquely recovered from c_i $(i = 1, 2, \ldots, m)$, since

$$I = c_i G_i^{-1}.$$

Equivalently, we may

(b) Subdivide block I into (n/l') subblocks, each of length l'.
i.e.

$$I = (i_1, i_2, .., i_{l'} | i_{l'+1}, .., i_{2l'} | \ldots . | i_{n-l'+1}, .., i_n)$$
$$= (I_1 | I_2 | .. | I_{n/l'})$$

Each of these subblocks is multiplied by C, the generator matrix of the (ml', l') KM code.
i.e.

$$I_j C = I_j [C_1 C_2 \ldots C_m] = [I_j C_1 | I_j C_2 | \ldots | I_j C_m] \qquad j = 1, 2, .., n/l'$$

It is obvious that the l' digits of $I_j C_1$ correspond to the $(l'(j-1)+1)$th up to $l' j$ th digit of c_1, the l' digits of $I_j C_2$ correspond to the $(l'(j-1)+1)$th up to $l' j$ th digit of $c_2, \ldots,$ and finally $I_j C_m$ corresponds to the $(l'(j-1)+1)$th up to $l' j$ th digit of c_m $(j = 1, 2, .., n/l')$.
i.e.

$$c_i = [I_1 C_i | I_2 C_i | \ldots | I_{n/l'} C_i] \qquad i = 1, 2, \ldots, m.$$

Let R_i denote the block received corresponding to the transmitted block c_i $(i = 1,2,..,m)$. From R_i, we may obtain E_i, an estimate of the block I in one of two ways.

(a) Recall,

$$c_i = IG_i.$$

Hence $\qquad i = 1,2,\ldots,m$

$$E_i = R_i G_i^{-1}$$

or,

(b) Subdivide block R_i into n/l' subblocks of length l'. Then multiply each subblock by the matrix C_i^{-1}, to give an estimate of the corresponding subblock of E_i $(i=1,2,..,m)$

In either case, if no errors have occurred, then $R_i = c_i$ and $E_i = I$. Blocks c_1, $c_2, \ldots, c_m$ are transmitted in turn until ACK is delivered to the transmitter.

The full receiver configuration corresponding to method (a) is detailed in Section 2.3. For completeness, the receiver configuration related to method (b) is now given:

On receiving R_i, decoding as described in (b) above is performed, to obtain E_i, an estimate of I. Then using C0, E_i is checked for errors. If E_i is found to be error-free, then it is assumed that $E_i = I$ and ACK is sent to the transmitter. If however, E_i contains detectable errors, $R_1, R_2, \ldots, R_i$ are each divided into n/l' subblocks of length l'. The jth subbblocks of $R_1, R_2, \ldots, R_i$ are combined to form a vector v_j of the (il', l') KM code, $C1^{(i)}$, with generator matrix $[C_1 | C_2 | \ldots | C_i]$ $(j=1,2,\ldots, n/l')$. Vector v_j is decoded, using $C1^{(i)}$, to obtain I_j^0, an estimate of I_j. Then $I^0 = [I_1^0 | I_2^0 | \ldots | I_{n/l'}^0]$ is tested for errors using C0. If I^0 is found to contain errors, NACK is sent to the transmitter and $R_1, R_2,$.

. ,R_i are stored in the receiver buffer. Otherwise, it is assumed that $I^0=I$ and ACK is sent to the transmitter.

Example 7.1.1

In the above general form, the receiver configuration may appear very complicated, so we shall consider a specific example where C0 is a (500,480) code obtained from (1023,1003) BCH code and a (15,5,5) KM code with (10,5,3) subcode is used. This is a depth 3 GH-ARQ scheme.

Consider the first transmission. Suppose that R_1 contains errors as shown below. Then

$c_1 = 5\ 5\ 5\ 5. \dots 5$ →→transmit→→ $R_1=5\ 5^*\ 5^*\ 5^* \dots . 5$

100 blocks each of length 5 *contains errors

E_1, an estimate of I based on R_1 is found to contain errors - another transmission is requested. Suppose that E_2 also contains detectable errors. Then

$R_1= 5\ 5^*5^*5^* \dots . 5\ 5\ 5\ 5$

↓ ↓ ↓ ↓

Decode each vector of (10,5,3) KM code

↓ ↓ ↓ ↓

$R_2= 5\ 5^*5\ 5 \dots . 5^*5\ 5^*5$

↓ ↓ ↓ ↓

$I_1^0\ I_2^0$ $I_{99}^0 I_{100}^0$

Each pair of corresponding blocks is combined to form a vector of the (10,5,3) KM code. Each of these vectors is decoded to I_j^0 (j=1,2,. .,100). Then $I^0 = (I_1^0,..,I_{100}^0)$, an estimate of I is formed. However, the (10,5,3) code fails to correct all the errors present. Thus $I^0 \neq I$ and another transmission is required.

If the estimate of I obtained from R_3 is found to be incorrect, then

$R_1= 5\ 5^*5^*5^* \dots . 5\ 5\ 5\ 5$

↓ ↓ ↓ ↓

$R_2= 5\ 5^*5\ 5 \dots . 5^*5\ 5^*5$

↓ ↓ ↓ ↓

Decode each vector of (15,5,5) KM code

↓ ↓ ↓ ↓

$R_3= 5^*5\ 5^*5 \dots . 5\ 5^*5\ 5$

↓ ↓ ↓ ↓

$I_1^0\ I_2^0$ $I_{99}^0 I_{100}^0$

This time each triplet of corresponding blocks combine and are decoded, using the (15,5,5) KM code, to form I_j^0 (j=1,2,. . ,100). The errors in the blocks are

removed. Decoding is successful, $I^0 = (I_1{}^0,..,I_{100}{}^0)$, a codeword of C0 is found to be error-free. We assume $I^0=I$ and ACK is sent to the transmitter.
Note If this I^0 had been found to contain errors, R_1 would be discarded and NACK sent to the transmitter. The blocks will continue to be transmitted in turn until decoding is successful.

7.2 - The Partitioning of the Generator Matrix of KM Codes

Recall, C, the generator matrix of a (n, k, d) KM Code has $(t+1)$ blocks. Each block C_i, corresponds to a computation of the form

$$\Phi_i(u) \equiv Z_i(u)Y_i(u) \text{ modulo } P_i(u) \quad \text{where } \deg[P_i(u)] = \alpha_i.$$

Further, it was noted that block C_i has α_i linearly independent columns which correspond to the computation

$$Z_i(u) \equiv Z(u) \text{ modulo } P_i(u).$$

Each α_i is small and so the computation of $\Phi_i(u)$ can be performed such that the rank of the remaining $(n_i-\alpha_i)$ columns is equal to $\max(\alpha_i, n_i-\alpha_i)$. This ensures that the partitions of the generator matrix are invertible.

If the polynomial $P_i(u)$ corresponding to the block C_i, is of the form $(u+a_i)^{\alpha_i}$, $a_i \in F$, then this block may be divided into sub-blocks given by,

$$C_i = [C_i{}^1 | \, C_i{}^2 | \, ...|C_i{}^{\alpha_i}]$$

where the sub-block $C_i{}^1$ corresponds to the computation

$$\Phi_i(u) \equiv Z_i(u)Y_i(u) \text{ modulo } (u+a_i)$$

and the sub-blocks $[C_i{}^1|C_i{}^2|......|C_i{}^j]$ correspond to the computation

$$\Phi_i(u) \equiv Z_i(u)Y_i(u) \text{ modulo } (u+a_i)^j.$$
$$\text{for } j = 1, 2, . ., \alpha_i$$

This property holds also for block C_{t+1}, the wraparound block.

The following example illustrates how this partitioning procedure suggests a way in which to arrange the columns of the generator matrix of a KM code so that it may be divided into $(k \times k)$ invertible subblocks.

Example 7.2.1

Consider the (15,5,5) KM code. Here $P(u) = u^3(u^2+1)(u^2+u+1)$, $s = 2$, $Z(u) = z_0+z_1u+z_2u^2+z_3u^3+z_4u^4$ and $Y(u) = y_0+y_1u+y_2u^2+y_3u^3+y_4u^4$.

Note With $k = 5$ & $d = 5$, it is possible to obtain a KM code of length 14 by taking $P(u) = u(u^2+1)(u^2+u+1)(u^3+u^2+1)$ and $s = 1$. Indeed up until now, we have concerned ourselves with finding a KM code of minimal length for given k and d. However, as we previously observed, in order that these codes may be employed in a GH-ARQ scheme, it is required that the generator matrix can be divided into subblocks of dimension (k x k). Hence the length must be an integral multiple of the dimension k. For this particular k and d, the minimal KM code does not meet these requirements. This is not always the case. For example with $k = 4$ and $d = 7$, the minimal KM code is of length 16 and so clearly could be employed in a GH-ARQ scheme as described in Section 7.1.

Reducing the polynomials $Z(u)$ and $Y(u)$ modulo each $P_i(u)$, we obtain

$i = 1$

$$Z_1(u) \equiv Z(u) \text{ modulo } u^3 \\ = z_0+z_1u+z_2u^2$$

and

$$Y_1(u) \equiv Y(u) \text{ modulo } u^3 \\ = y_0+y_1u+y_2u^2.$$

Let,

$$m_0 = z_0y_0, \\ m_1 = z_1y_1, \\ m_2 = z_2y_2, \\ m_3 = (z_0+z_1).(y_0+y_1), \\ m_4 = (z_0+z_2).(y_0+y_2)$$

and

$$m_5 = (z_1+z_2).(y_1+y_2).$$

$i = 2$

$$Z_2(u) \equiv Z(u) \text{ modulo } (u^2+1) \\ = (z_0+z_2+z_4) + (z_1+z_3)u$$

and

$$Y_2(u) \equiv Y(u) \text{ modulo } (u^2+1) \\ = (y_0+y_2+y_4) + (y_1+y_3)u.$$

Let,

$$m_6 = (z_0+z_2+z_4).(y_0+y_2+y_4), \\ m_7 = (z_1+z_3).(y_1+y_3)$$

and

$$m_8 = (z_0+z_1+z_2+z_3+z_4).(y_0+y_1+y_2+y_3+y_4).$$

$i = 3$

$$Z_3(u) \equiv Z(u) \text{ modulo } (u^2+u+1) \\ = (z_0+z_2+z_3) + (z_1+z_2+z_4)u$$

and

$$Y_3(u) \equiv Y(u) \text{ modulo } (u^2+u+1)$$
$$= (y_0+y_2+y_3) + (y_1+y_2+y_4)u.$$

Let,

$$m_9 = (z_0+z_2+z_3).(y_0+y_2+y_3),$$
$$m_{10} = (z_1+z_2+z_4).(y_1+y_2+y_4)$$

and

$$m_{11} = (z_0+z_1+z_3+z_4).(y_0+y_1+y_3+y_4).$$

Wraparound

$$\overline{Z}(u) \equiv z_4 + z_3 u \text{ modulo } u^2$$

and

$$\overline{Y}(u) \equiv y_4 + y_3 u \text{ modulo } u^2 .$$

Let,

$$m_{12} = z_4.y_4,$$
$$m_{13} = z_3.y_3$$

and

$$m_{14} = (z_3+z_4).(y_3+y_4).$$

It follows that C, the generator matrix of the (15, 5, 5) KM code is given by ,

$$C = \begin{array}{cccccc|ccc|ccc|ccc} 1&0&0&1&1&0&1&0&1&1&0&1&0&0&0 \\ 0&1&0&1&0&1&0&1&1&0&1&1&0&0&0 \\ 0&0&1&0&1&1&1&0&1&1&1&0&0&0&0 \\ 0&0&0&0&0&0&0&1&1&1&0&1&0&1&1 \\ 0&0&0&0&0&0&1&0&1&0&1&1&1&0&1 \\ &&&\Uparrow&&&&\Uparrow&&&\Uparrow&&&\Uparrow& \\ &&&C_1&&&&C_2&&&C_3&&&C_4& \end{array}$$

The block corresponding to the computation $Z_1(u)Y_1(u)$ modulo u^3 is C_1. This block may be divided into three sub-blocks $C_1{}^1$, $C_1{}^2$ and $C_1{}^3$ where $[C_1{}^1]$ corresponds to the computation $Z_1(u)Y_1(u)$ modulo u, $[C_1{}^1|C_1{}^2]$ corresponds to $Z_1(u)Y_1(u)$ modulo u^2 and $[C_1{}^1|C_1{}^2|C_1{}^3]$ corresponds to $Z_1(u)Y_1(u)$ modulo u^3.
i.e.

$$C_1^1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad C_1^2 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad C_1^3 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Similarly, C_4, the block corresponding to the computation $\overline{Z}(u)\ \overline{Y}(u)$ modulo u^2 may be partitioned into two subblocks, $C_4{}^1$ and $C_4{}^2$. $[C_4{}^1]$ is associated with the computation $\overline{Z}(u)\ \overline{Y}(u)$ modulo u while $[C_4{}^1 | C_4{}^2]$ is associated with the computation $\overline{Z}(u)\ \overline{Y}(u)$ modulo u^2. These are given by :

$$C_4^1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad\qquad C_4^2 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}$$

By removing the subblocks $C_1{}^3$ and $C_4{}^2$ from C and rearranging the remaining columns, we obtain C', the generator matrix of a (10, 5, 3) code corresponding to $P(u) = u^2(u^2+1)(u^2+u+1)$ and $s = 1$. C' may be partitioned into two subblocks, $C_1{}'$ and $C_2{}'$ each of which is invertible.
i.e.

$$C' = \left[\begin{array}{ccccc|ccccc} 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{array}\right]$$
$$\begin{array}{cc} \Uparrow & \Uparrow \\ C_1' & C_2' \end{array}$$

The columns of $C_1{}^3$ and $C_4{}^2$ may then be added to the end of C' to give C_m, a modified generator matrix of (15, 5, 5) code. C_m consists of three invertible partitions.
i.e.

$$C_m = \left[\begin{array}{ccccc|ccccc|ccccc} 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right]$$
$$\begin{array}{ccc} \Uparrow & \Uparrow & \Uparrow \\ C_m^1 & C_m^2 & C_m^3 \end{array}$$

Figure 7.1 shows the transmission procedure while Figure 7.2 shows the receiver configuration for a depth 3 GH-ARQ scheme based on a (15,5,5) KM code.

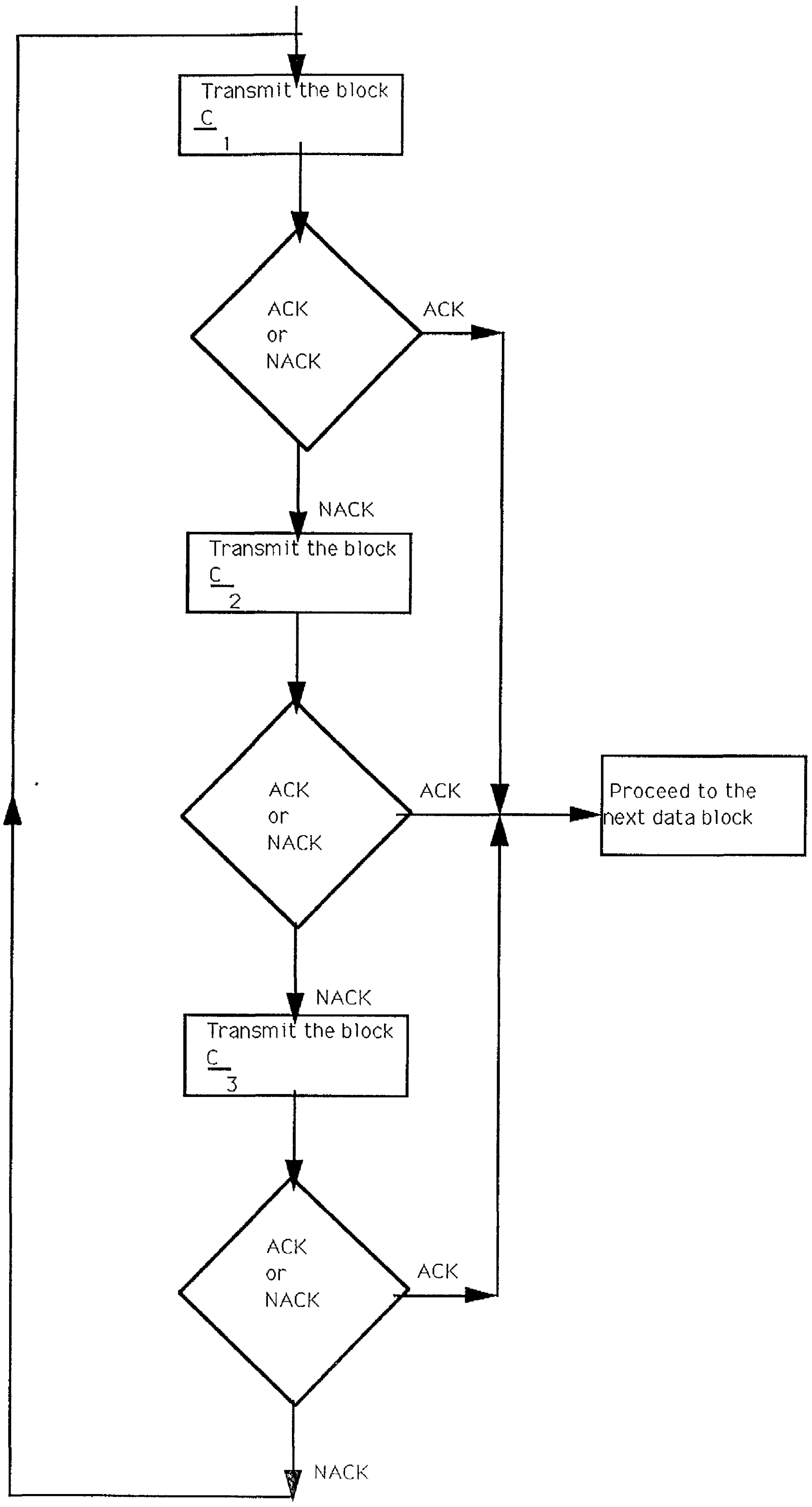
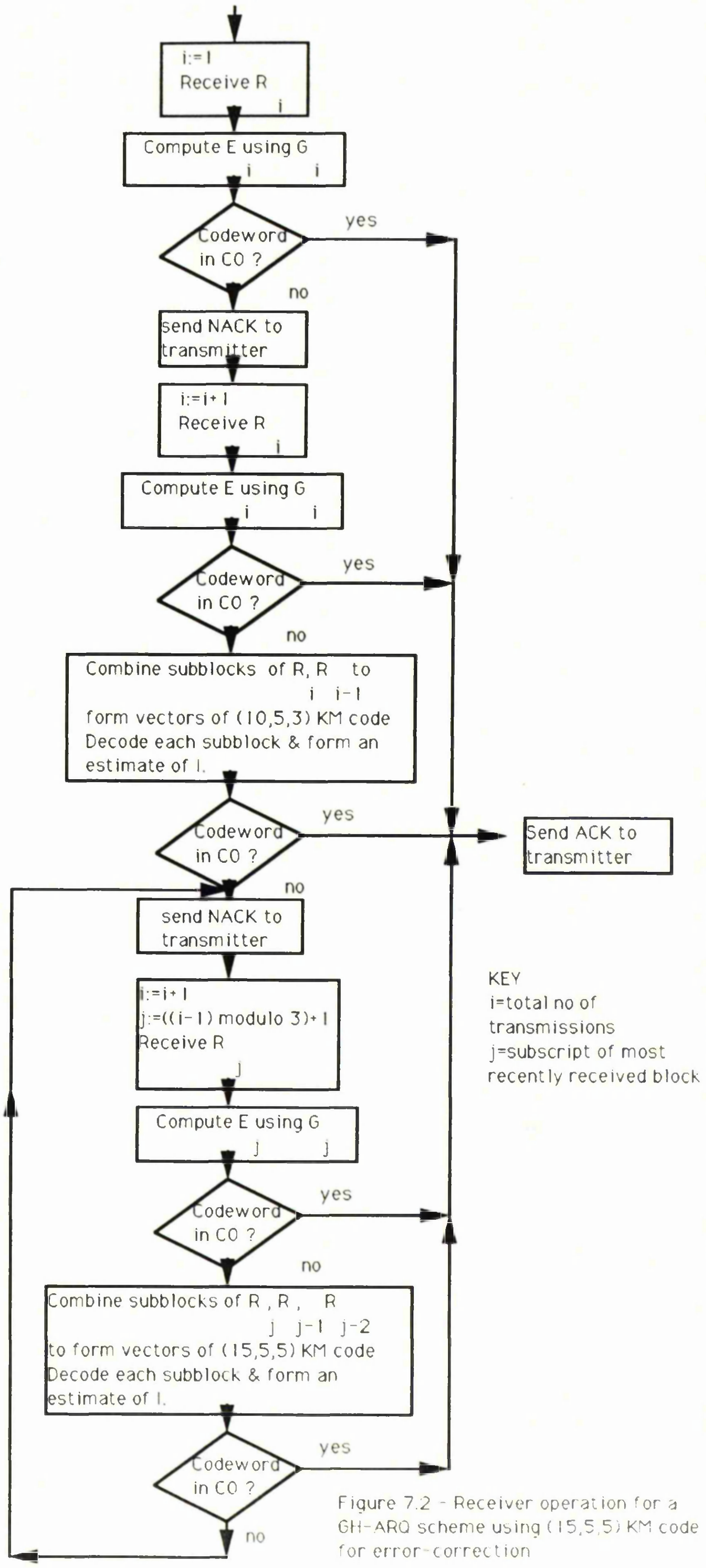


Figure 7.1- Transmission procedure for a GH-ARQ scheme using depth 3 code for error-correction



i:=1
Receive R_i
Compute E_i using G_i
Codeword in C0 ?
yes
no
send NACK to transmitter
i:=i+1
Receive R_i
Compute E_i using G_i
Codeword in C0 ?
yes
no
Combine subblocks of R_i, R_{i-1} to form vectors of (10,5,3) KM code Decode each subblock & form an estimate of I.
Codeword in C0 ?
yes
no
Send ACK to transmitter
send NACK to transmitter
i:=i+1
j:=((i-1) modulo 3)+1
Receive R_j
Compute E_j using G_j
Codeword in C0 ?
yes
no
Combine subblocks of R_j, R_{j-1}, R_{j-2} to form vectors of (15,5,5) KM code Decode each subblock & form an estimate of I.
Codeword in C0 ?
yes
no
KEY
i=total no of transmissions
j=subscript of most recently received block

Figure 7.2 - Receiver operation for a GH-ARQ scheme using (15,5,5) KM code for error-correction

7.3 - Error-Detection

We now consider the error-detection capability of the GH-ARQ scheme - an important factor of the system performance.

Recall, an (n, k) code C0 is employed for error-detection in GH-ARQ schemes. Let P_e be the probability that an error will pass undetected and ε be the bit error rate of the channel. If an (n,k) code satisfies [4]

$$P_e \leq [1-(1-\varepsilon)^k]2^{-(n-k)} \qquad 0 \leq \varepsilon \leq 1/2 \tag{7.1}$$

then P_e may be reduced simply by using more parity check bits (i.e.by increasing n-k). Although, it is not possible to prove the existence of families of codes which satisfy this bound, for specific n and k, it is usually possible to find an (n, k) code satisfying (7.1) [2].

Assuming the (n, k) code C0 employed in our GH-ARQ scheme satisfies bound (7.1), P_e can be made arbitrarily small.

Recall, in GH-ARQ schemes, the receiver obtains E_i , an estimate of the original codeword I , by taking the inverse of the received vector R_i. C0 is then used to determine whether E_i is a valid codeword. That is,

$$E_i \text{ is a codeword in C0 iff } R_i G_i^{-1} H^T = 0$$

where the subscript i represents the i th transmission.

Let $H_i^{\sim T} = G_i^{-1} H^T$. Then an error pattern e in the i th transmission of a block will be undetectable if and only if e is a codeword of a linear code having parity check matrix $H_i^{\sim}$.

At present, it is not possible to prove that the probability of undetected error of a general GH-ARQ scheme satisfies the above bound. However, below is an example of a specific GH-ARQ scheme which does indeed satisfy bound (7.1) [2].

Example 7.3.1

Consider a GH-ARQ scheme employing the following codes; C0 for error-detection - (500, 480) code obtained by shortening the distance 5 (1023, 1003) BCH code and C1 for error-correction - (1500, 500) invertible code obtained from the (15, 5, 5) KM code derived in Example 7.2.1 .

Figure 7.3 shows the probability of undetected error for successive transmissions in the above GH-ARQ scheme. Note, that since this GH-ARQ scheme is of depth 3, there are three plots for the probability of undetected errors corresponding to the first transmission and the first and second retransmissions.

We notice that the probability of undetected error is a monatomic function of ε, $0 \leq \varepsilon \leq 1/2$. This condition is necessary and sufficient for a code to satisfy bound (7.1) [13] .

KEY

$n = 500$
$k = 480$

1. 1st transmission
2. 1st retransmission
3. 2nd retransmission

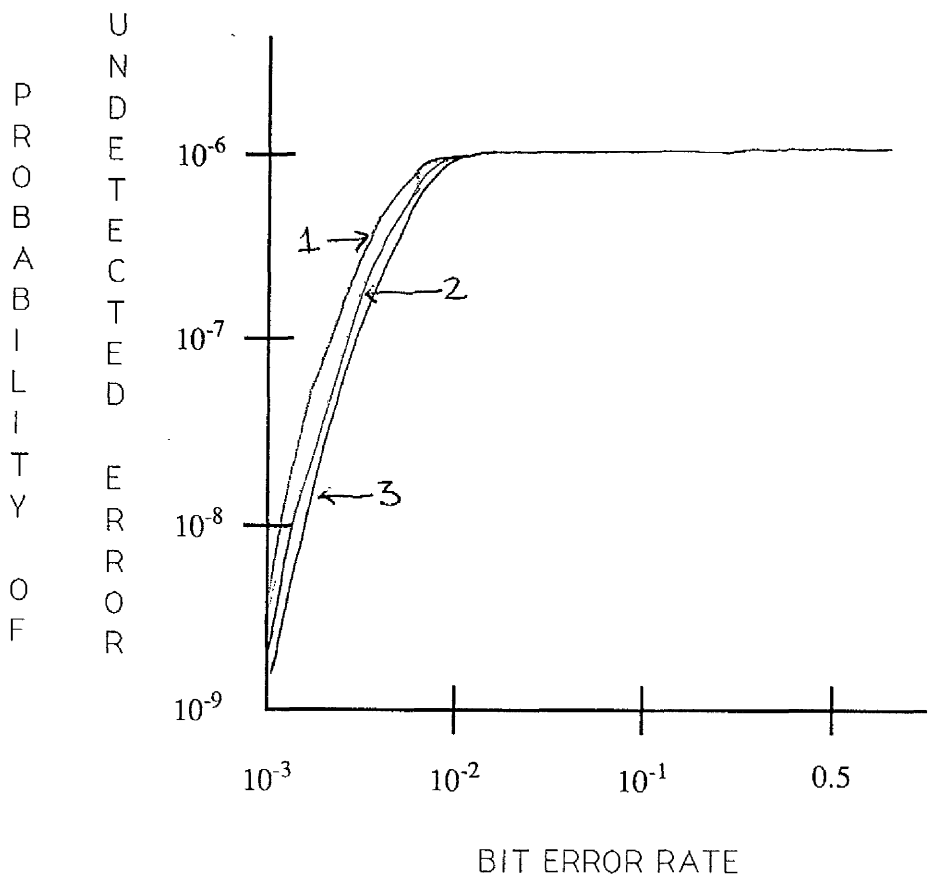


Figure 7.3 - Probability of undetected error for the GH-ARQ scheme using (15, 5, 5) KM code for error-correction.

7.4 - Burst Error-Detection Capability of GH-ARQ Schemes

We now turn our attention to the burst error-detection capability of the error-detecting codes used in GH-ARQ schemes. (n, k) cyclic codes are commonly chosen for error-detection in ARQ schemes as they have a fairly good burst-error detection capability. In fact, an (n, k) cyclic code can detect error-bursts of up to length $(n - k)$.

Recall, for GH-ARQ schemes, the receiver multiplies subblocks of the received vector by the corresponding inverse matrix to find an estimate of the transmitted data block I. Thus, any errors which occur in a received subblock will also be present in the inverted subblock. We wish to determine the maximum length of an error burst which can be detected in this scheme. Consider the worst case. Assume that if a received subblock contains at least one error, then all the digits of the corresponding inverted subblock will be in error. Let l' be the length of each subblock and let n' be the number of received blocks containing an error burst. Then, after inversion, for the worst case, the length of the burst will be $l'n'$. If this error burst is to be detected by the (n, k) cyclic error-detecting code, we must have :

$$l'n' \leq (n - k).$$

i.e.

$$n' \leq [(n - k)/l']$$

Hence, the maximum number of received subblocks affected by a burst of errors which will be detected by the receiver is given by :

$$n_{max}' = [(n - k)/l'].$$

It can be shown that the maximum length of a detectable error burst which affects n_{max}' subblocks is $l'(n_{max}' - 1) + 1$ [2]. Thus the burst error-detection capability of the GH-ARQ scheme is underbounded by $l'(n_{max}' - 1) + 1$.

Example 7.4.1

Consider the GH-ARQ scheme described in Example 7.3.1 where C0 = (500, 480) code and C1 = (15, 5, 5) KM code.

Then $n - k = 500 - 480 = 20$ and $l' = 5$. So $n_{max}' = 20/5 = 4$ and the burst error-detection capability is at least $5(4-1)+1 = 16$.

7.5 - Performance Analysis of GH-ARQ Schemes

To determine the system performance of a GH-ARQ scheme, we shall consider its throughput efficiency and its reliability. Throughout, the following analysis, let T_0^c, T_0^d and T_0^e be the events that a data block contains no errors, detectable errors and undetectable errors respectively, in the first transmission. Also, let B_i^c, B_i^d, B_i^e denote the events that the i th retransmission of a block contains no errors, detectable errors and undetectable errors, respectively. Further, upon the i th retransmission, let D_i^c, D_i^d, D_i^e be the events that the block obtained by decoding the blocks received up to the i th retransmission is error-free, contains detectable errors and contains undetectable errors, respectively. Finally, upon the i th retransmission, let E_i^c be the event that the receiver recovers the data block correctly; E_i^d be the event that the receiver fails to recover the data block, is aware of the presence of errors and thus requests another retransmission; and E_i^e be the event that the receiver recovers the data block incorrectly but declares it error-free. Clearly,

$$\Pr(T_0^c) + \Pr(T_0^d) + \Pr(T_0^e) = 1$$

and

$$\Pr(B_i^c) + \Pr(B_i^d) + \Pr(B_i^e) = 1. \tag{7.2}$$

Further, let the probability that one given bit will be received incorrectly be ε, then the probability that it will be error-free is $(1-\varepsilon)$. As a block contains n bits,

$$\Pr(T_0^c)=\Pr(B_i^c)=(1-\varepsilon)^n\,. \tag{7.3}$$

If the event E_i^c takes place then either

(a) the i th retransmission of a block is successful and so the receiver may obtain the data block from this

or

(b)the i th retransmission for a block contains detectable errors.These errors are corrected and the block obtained by decoding all the blocks received up to the i th retransmission is error-free.

Hence,

$$E_i^c = B_i^c \cup B_i^d D_i^c. \tag{7.4}$$

When the event E_i^d occurs then

(a) the i th retransmission for the block is found to contain errors

and

(b) decoding all the blocks received up to the i th retransmission, results in a block which again is found to contain errors.

Hence,

$$E_i^d = B_i^d D_i^d. \tag{7.5}$$

Finally the event E_i^e means

(a) the receiver fails to detect the presence of the errors which have occured in the i th retransmission for a block

or

(b) the receiver does detect the presence of errors in this block but when the blocks received up to the i th retransmission are decoded, the recovered block contains errors. However, the receiver fails to detect these errors.

Hence,

$$E_i^e = B_i^e \cup B_i^d D_i^e. \tag{7.6}$$

Let T denote the total number of transmissions required to recover a block successfully in a GH-ARQ scheme. Note that here the i th retransmission corresponds to a total of $(i+1)$ transmissions for a block.

7.5.1 - Throughput Efficiency of GH-ARQ Schemes

We wish to investigate the throughput of a GH-ARQ scheme. Selective-repeat ARQ is the most efficient ARQ scheme and so we shall study the throughput of the GH-ARQ scheme in the selective-repeat mode. The throughput of such a scheme depends on the buffer size. To simplify our analysis, the buffer is assumed to be of infinite size. Further, we shall assume that the feedback channel is noiseless. (Note, ε denotes the bit error rate of the forward transmission channel.)

Then, $E\,[T\,]$, the expected value for T is given by

$$E\,[T\,] = \Pr[T_0{}^c+T_0{}^e\,] + 2\Pr[T_0{}^d\,(E_1{}^c+E_1{}^e\,)] + 3\Pr[T_0{}^dE_1{}^d\,(E_2{}^c+E_2{}^e\,)] + \dots + (i+1)\Pr[T_0{}^dE_1{}^dE_2{}^d..E_{i-1}{}^d\,(E_i{}^c+E_i{}^e\,)] + \dots \tag{7.7}$$

and η, the throughput efficiency of this system is given by

$$\eta = \frac{1}{E\,[T\,]}.\frac{k}{n} \tag{7.8}$$

where k/n is the rate of the error-detecting code C0.
The inequalities $\Pr[E_i{}^c\,] \gg \Pr[E_i{}^d\,]$ and $\Pr[T_0{}^c\,] \gg \Pr[T_0{}^e\,]$ may be used to obtain an excellent approximation of $E\,[T\,]$. However, the expression (7.8), above for $E\,[T\,]$ involves the probability of joint events which are difficult to determine. Thus another approach must be sought to continue any further analysis.

In particular, we shall look at the performance of a GH-ARQ scheme which employs a depth 3 code C1 for error-correction. Let C1 be the code derived from a $(3l', l')$ KM code as described in Section 7.2 (method (a)). Then, G, the generator matrix of C1 is given by

$$G = [G_1\,|G_2\,|\,G_3]$$

and let C1$^{(2)}$ be the code with generator matrix $[G_1\,|G_2]$. Then C1$^{(2)}$ is obtained from the $(2l', l')$ KM code, a subcode of the $(3l', l')$ KM code. Also, let t_1 and t_2 be the error-correcting capability of the $(3l', l')$ and $(2l', l')$ codes respectively. Clearly $t_1 > t_2$.

We shall introduce two systems, each having a reduced throughput compared to the proposed system.

System A - error-correction using C1$^{(2)}$ is performed at every odd retransmission while only error-detection is performed at every even retransmission.

System B - error-correction based on the code C1 takes place at every third transmission and only error-detection is performed on all other transmissions.

Each of these inferior schemes can be easily analysed. Thus we may obtain the throughput of both System A and System B. The throughput of the actual GH-ARQ scheme is underbounded by the maximum of the throughputs of the two inferior systems. This approach is very similar to that used by Wang and Lin in reference [15].

Recall, error-detection in the GH-ARQ scheme is performed using codes having parity check matrices $H_i^{\sim}$ ($H_i^{\sim T} = G_i^{-1}H^T$ where H is the parity check matrix of code C0). Let Pe_i be the probability of undetected error for these codes and let,

$$P_e = \max(Pe_i,\ i = 1, 2, 3).$$

We shall assume that P_e satisfies bound (7.1) and so P_e can be made arbitrarily small.

Let P_c be the probability that the j th transmission of any block received is error-free and let P_d be the probability that errors are detected in this transmission. From (7.3), P_c is given by :

$$P_c = (1 - \varepsilon)^n.$$

Also, for the j th transmission,

$P_d =$ 1- (probability that no errors occur) - (probability that undetected errors occur).

However, as the probability of a transmission being error-free is significantly larger than Pe_i (i = 1, 2, 3), we may use the approximation,

$$P_d \cong 1-P_c. \tag{7.9}$$

We are now ready to analyse systems A and B.

System A

For R_{i-1} and R_i , two consecutively received blocks corresponding to a data block I, let,

δ_0 = Probability that correct decoding takes place based on $C1^{(2)}$,

y = Probability that
 (a) decoding based on $C1^{(2)}$ is correct and
 (b) at least one of R_{i-1} and R_i is error-free,

δ_1 = Conditional probabilty that decoding based on $C1^{(2)}$ is correct given that R_{i-1} and R_i are found to contain errors.

Now $\Pr[B_i^c] = P_c$ and $\Pr[B_{i-1}^d] = \Pr[B_i^d] = P_d \cong 1-P_c$ (by (7.9)).

Consider, the joint probabilty $\Pr[T_0^dE_1^d..E_{i-1}^d(E_i^c+E_i^e)]$. This may be expressed as

$$\Pr\left[T_0^d E_1^d \ldots E_{i-1}^d (E_i^c + E_i^e)\right] =$$

$$\begin{cases} \Pr\left[T_0^d E_1^d\right]\Pr\left[E_2^d E_3^d\right]\ldots \Pr\left[E_{i-1}^d (E_i^c + E_i^e)\right] & \text{for } i \text{ odd} \\ \Pr\left[T_0^d E_1^d\right]\Pr\left[E_2^d E_3^d\right]\ldots \Pr\left[E_{i-2}^d E_{i-1}^d\right]\Pr\left[E_i^c + E_i^e\right] & \text{for } i \text{ even} \end{cases} \tag{7.10}$$

We must deal with each of these cases separately.

i odd

Then only error-detection is performed upon transmission (i -1) and so

$$B_{i-1}^{d} = E_{i-1}^{d}. \tag{7.11}$$

Thus for i odd,

$$\begin{aligned}
&\Pr[E_{i-1}^{d}(E_i^c+E_i^e)] \\
&\quad \leq \Pr[E_{i-1}^{d}E_i^c] \\
&\quad = \Pr[E_{i-1}^{d}]\Pr[E_i^c \mid E_{i-1}^{d}] \\
&\quad = \Pr[B_{i-1}^{d}]\Pr[B_i^c \cup B_i^d D_i^c \mid B_{i-1}^{d}] && \text{by (7.11) \&(7.4)} \\
&\quad = \Pr[B_{i-1}^{d}]\{\Pr[B_i^c] + \Pr[B_i^d D_i^c \mid B_{i-1}^{d}]\} \\
&\quad = \Pr[B_{i-1}^{d}]\{\Pr[B_i^c] + \Pr[B_i^d]\Pr[D_i^c \mid B_{i-1}^{d}B_i^d]\} \\
&\quad \cong (1-P_c)\{P_c + (1-P_c).\Pr[D_i^c \mid B_{i-1}^{d}B_i^d]\}. && (7.12)
\end{aligned}$$

$\Pr[D_i^c \mid B_{i-1}^{d}B_i^d]$ is the conditional probability that the block obtained by decoding the blocks received up to the i th retransmission will be error-free, given that the (i -1)th and the i th retransmissions for the block contain errors.
Then

$$\Pr\left[D_i^{\,c} \middle| B_{i-1}^{\,d} B_i^{\,d}\right] = \delta_1 = \frac{\delta_0 - y}{1-y} \tag{7.13}$$

Hence, substituting (7.13) into (7.12), we have for i odd,

$$\begin{aligned}
\Pr[E_{i-1}^{d}(E_i^c+E_i^e)] &\leq (1-P_c)[P_c+(1-P_c)\delta_1] \\
&= (1-P_c)P_t
\end{aligned} \tag{7.14}$$

where $P_t = [P_c+(1-P_c)\delta_1]$.

Looking at our expansion of $\Pr[E_{i-1}^{d}(E_i^c+E_i^e)]$ (i odd), we can see that $\Pr[E_i^c \mid E_{i-1}^{d}]$ has been replaced by P_t. Hence, P_t is the event that the receiver recovers the data block correctly, upon the ith retransmission, given that received block R_{i-1} is detected in error.

i even

Then, on the i th retransmission, only error-detection takes place. Hence, if the receiver recovers the data block correctly upon the i th retransmission, the i th retransmission of the block must have been delivered error-free.
i.e.

$$E_i^c = B_i^c \tag{7.15}$$

Then,

$$\begin{aligned} \Pr[E_i^c+E_i^e] &\le \Pr[E_i^c] \\ &= \Pr[B_i^c] && \text{by (7.15)} \\ &= P_c. && (7.16) \end{aligned}$$

Now consider the probability $\Pr[E_{j-1}^d E_j^d]$ for j odd.

$$\begin{aligned} \Pr[E_{j-1}^d E_j^d] &= \Pr[E_{j-1}^d]\Pr[E_j^d | E_{j-1}^d] \\ &= \Pr[E_{j-1}^d]\{1-\Pr[E_j^c | E_{j-1}^d] - \Pr[E_j^e | E_{j-1}^d]\} \\ &\le \Pr[B_{j-1}^d]\{1-\Pr[E_j^c | E_{j-1}^d]\} && \text{by (7.11)} \\ &\cong (1-P_c)\{1-\Pr[E_j^c | E_{j-1}^d]\} \\ &= (1-P_c)(1-P_t). && (7.17) \end{aligned}$$

Further, it can be shown that [2],

$$\delta_0 = \left[\sum_{j=0}^{t_2}\binom{2l'}{j}\varepsilon^j(1-\varepsilon)^{2l'-j}\right]^{n/l'}$$

and (7.18)

$$y = (1-\varepsilon)^n\left\{2\left[\sum_{j=0}^{t_2}\binom{l'}{j}\varepsilon^j(1-\varepsilon)^{l'-j}\right]^{n/l'} - (1-\varepsilon)^n\right\}.$$

Let $\sigma = 1-(1-P_c)(1-P_t)$, combining the above inequalities, we obtain

$$\Pr\left[T_0^d E_1^d \ldots E_{i-1}^d (E_i^c + E_i^e)\right] \le \begin{cases} P_t(1-P_c)(1-\sigma)^{\frac{i-1}{2}} & \text{for } i \text{ odd} \\ P_c(1-\sigma)^{\frac{i}{2}} & \text{for } i \text{ even} \end{cases}. \quad (7.19)$$

Hence for system A,

$$\begin{aligned} E[T]_A &= 1.\Pr\left[T_0^c + T_0^e\right] + 2.\Pr\left[T_0^d(E_1^c+E_1^e)\right] + \\ &\quad 3.\Pr\left[T_0^d E_1^d(E_2^c+E_2^e)\right] + \ldots \\ &\quad \ldots + (i+1).\Pr\left[T_0^d E_1^d \ldots E_{i-1}^d(E_i^c+E_i^e)\right] + \ldots \\ &\le P_c + 2P_t(1-P_c) + 3P_c(1-\sigma) + 4P_t(1-P_c)(1-\sigma) + \\ &\quad 5P_c(1-\sigma)^2 + 6P_t(1-P_c)(1-\sigma)^2 + \\ &\quad 7P_c(1-\sigma)^3 + 8P_t(1-P_c)(1-\sigma)^3 + \ldots \\ &= P_c\sum_{i=0}^{\infty}(2i+1)(1-\sigma)^i + 2P_t(1-P_c)\sum_{j=o}^{\infty}(j+1)(1-\sigma)^j \\ &= \frac{2-P_c}{P_c+P_t-P_cP_t}. \end{aligned}$$

i.e.

$$E[T]_A \le \frac{2-P_c}{P_c+P_t-P_c P_t} \tag{7.20}$$

System B

For R_{i-2}, R_{i-1} and R_i, three consecutively received vectors corresponding to I, let,

δ_0 = Probability that decoding based on C1 is correct,

y = Probability that
 (a) decoding based on C1 is correct and
 (b) at least one of R_{i-2}, R_{i-1} and R_i contains no errors,

δ_1 = Conditional probability that decoding based on C1 is correct given that R_{i-2}, R_{i-1}, R_i all contain errors.

Using a similar method as that for system A, it can be shown for system B that the probability of correctly obtaining I from R_{i-2}, R_{i-1} and R_i given that R_{i-2}, and R_{i-1} contain errors is given by [2]:

$$P_t = P_c + (1-P_c)\delta_1. \tag{7.21}$$

Further, for system B,

$$E[T]_B \le (P_c(1+2a_1)+a_2(2+a_1)+3a_3)\cdot\frac{1}{(1-a_1)^2} \tag{7.22}$$

where $a_1 = (1-P_c)^2(1-P_t)$,
$a_2 = (1-P_c)P_c$
and $a_3 = (1-P_c)^2P_t$. (7.23)

The probabilities δ_0 and y may be shown to be [2] ,

$$\delta_0 = \left[\sum_{j=0}^{t_1}\binom{3l'}{j}\varepsilon^j(1-\varepsilon)^{3l'-j}\right]^{n/l'}$$

and

$$y = (1-\varepsilon)^n\left[3\left\{\sum_{j=0}^{t_1}\binom{2l'}{j}\varepsilon^j(1-\varepsilon)^{2l'-j}\right\}^{n/l'} - 3(1-\varepsilon)^n\left\{\sum_{j=0}^{t_1}\binom{l'}{j}\varepsilon^j(1-\varepsilon)^{l'-j}\right\}^{n/l'} - 5(1-\varepsilon)^{2n}\right]. \tag{7.24}$$

Finally δ_1 for system B is given by the right hand side of (7.13).

Using equation (7.8) and $E\,[T\,]_A$ and $E\,[T\,]_B$, we may obtain the throughput of Systems A and B . The throughput of the actual GH-ARQ scheme is underbounded by the maximum of these two inferior throughputs.

Consider a GH-ARQ scheme of depth 2 employing an (8, 4, 3) KM code for error-correction. Figure 7.4 shows how the throughput of such a scheme varies with decreasing channel conditions.

As a comparision, Figure 7.4 also shows the throughput of Type-II hybrid ARQ systems using C1, with error-correcting capabilities $t_1 = 5, 10, 20$.

It should be noted that the throughput of the GH-ARQ scheme decreases slowly as it approaches 0.5. Precise explanations are difficult to give. However, since the receiver performs error-detection and error-correction upon the first retransmission, obviously the probability of further retransmissions will be greatly reduced. This reduction will result in an increased throughput. Provided, the probability of error-correction upon the first retransmission is high, throughput will remain close to 0.5.

KEY

1. Pure ARQ
2. C1 : $(2n, n)$ code derived from (8, 4, 3) KM code
3. C1 : $(2n, n)$ code, $t_1 = 5$
4. C1 : $(2n, n)$ code, $t_1 = 10$
5. C1 : $(2n, n)$ code, $t_1 = 20$

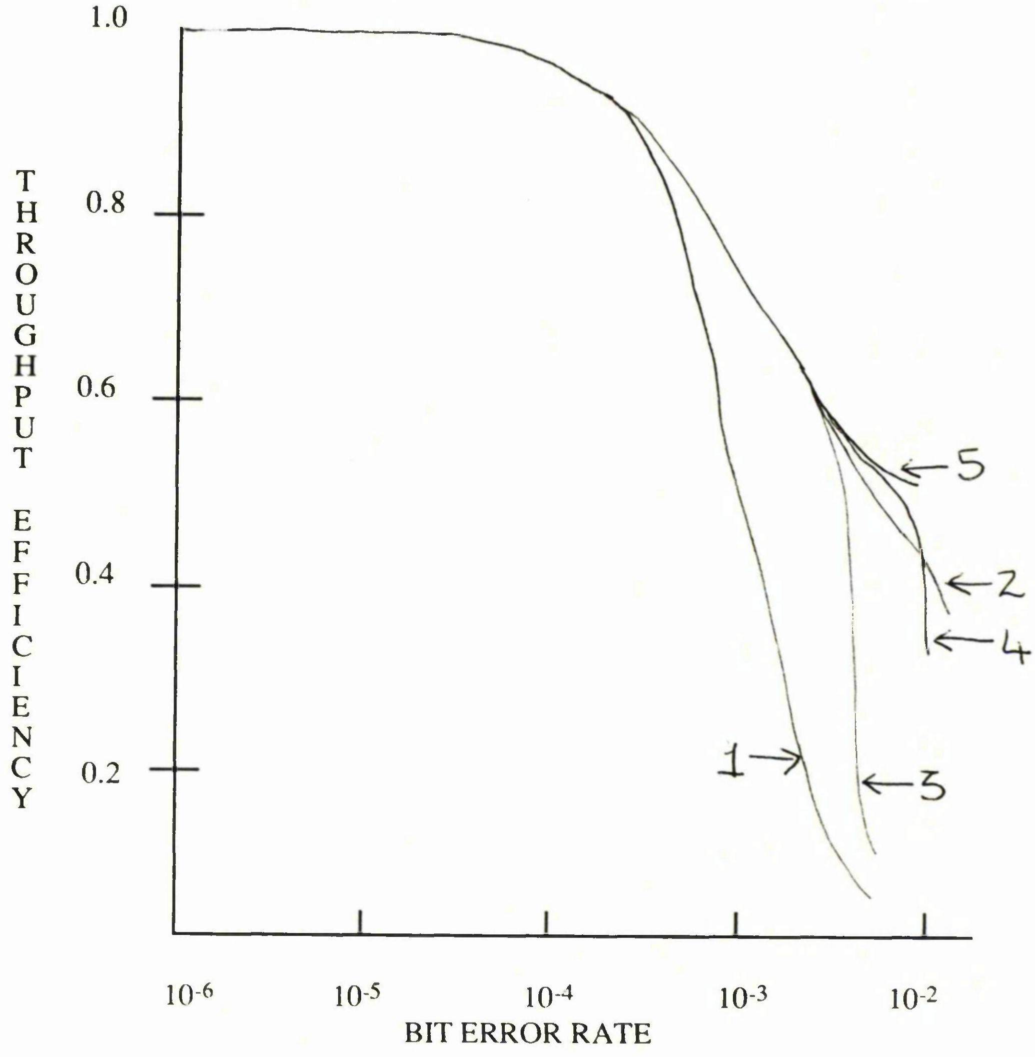


Figure 7.4 (i) - Throughput efficiency of the selective-repeat GH-ARQ schemes using depth 2 codes for error-correction ($n = 500$).

KEY

1. Pure ARQ
2. C1 : $(2n, n)$ code derived from (8, 4, 3) KM code
3. C1 : $(2n, n)$ code, $t_1 = 5$
4. C1 : $(2n, n)$ code, $t_1 = 10$
5. C1 : $(2n, n)$ code, $t_1 = 20$

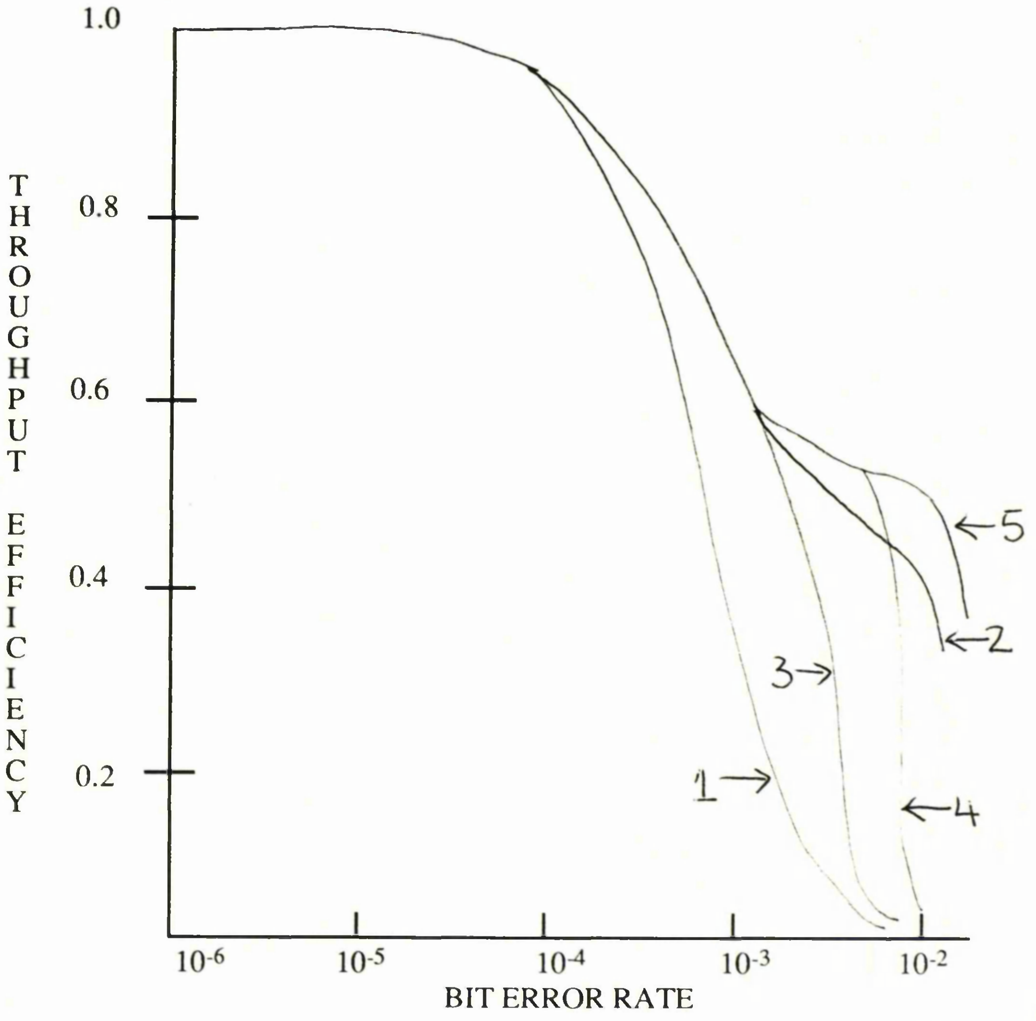


Figure 7.4 (ii) - Throughput efficiency of the selective-repeat GH-ARQ schemes using depth 2 codes for error-correction (n = 1000).

KEY

1. Pure ARQ
2. C1 : ($2n$, n) code derived from (8, 4, 3) KM code
3. C1 : ($2n$, n) code, $t_1 = 5$
4. C1 : ($2n$, n) code, $t_1 = 10$
5. C1 : ($2n$, n) code, $t_1 = 20$

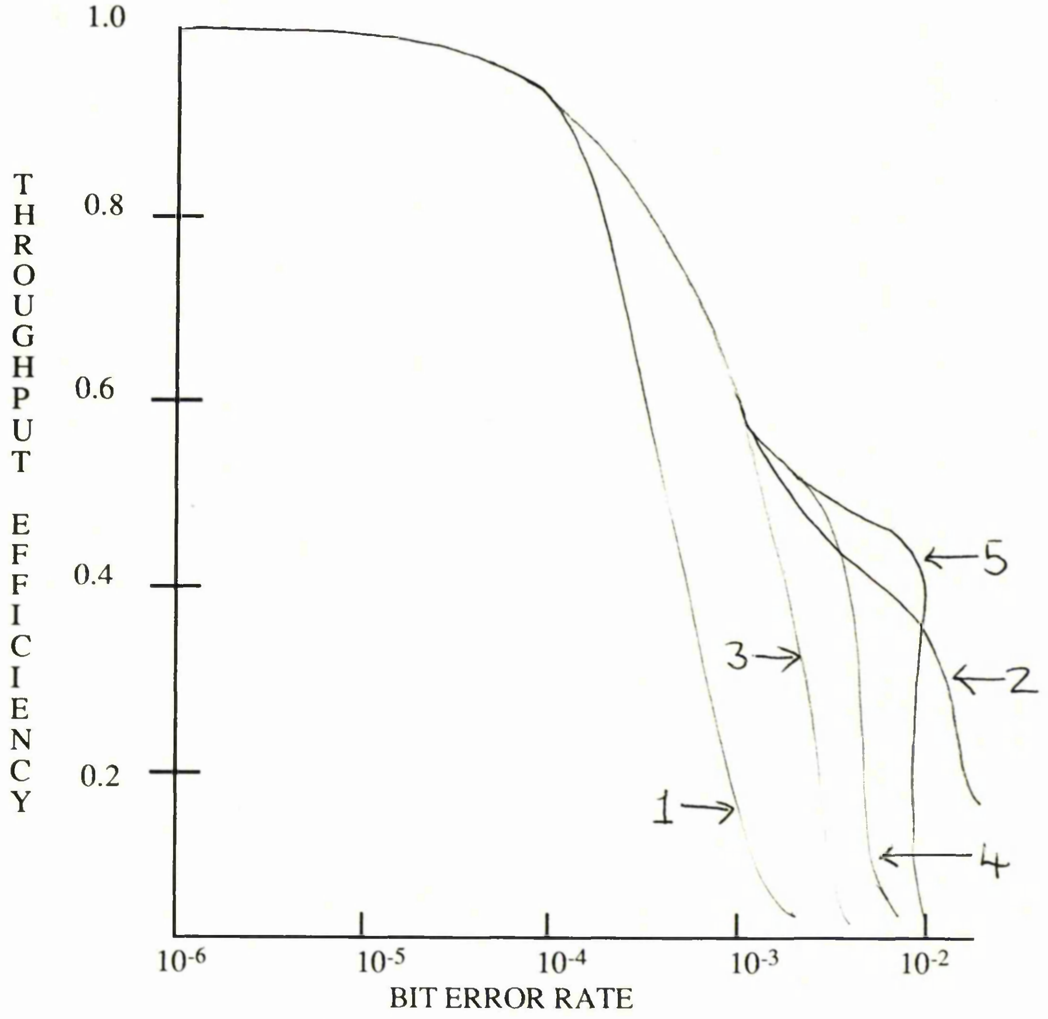


Figure 7.4 (iii) - Throughput efficiency of the selective-repeat GH-ARQ schemes using depth 2 codes for error-correction (n = 2000).

7.5.2 - Reliability of GH-ARQ Schemes

Type-II hybrid ARQ systems have been shown to provide the same order of reliability as ARQ systems [15]. We now wish to show that the GH-ARQ schemes offer the same order of reliabilty as ARQ systems.

For a GH-ARQ system, let E be the event that the receiver accepts a block containing undetectable errors. Then, the reliabilty of this system can be found via the probability of the event E, $\Pr[E]$.

Clearly,

$$\Pr[E] = \Pr[T_0{}^e] + \Pr[T_0{}^dE_1{}^e] + \Pr[T_0{}^dE_1{}^dE_2{}^e] + \ldots$$
$$\ldots + \Pr[T_0{}^dE_1{}^d\ldots E_{i-1}{}^dE_i{}^e] + \ldots$$

$$= \Pr\left[T_0{}^e\right] + \sum_{i=1}^{\infty} \Pr\left[T_0{}^d E_1{}^d \ldots E_{i-1}{}^d E_i{}^e\right] \tag{7.25}$$

Again, we are faced with the problem of computing joint probabilities. As an alternative, we aim to find an upperbound for each term in equation (7.25) above.

Recall, in a GH-ARQ scheme of depth m, a received vector R_i is a codeword iff $R_i{}^TH_i{}^{\sim T} = 0$; $i = 1, 2, \ldots, m$. Let Pe_i be the probability of undetected errors, associated with the error-detecting code, which has parity check matrix $H_i{}^{\sim}$ and let

$$P_e = \max(Pe_i;\, i = 1, 2, \ldots, m) \quad P_f = \min(Pe_i;\, i = 1, 2, \ldots, m).$$

Assuming that Pe_i ; $i = 1, 2, \ldots, m$ satisfy bound (7.1), P_e can be made arbitrarily small. Therefore,

$$\Pr[T_0{}^e] \le P_e$$

and

$$\Pr[B_i{}^e] \le P_e \qquad i = 1, 2, \ldots, m.$$

Again, let P_d be the probability of errors being detected in any transmission. Then,

P_d = 1- (Probability that errors do not occur) - (Probability that errors are undetected)
$\le 1 - P_c - P_f$.

We shall use the approximation

$$P_d \approx 1 - P_c - P_f. \tag{7.26}$$

Consider the term $\Pr[T_0{}^dE_1{}^d\ldots E_{i-1}{}^dE_i{}^e]$ of equation (7.25). Recall $E_i{}^d = B_i{}^dD_i{}^d$ and $E_i{}^e = B_i{}^e \cup B_i{}^dD_i{}^e$. Then,

$$\Pr[T_0{}^dE_1{}^d\ldots E_{i-1}{}^dE_i{}^e] \le \Pr[T_0{}^dB_1{}^d\ldots B_{i-1}{}^dE_i{}^e]$$
$$= \Pr[T_0{}^dB_1{}^d\ldots B_{i-1}{}^d]\Pr[E_i{}^e | T_0{}^dB_1{}^d\ldots B_{i-1}{}^d]$$
$$\le (P_d)^i \Pr[B_i{}^e \cup B_i{}^dD_i{}^e | T_0{}^d\ldots B_{i-1}{}^d]$$

by (7.6)

$$= (P_d)^i \{\Pr[B_i^e] + \Pr[B_i^d]\Pr[D_i^e | T_0^d \,..\, B_{i-1}^d]\}$$
$$\leq (P_d)^i \{P_e + P_d.\Pr[D_i^e | T_0^d B_1^d \,.\,. B_i^d]\}. \qquad (7.27)$$

$\Pr[D_i^e | T_0^d B_1^d \,..\, B_i^d]$ denotes the probability that the block recovered by decoding blocks up to the i th block will contain undetectable errors given that all transmissions resulted in detectable errors. Since the decoded data block is checked for the presence of errors at every retransmission, we must have,

$$\Pr[D_i^e | T_0^d B_1^d \,..\, B_i^d] \leq P_e. \qquad (7.28)$$

By substituting (7.28) into equation (7.27), we get,

$$\Pr[T_0^d E_1^d \,..\, E_{i-1}^d E_i^e] \leq (P_d)^i (P_e + P_d P_e)$$
$$= P_d^i (1+P_d) P_e. \qquad (7.29)$$

And after, substituting (7.29) into (7.25) we obtain,

$$\Pr[E\,] \leq P_e + \sum_{i=1}^{\infty} P_d^{\,i}\,(1 + P_d)P_e$$
$$= P_e + \frac{P_d\,(1+P_d)P_e}{1 - P_d}$$
$$= \frac{P_e\,(1-P_d) + P_d\,(1+P_d)P_e}{P_c + P_f}$$
$$= \frac{P_e - P_e P_d + P_e P_d + P_e P_d^{\,2}}{P_c + P_f}$$
$$= \left(1+P_d^{\,2}\right).\frac{P_e}{P_c + P_f}$$
$$= \left(1+P_d^{\,2}\right).\frac{P_c + P_e}{P_c + P_f}.\frac{P_e}{P_c + P_e}$$
$$= \left(1+P_d^{\,2}\right).\left(1+\frac{P_e - P_f}{P_c + P_f}\right).\left(\frac{P_e}{P_c + P_e}\right) \qquad (7.30)$$

For GH-ARQ systems, $P_d < 1$, $P_c \gg P_e$ and $P_c \gg P_f$. Therefore,

$$1+P_d^2 < 2 \qquad (7.31)$$

and

$$1+\frac{P_e - P_f}{P_c + P_f} \cong 1\,. \qquad (7.32)$$

Using (7.31) and (7.32), we are able to simplify equation (7.30) to obtain the approximation,

$$\Pr[E\,] \leq 2.1.\frac{P_e}{P_c + P_e} \,.$$

Consider a pure ARQ-system which employs the error-detecting code C0. Suppose, for this system, the probability of undetected error is P_e. Then, the probability of event E ($\Pr[E]_{\mathrm{ARQ}}$) can be shown to be [4] ,

$$\Pr[E\,]_{ARQ} = \frac{P_e}{P_c + P_e} \,.$$

And, so

$$\Pr[E\,] \leq 2\,\Pr[E\,]_{\mathrm{ARQ}}.$$

Hence, the GH-ARQ system provides the same order of reliability as a pure ARQ scheme, using the same error-detecting code C0, with the same probability of undetected error P_e.

7.6 - A Discussion on the Suitability of KM Codes to GH-ARQ Schemes

We have illustrated above, how KM-Codes can be incorporated into GH-ARQ schemes. Further, it has been shown that the throughput of such schemes reaches the required standard, even over poor channels. The reliability is of the same order as that provided by pure ARQ schemes.

It was noted previously, that KM-codes are simple to implement. The decoding is straightforward and does not involve excessive overheads.

We may conclude that KM-Codes are an ideal candidate for error-correction in GH-ARQ schemes.

CHAPTER 8 - Summary

8.1 - Summary

This work has illustrated a way of obtaining a recently introduced class of linear error-correcting codes, namely KM codes, from a particular bilinear form. It has been shown that the dual of this bilinear form may be computed as the aperiodic convolution of two sequences. The generator matrix of the corresponding KM code was found to be a direct result of this computation. Further, the length of this code is equal to the multiplicative complexity of the computation.

Two algorithms for the aperiodic convolution of sequences were described, although more interest was shown in the Convolution Algorithm based on the CRT. It was noted that a great number of codes may be obtained from this method. A few examples, illustrate some of the unique properties of these codes.

The encoding/decoding procedures were also studied and the limitations of Krishna and Morgera's decoding scheme were discussed. We observed that the minimum distance (hence the error-correcting capability) and the length of the code can be varied easily with little change to the encoding/decoding configuration. Moreover, the complexity of the decoding algorithm is proportional to the number of relatively prime factors of $P(u)$. A large number of small degree factors results in a shorter length code but the decoding of this reduced code is more complex.

Although, our main concern was binary codes, an example of these codes over $GF(3)$ was given. We noted that it is possible to obtain these codes over $GF(p^m)$ and in particular $GF(2^m)$.

An error control scheme, GH-ARQ was described and a GH-ARQ scheme based on KM codes was studied. This scheme can provide high throughput even under poor channel conditions and offers the same order of reliability as pure ARQ schemes. Since the decoding procedures of small codes form the basis of the overall decoding procedure for these codes, the decoder can process parts of the received vector independently. This reduces the overall decoding time and is of particular importance in high data rate communications systems. We concluded that KM codes perform well in GH-ARQ schemes. Their unique block structure means they are ideal for providing adaptive error control.

8.2 - Future Topics of Research

These include :

(1) Generalise the procedures to obtain efficient algorithms and the associated linear codes over GF(2^m).
(2) All the codes in this work originated from a particular bilinear form. Study the possibilty of other classes of codes which may perhaps be obtained from a different bilinear form.
(3) Study the complexity of decoding and find the 'balance' between this complexity and the length of codes.
(4) Study the performance of the GH-ARQ scheme when

(a) The buffer is of finite size,
(b) The feedback channel is noisy.

(5) Study the possibility of using other known codes in GH-ARQ schemes.

REFERENCES

[1] W.W. Peterson & E.J. Weldon Jr., Error-Correcting Codes, The MIT Press, 1978.

[2] H. Krishna, Computational Complexity of Bilinear Forms : Algebraic Coding Theory and Applications to Digital Communications Systems, New York : Springer-Verlag, 1987.

[3] H. Krishna & S.D. Morgera, A New Error Control Scheme for Hybrid ARQ Systems, IEEE Trans. on Comms., Vol. COM-35, No. 10, pp981-990, Oct. 1987.

[4] S. Lin & D.J. Costello Jr., Error Control Coding : Fundamentals and Applications, Prentice Hall, 1983.

[5] S. Lin & P.S. Yu, A Hybrid ARQ Scheme with Parity Retransmissions for Error Control of Satellite Channels, IEEE Trans. on Comms., Vol. COM-30, No. 7, pp1706-1719, July 1982.

[6] J. Hopcroft & J. Musinski, Duality Applied to the Complexity of Matrix Multiplications and Other Bilinear Forms, SIAM J. Computing, Vol. 2, No. 3, pp159-173, Sept. 1973.

[7] A. Lempel & S. Winograd, A New Approach to Error-Correcting Codes, IEEE Trans. on Info. Theory, Vol. IT-23, No. 4, pp 503-508, July 1977.

[8] S. Winograd, On the number of Multiplications Necessary to Compute Certain Functions, Comms. on Pure and Applied Maths, Vol. XXIII, pp 165-179,1970.

[9] C.M. Fiduccia, Fast Matrix Multiplication, Proc. 3rd Ann. Acm. Symp. of Theory of Computing, Shaker Heights, pp 45-49, May 1971.

[10] J.L. Dornstetter, On the Computation of the Product of Two Polynomials Over a Finite Field, International Symposium on Information Theory, St. Jovite, Canada, Sept. 1983.

[11] S. Winograd, Some Bilinear Forms Whose Multiplicative Complexity Depends on the Field of Constants, Mathematical Systems Theory 10, pp 169-180, 1977.

[12] M.D. Wagh & S.D. Morgera, A New Structured Design Method for Convolutions over Finite Fields, Part I, IEEE Trans. on Information Theory, Vol. IT-29, No. 4, pp 583-595, July 1983.

[13] T. Kasami, T. Klove & S. Lin, Linear Block Codes for Error-Detection, IEEETrans. on Info. Theory, Vol. IT-29, No. 1, Jan. 1983.

[14] P.L. Meyer, Introductory Probability and Statistical Applications, Reading (Mass),1966.

[15] Y.M. Wang & S. Lin, A Modified Selective-Repeat Type-II Hybrid ARQ System and its Performance Analysis, IEEE Trans. on Comms., Vol. COM-31, No. 5, pp 593-607, May 1983.

APPENDIX A

In this appendix, we wish to show that given ϑ, a system of k bilinear forms in the form (3.6), it is possible to express Φ, the P-dual of ϑ in the form (3.7).

ϑ in the form (3.6) is given by,

$$\vartheta = X\,\mathbf{y} = \begin{bmatrix} x_0 & x_1 & . & . & . & x_{d-1} \\ x_1 & x_2 & . & . & . & x_d \\ & . & & & . & \\ & . & & & . & \\ x_{k-1} & x_k & . & . & . & x_{k+d-2} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ . \\ . \\ . \\ y_{d-1} \end{bmatrix}. \qquad \text{(A - 1)}$$

Expanding (A-1), we obtain

$$\vartheta = \begin{bmatrix} x_0y_0 + x_1y_1 + \ldots + x_{d-1}y_{d-1} \\ x_1y_0 + x_2y_1 + \ldots + x_d\,y_{d-1} \\ . \\ . \\ . \\ x_{k-1}y_0 + x_k\,y_1 + \ldots + x_{k+d-2}y_{d-1} \end{bmatrix}. \qquad \text{(A- 2)}$$

Looking at (A-2), it is obvious that ϑ may be computed using only the following kd terms, $x_{i+j}y_j$, $i = 0,1,\ldots,k-1$, $j = 0,1,\ldots,d-1$. Thus, ϑ may be expressed as $C(Ax \times By)$ where

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

$$A = \begin{bmatrix}
1 & 0 & \cdot & \cdot & 0 & 0 & \cdot & \cdot & \cdot & 0 & 0 \\
0 & 1 & 0 & \cdot & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & 0 \\
 & & \cdot & & & & & & & & \cdot \\
 & & & \cdot & & & & & & & \cdot \\
0 & \cdot & \cdot & 0 & 1 & 0 & & & & & \cdot \\
0 & 1 & 0 & \cdot & \cdot & 0 & & & & & \cdot \\
\cdot & 0 & 1 & 0 & \cdot & 0 & & & & & \cdot \\
 & & & \cdot & & & & & & & \cdot \\
 & & & & \cdot & & & & & & \cdot \\
0 & 0 & \cdot & \cdot & 0 & 1 & 0 & & & & \cdot \\
\cdot & & & & & & & & & & \cdot \\
\cdot & & & & & & & & & & \cdot \\
\cdot & & & & & & & & & & \cdot \\
\cdot & & & & & & & & & & \cdot \\
\cdot & & & & & & & & & & \cdot \\
\cdot & & & & & 1 & 0 & \cdot & \cdot & 0 & 0 \\
\cdot & & & & & 0 & 1 & 0 & \cdot & 0 & 0 \\
\cdot & & & & & & & \cdot & & & \\
\cdot & & & & & & & & \cdot & & \\
\cdot & & & & & 0 & \cdot & \cdot & 0 & 1 & 0 \\
\cdot & & & & & 0 & 1 & 0 & \cdot & \cdot & 0 \\
\cdot & & & & & & 0 & 1 & 0 & \cdot & 0 \\
\cdot & & & & & & & & \cdot & & \\
\cdot & & & & & & & & & \cdot & \\
0 & 0 & \cdot & \cdot & \cdot & 0 & 0 & \cdot & \cdot & \cdot & 1
\end{bmatrix}
\quad \text{and} \quad
B = \begin{bmatrix}
1 & 0 & \cdot & \cdot & 0 \\
0 & 1 & 0 & \cdot & 0 \\
 & & \cdot & & \\
 & & & \cdot & \\
0 & \cdot & \cdot & \cdot & 1 \\
1 & 0 & \cdot & \cdot & 0 \\
0 & 1 & 0 & \cdot & 0 \\
 & & \cdot & & \\
 & & & \cdot & \\
0 & \cdot & \cdot & \cdot & 1 \\
 & \cdot & & & \cdot \\
 & \cdot & & & \cdot \\
 & \cdot & & & \cdot \\
 & \cdot & & & \cdot \\
 & \cdot & & & \cdot \\
1 & 0 & \cdot & \cdot & 0 \\
0 & 1 & 0 & \cdot & 0 \\
 & & \cdot & & \\
 & & & \cdot & \\
0 & \cdot & \cdot & \cdot & 1 \\
1 & 0 & \cdot & \cdot & 0 \\
0 & 1 & 0 & \cdot & 0 \\
 & & \cdot & & \\
 & & & \cdot & \\
0 & \cdot & \cdot & \cdot & 1
\end{bmatrix}.$$

Recall, Φ, the P-dual of ϑ may be expressed in terms of matrices A, B and C. Infact, if $z = (z_0, z_1, \ldots, z_{k-1})$ then

$$\Phi = A^T(C^T z \times By)$$

Multiplying out these matrices and re-arranging terms, it is possible to express Φ in the form (3.7)

$$\Phi = \begin{bmatrix} z_0 y_0 \\ z_1 y_0 + z_0 y_1 \\ z_2 y_0 + z_1 y_1 + z_0 y_2 \\ . \\ . \\ z_{k-1} y_0 + z_{k-2} y_1 + .. + z_{k-d} y_{d-1} \\ z_{k-1} y_1 + z_{k-2} y_2 + .. + z_{k-d+1} y_{d-1} \\ . \\ . \\ z_{k-1} y_{d-2} + z_{k-2} y_{d-1} \\ z_{k-1} y_{d-1} \end{bmatrix}$$

$$= \begin{bmatrix} z_0 & 0 & 0 & . & . & 0 & 0 \\ z_1 & z_0 & 0 & . & . & 0 & 0 \\ z_2 & z_1 & z_0 & . & . & 0 & 0 \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ z_{k-1} & z_{k-2} & z_{k-3} & . & . & . & . \\ 0 & z_{k-1} & z_{k-2} & . & . & . & . \\ 0 & 0 & z_{k-1} & . & . & . & . \\ 0 & 0 & 0 & . & . & . & . \\ . & . & . & . & . & z_0 & 0 \\ . & . & . & . & . & . & z_0 \\ . & . & . & . & . & . & . \\ . & . & . & . & . & z_{k-1} & z_{k-2} \\ 0 & 0 & 0 & 0 & 0 & 0 & z_{k-1} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ . \\ . \\ . \\ y_{d-1} \end{bmatrix} .$$

Hence result.

Non-commutative Algorithms for Small Degree Polynomial Multiplication

Algorithm Multi 1 is due to Karatsuba. Multi 2B can be found in [12] while Multi 2A may be obtained from Multi 2B by making suitable alterations. Algorithm Multi 3B was derived from Multi 2B using multi-D techniques. A few alterations result in Multi 3A. Algorithm 4A and 4B were derived by the author using 'trial and error'. Using multi-D techniques, Multi 5B was obtained from Multi 1 and Multi 2B. Finally, making some changes Multi 5A was obtained.

Algorithm Multi 0

Degree 0 $z_0.y_0 = \phi_0$

Computation : direct 1 multiplication

Algorithm Multi 1

Degree 1 $(z_0+z_1u).(y_0+y_1u)$
$= \phi_0+\phi_1u+\phi_2u^2$

Computation : 3 multiplications

Let $m_0 = z_0.y_0$
$m_1= z_1.y_1$
$m_2=(z_0+z_1).(y_0+y_1)$

then

$\phi_0=m_0$

$\phi_1=-m_0-m_1+m_2$

$\phi_2=m_1$

Algorithm Multi 2A

Degree 2 $(z_0+z_1u+z_2u^2).(y_0+y_1u+y_2u^2)$ modulo u^3
$= \phi_0+\phi_1u+\phi_2u^2$

Computation : 5 multiplications

Let $m_0=z_0.y_0$
$m_1=z_1.y_1$
$m_2=z_2.y_2$
$m_3=(z_0+z_1).(y_0+y_1)$
$m_4=(z_0+z_2).(y_0+y_2)$

then

$\phi_0=m_0$

$\phi_1=-m_0-m_1+m_3$

$\phi_2=-m_0+m_1-m_2+m_4$

Algorithm Multi 2B

Degree 2 $(z_0+z_1u+z_2u^2).(y_0+y_1u+y_2u^2)$
$= \phi_0+\phi_1u+\phi_2u^2+\phi_3u^3+\phi_4u^4$

Computation : 6 multiplications

Let $m_0=z_0.y_0$
$m_1=z_1.y_1$
$m_2=z_2.y_2$
$m_3=(z_0+z_1).(y_0+y_1)$
$m_4=(z_1+z_2).(y_1+y_2)$
$m_5=(z_0+z_2).(y_0+y_2)$

then

$\phi_0=m_0$
$\phi_1=-m_0-m_1+m_3$
$\phi_2=-m_0+m_1-m_2+m_5$
$\phi_3=-m_1-m_2+m_4$
$\phi_4=m_2$

Algorithm Multi 3A

Degree 3 $(z_0+z_1u+z_2u^2+z_3u^3).(y_0+y_1u+y_2u^2+y_3u^3)$ modulo u^4
$= \phi_0+\phi_1u+\phi_2u^2+\phi_3u^3$

Computation : 8 multiplications

Let $m_0=z_0.y_0$
$m_1=z_1.y_1$
$m_2=z_2.y_2$
$m_3=z_3.y_3$
$m_4=(z_0+z_1).(y_0+y_1)$
$m_5=(z_0+z_2).(y_0+y_2)$
$m_6=(z_1+z_2).(y_1+y_2)$
$m_7=(z_0+z_3).(y_0+y_3)$

then

$\phi_0=m_0$
$\phi_1=-m_0-m_1+m_4$
$\phi_2=-m_0-m_2+m_5$
$\phi_3=-m_0-m_1-m_2-m_3+m_6+m_7$

Algorithm Multi 3B

Degree 3 $(z_0+z_1u+z_2u^2+z_3u^3).(y_0+y_1u+y_2u^2+y_3u^3)$
$= \phi_0+\phi_1u+\phi_2u^2+\phi_3u^3+\phi_4u^4+\phi_5u^5+\phi_6u^6$

Computation : 9 multiplications

Let $m_0=z_0.y_0$
$m_1=z_1.y_1$
$m_2=(z_0+z_1).(y_0+y_1)$
$m_3=z_2.y_2$
$m_4=z_3.y_3$
$m_5=(z_2+z_3).(y_2+y_3)$
$m_6=(z_0+z_2).(y_0+y_2)$

$m_7=(z_1+z_3).(y_1+y_3)$
$m_8=(z_0+z_1+z_2+z_3).(y_0+y_1+y_2+y_3)$

then

$\phi_0=m_0$

$\phi_1=-m_0-m_1+m_2$

$\phi_2=-m_0+m_1-m_3+m_6$

$\phi_3=m_0+m_1-m_2+m_3+m_4-m_5-m_6-m_7+m_8$

$\phi_4=-m_1+m_3-m_4+m_7$

$\phi_5=-m_3-m_4+m_5$

$\phi_6=m_4$

Algorithm multi4A

Degree 4 $(z_0+z_1u+z_2u^2+z_3u^3+z_4u^4).(y_0+y_1u+y_2u^2+y_3u^3+y_4u^4)$ modulo u^5
$= \phi_0+\phi_1u+\phi_2u^2+\phi_3u^3+\phi_4u^4$

Computation: 11 multiplications

Let, $m_0=z_0.y_0$
$m_1=z_1.y_1$
$m_2=(z_0+z_1).(y_0+y_1)$
$m_3=z_2.y_2$
$m_4=(z_0+z_2).(y_0+y_2)$
$m_5=z_3.y_3$
$m_6=(z_0+z_3).(y_0+y_3)$
$m_7=(z_1+z_2).(y_1+y_2)$
$m_8=z_4.y_4$
$m_9=(z_0+z_4).(y_0+y_4)$
$m_{10}=(z_1+z_3).(y_1+y_3)$

then

$\phi_0=m_0$

$\phi_1=-m_0-m_1+m_2$

$\phi_2=-m_0+m_1-m_3+m_4$

$\phi_3=-m_0-m_1-m_3-m_5+m_6+m_7$

$\phi_4=-m_0-m_1+m_3-m_5-m_8+m_9+m_{10}$

Algorithm multi4B

Degree 4 $(z_0+z_1u+z_2u^2+z_3u^3+z_4u^4).(y_0+y_1u+y_2u^2+y_3u^3+y_4u^4)$
$= \phi_0+\phi_1u+\phi_2u^2+\phi_3u^3+\phi_4u^4+\phi_5u^5+\phi_6u^6+\phi_7u^7+\phi_8u^8$

Computation: 14 multiplications

Let, $m_0=z_0.y_0$
$m_1=z_1.y_1$
$m_2=(z_0+z_1).(y_0+y_1)$
$m_3=z_2.y_2$
$m_4=(z_0+z_2).(y_0+y_2)$
$m_5=z_3.y_3$
$m_6=(z_1+z_3).(y_1+y_3)$
$m_7=(z_2+z_3).(y_2+y_3)$

$m_8=(z_0+z_1+z_2+z_3).(y_0+y_1+y_2+y_3)$
$m_9=z_4.y_4$
$m_{10}=(z_0+z_4).(y_0+y_4)$
$m_{11}=(z_1+z_4).(y_1+y_4)$
$m_{12}=(z_2+z_4).(y_2+y_4)$
$m_{13}=(z_3+z_4).(y_3+y_4)$

then

$\phi_0=m_0$

$\phi_1=-m_0-m_1+m_2$

$\phi_2=-m_0+m_1-m_3+m_4$

$\phi_3=m_0+m_1-m_2+m_3-m_4+m_5-m_6-m_7+m_8$

$\phi_4=-m_0-m_1+m_3-m_5+m_6-m_9+m_{10}$

$\phi_5=-m_1-m_3-m_5+m_7-m_9+m_{11}$

$\phi_6=-m_3+m_5-m_9+m_{12}$

$\phi_7=-m_5-m_9+m_{13}$

$\phi_8=m_9$

Algorithm multi5A

Degree 5 $(z_0+z_1u+z_2u^2+z_3u^3+z_4u^4+z_5u^5).(y_0+y_1u+y_2u^2+y_3u^3+y_4u^4+y_5u^5)$ modulo u^6

$= \phi_0+\phi_1u+\phi_2u^2+\phi_3u^3+\phi_4u^4+\phi_5u^5$

Computation: 14 multiplications

Let

$m_0=z_0.y_0$
$m_1=z_1.y_1$
$m_2=(z_0+z_1).(y_0+y_1)$
$m_3=z_2.y_2$
$m_4=(z_0+z_2).(y_0+y_2)$
$m_5=z_3.y_3$
$m_6=z_4.y_4$
$m_7=(z_2+z_3).(y_2+y_3)$
$m_8=(z_1+z_3).(y_1+y_3)$
$m_9=(z_0+z_1+z_2+z_3).(y_0+y_1+y_2+y_3)$
$m_{10}=(z_0+z_4).(y_0+y_4)$
$m_{11}=z_5.y_5$
$m_{12}=(z_0+z_5).(y_0+y_5)$
$m_{13}=(z_1+z_4).(y_1+y_4)$

then

$\phi_0=m_0$

$\phi_1=-m_0-m_1+m_2$

$\phi_2=-m_0+m_1-m_3+m_4$

$\phi_3=m_0+m_1-m_2+m_3-m_4+m_5-m_7-m_8+m_9$

$\phi_4=-m_0-m_1+m_3-m_5-m_6+m_8+m_{10}$

$\phi_5=-m_0-m_1-m_3-m_5-m_6+m_7-m_{11}+m_{12}+m_{13}$

Algorithm multi5B

Degree 5 $(z_0+z_1u+z_2u^2+z_3u^3+z_4u^4+z_5u^5).(y_0+y_1u+y_2u^2+y_3u^3+y_4u^4+y_5u^5)$

$=\phi_0+\phi_1u+\phi_2u^2+\phi_3u^3+\phi_4u^4+\phi_5u^5+\phi_6u^6+\phi_7u^7+\phi_8u^8+\phi_9u^9+\phi_{10}u^{10}$

Computation: 18 multiplications

Let

$m_0=z_0.y_0$

$m_1=z_1.y_1$

$m_2=z_2.y_2$

$m_3=(z_0+z_1).(y_0+y_1)$

$m_4=(z_1+z_2).(y_1+y_2)$

$m_5=(z_0+z_2)(y_0+y_2)$

$m_6=z_3.y_3$

$m_7=z_4.y_4$

$m_8=z_5.y_5$

$m_9=(z_3+z_4).(y_3+y_4)$

$m_{10}=(z_4+z_5).(y_4+y_5)$

$m_{11}=(z_3+z_5).(y_3+y_5)$

$m_{12}=(z_0+z_3).(y_0+y_3)$

$m_{13}=(z_1+z_4).(y_1+y_4)$

$m_{14}=(z_2+z_5).(y_2+y_5)$

$m_{15}=(z_0+z_1+z_3+z_4).(y_0+y_1+y_3+y_4)$

$m_{16}=(z_1+z_2+z_4+z_5).(y_1+y_2+y_4+y_5)$

$m_{17}=(z_0+z_2+z_3+z_5).(y_0+y_2+y_3+y_5)$

then

$\phi_0=m_0$

$\phi_1=-m_0-m_1+m_3$

$\phi_2=-m_0+m_1-m_2+m_5$

$\phi_3=-m_0-m_1-m_2+m_4-m_6+m_{12}$

$\phi_4=m_0+m_1-m_3+m_6+m_7-m_9-m_{12}-m_{13}+m_{15}$

$\phi_5=m_0-m_1+m_2-m_5+m_6-m_7+m_8-m_{11}-m_{12}+m_{13}-m_{14}+m_{17}$

$\phi_6=m_1+m_2-m_4+m_6+m_7+m_8-m_{10}-m_{13}-m_{14}+m_{16}$

$\phi_7=-m_2-m_6-m_7-m_8+m_9+m_{14}$

$\phi_8=-m_6+m_7-m_8+m_{11}$

$\phi_9=-m_7-m_8+m_{10}$

$\phi_{10}=m_8$

APPENDIX C

Using the same notation as Lemma 6.5, we aim to show that

$$T_p = Zy \quad .$$

where

$$Z = (z \mid C_p\, z \mid C_p^2\, z \mid \ldots \mid C_p^{\alpha-1} z\,).$$

This may be proved for any $\alpha > 1$, however we shall simply show that the result holds for $\alpha = 3$. Then, $Z(u) = z_0+z_1u+z_2u^2$, $Y(u) = y_0+y_1u+y_2u^2$ and $P(u) = a_0+a_1u+a_2u^2+u^3$. Further,

$$\begin{aligned}\Phi(\mathrm{u}) &\equiv Z(u)Y(u) \text{ modulo } P(u)\\ &\equiv (z_0y_0-a_0z_2y_1-a_0z_1y_2+a_0a_2z_2y_2) +\\ &\qquad (z_1y_0+z_0y_1-a_1z_2y_1-a_1z_1y_2-a_0z_2y_2+a_1a_2z_2y_2)u +\\ &\qquad\qquad (z_2y_0+z_1y_1-a_2z_2y_1+z_0y_2-a_2z_1y_2-a_1z_2y_2+a_2^2z_2y_2)u^2.\end{aligned}$$

Recall, T_p is the system of bilinear forms given by the coefficients of $\Phi(u)$.
i.e.

$$T_p = \begin{bmatrix} z_0y_0 - a_0z_2y_1 - a_0z_1y_2 + a_0a_2z_2y_2 \\ z_1y_0 + z_0y_1 - a_1z_2y_1 - a_1z_1y_2 - a_0z_2y_2 + a_1a_2z_2y_2 \\ z_2y_0 + z_1y_1 - a_2z_2y_1 + z_0y_2 - a_2z_1y_2 - a_1z_2y_2 + a_2^2z_2y_2 \end{bmatrix}$$

$$= \begin{bmatrix} z_0 & -a_0z_2 & -a_0z_1 + a_0a_2z_2 \\ z_1 & z_0 - a_1z_2 & -a_0z_2 - a_1z_1 + a_1a_2z_2 \\ z_2 & z_1 - a_2z_2 & z_0 - a_2z_1 - a_1z_2 + a_2^2z_2 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} .$$

Now,

$$C_p = \begin{bmatrix} 0 & 0 & -a_0 \\ 1 & 0 & -a_1 \\ 0 & 1 & -a_2 \end{bmatrix}$$

so that,

$$C_p\,.\mathbf{z} = \begin{bmatrix} -a_0z_2 \\ z_0 - a_1z_2 \\ z_1 - a_2z_2 \end{bmatrix} \text{ and } C_p^{\,2}.\,\mathbf{z} = \begin{bmatrix} -a_0z_1 + a_0a_2z_2 \\ -a_1z_1 - a_0z_2 + a_1a_2z_2 \\ z_0 - a_2z_1 - a_1z_2 + a_2^2z_2 \end{bmatrix}.$$

Clearly,

$$T_p = (z \mid C_pz \mid C_p^2z)y.$$

There follows a progam (in THINK Pascal) which I have written to obtain the generator matrix of certain binary KM codes.

The user is asked to enter starting values for k and d, the amount by which k and d are to be increased after each stage is complete (*kstep*, *dstep*) and finally the *limit.*

For the initial values of k and d, $N = k+d-1$ is computed. D, the degree of the reducing polynomial $P(u)$, is set to N and s , the number of wraparounds, is set to zero. The generator matrix, the actual minimum distance and the burst error-detection capability of the corresponding KM code are found. Then D is reduced by one and s increased by one and the procedure is repeated until $s > min(6, k, d)$. At this point, $k := k + kstep$, $d := d + dstep$, N is updated and provided $N < limit$, the whole process is repeated for these new values of k and d. The program terminates when N is no longer less than the *limit.*

Recall, when deriving the generator matrix of a (k, d) KM code, we reduced the polynomials $Z(u)$ and $Y(u)$ of degree k-1 and d -1 respectively by each $P_i(u)$; $i = 1, 2, . ., t$. The corresponding reduced polynomials $Z_i(u)$ and $Y_i(u)$ were then multiplied together using the algorithms given in Appendix B. However, when $\deg P_i(u) > min(k, d)$, these algorithms cannot be used since the reduced polynomials $Z_i(u)$ and $Y_i(u)$ will not be of the same degree. Under these circumstances, the program states that it is not able to compute the corresponding KM code.

Suppose, the overall calculation, including the wraparound has multiplicative complexity n, then the multiplications $m_0, . ., m_{n-1}$ are sufficient to compute $\Phi(u) = Z(u)Y(u)$. Recall, $\Phi(u) = P(Qz \times Ry)$, where Q^T is the generator matrix of the corresponding KM code. Looking closely at how the matrix C is formed, it is clear that C consists of n columns where column i corresponds to m_i, $i = 0, 1, . ., n-1$. In fact, if $m_i = (a_0z_0+a_1z_1+. . +a_{k-1}z_{k-1}).(b_0y_0+b_1y_1+ . . +b_{d-1}y_{d-1})$ then the i th column of C would be $(a_0, a_1, . . , a_{k-1})^T$. i.e. C is controlled by the coefficients of the reduced polynomial $Z_i(u)$. Since, for the purposes of this program, we are only interested in the matrix C, the above observation is taken into account.

A linked list is used to store the m_i's and the procedure ProduceMatrix goes through this list and attains the corresponding columns of the generator matrtix C.

It was stated previously that if the reducing polynomial $P(u) = \prod_{i=1}^{t} P_i(u)$ where $\deg[P(u)]=D$, is chosen such that the polynomials $P_i(u)$ are coprime and of least possible degree, then the corresponding KM code should be the shortest possible KM code for this value of D. We will refer to this choice of polynomial $P(u)$ as the *minimal reducing polynomial.* I have designed an algorithm which will find, what I believe to be a minimal reducing polynomial of degree D (procedure ComputePolyP). The polynomials $P_1(u), . ., P_t(u)$ are stored in a linked list of type 'PList'.

In addition to computing the generator matrix, the actual minimum distance is found. As you would expect, the algorithm finds every possible non-zero codeword and its distance. The actual minimum distance is simply, the least value among these distances.

Finally, I have designed an algorithm which determines the burst error detection capability of the code. It is based on the idea suggested in Section 6.5.

The results are delivered to a text file, where they can be kept for future reference.

```
program KMCODES;
const
    degmax = 50;{max value of N=k+d-1}
    matrixmax = 125;{max length of any code generated}
type
    binary = 0..1;
    degtype = 0..degmax;
    PArray = array[0..degmax] of binary;
    Prec = record
            P: PArray;
            degP: degtype;
        end;
{a record to store a reducing poly, P_i(u), in binary form and its degree}
    PolyPtr = ^PList;
    PList = record
            PolyP: Prec;
            link: PolyPtr
        end;
{a linked list stores the reducing polys, P_1(u),..,P_t(u),where P(u) = ∏_{i=1}^{t} P_i(u) }
{and deg[P(u)]=D}
    Polys = array[0..degmax] of Prec;
    PolyArray = array[0..degmax] of array[0..degmax] of binary;
    Polyrec = record
                poly: PolyArray;
                deg: degtype
            end;
{a record to store the poly Z (u)=z_0+z_1u+. . +z_{k-1}u^{k-1} of degree k}
    multi = array[0..degmax] of binary;{stores one multiplication}
    multiptr = ^multinode;
    multinode = record
                m: multi;
                next: multiptr
            end;
{a linked list stores all the multiplications }
    SubCodeArray = array[0..matrixmax] of binary;
    Matrix = array[0..matrixmax] of SubCodeArray;
    MatrixRange = 0..matrixmax;
    wraprange = 0..7;{range of values s - the number of wraparounds }
    reducerecord = record
                RAPoly: PolyPtr;
                maxdeg: degtype
            end;
ReduceArray = array[1..degmax] of reducerecord;
```

```
{used to store the reducing polynomials $P(u)$ as they are computed}
{at position $i$, the minimal reducing polynomial $P(u)$ of degree $i$ is stored}
    subcoderec = record
                len: degtype;
                kfield: degtype
            end;
{each KM code can be subdivided into a certain number of subcodes}
{subcoderec stores the length and dimension of one subcode}
    subcodeptr = ^subcodelist;
    subcodelist = record
            sub: subcoderec;
            next: subcodeptr
        end;
{this linked list stores subcoderec - one for each subcode of the actual code}
{this is required to compute the burst error - detection capability}
var
    k,dim: kstep, dimstep, limit,N ,td, pd, nd, rd,remove: degtype;
    multihead: multiptr;
    A: boolean;
    RA: ReduceArray;
    len: matrixrange;
    Mx1: Matrix;
    F: text;
{*******************************************************************}
{The following 7 procedures produce all possible monic binary polys. of degree 1,..,6}
procedure GenPolys1 (var A1: Polys);
{monic binary polynomials of degree 1}
var
    l: degtype;
    i: binary;
begin
    l := 0;
    for i := 0 to 1 do
        begin
            A1[l].degP := 1;
            A1[l].P[0] := i;
            A1[l].P[1] := 1;
            l := l + 1
        end
end;
procedure GenPolys2 (var A2: Polys);
{monic binary polynomials of degree 2}
var
    l: degtype;
    i, j: binary;
begin
```

```
    l := 0;
    for i := 0 to 1 do
        for j := 0 to 1 do
            begin
                A2[l].degP := 2;
                A2[l].P[0] := i;
                A2[l].P[1] := j;
                A2[l].P[2] := 1;
                l := l + 1
            end
end;
procedure GenPolys3 (var A3: Polys);
{monic binary polynomials of degree 3}
var
    l: degtype;
    i, j, k: binary;
begin
    l := 0;
    for i := 0 to 1 do
        for j := 0 to 1 do
            for k := 0 to 1 do
                begin
                    A3[l].degP := 3;
                    A3[l].P[0] := i;
                    A3[l].P[1] := j;
                    A3[l].P[2] := k;
                    A3[l].P[3] := 1;
                    l := l + 1
                end
end;
procedure GenPolys4 (var A4: Polys);
{monic binary polynomials of degree 4}
var
    l: degtype;
    i, j, k, m: binary;
begin
    l := 0;
    for i := 0 to 1 do
        for j := 0 to 1 do
            for k := 0 to 1 do
                for m := 0 to 1 do
                    begin
                        A4[l].degP := 4;
                        A4[l].P[0] := i;
                        A4[l].P[1] := j;
                        A4[l].P[2] := k;
                    A4[l].P[3] := m;
                    A4[l].P[4] := 1;
                    l := l + 1
                end
end;
procedure GenPolys5 (var A5: Polys);
{monic binary polynomials of degree 5}
var
    l: degtype;
    i, j, k, m, n: binary;
begin
    l := 0;
    for i := 0 to 1 do
        for j := 0 to 1 do
            for k := 0 to 1 do
                for m := 0 to 1 do
                    for n := 0 to 1 do
                        begin
                            A5[l].degP := 5;
                            A5[l].P[0] := i;
                            A5[l].P[1] := j;
                            A5[l].P[2] := k;
                            A5[l].P[3] := m;
                            A5[l].P[4] := n;
                            A5[l].P[5] := 1;
                            l := l + 1
                        end
end;
procedure GenPolys6 (var A6: Polys);
{monic binary polynomials of degree 6}
var
    l: degtype;
    i, j, k, m, n, o: binary;
begin
    l := 0;
    for i := 0 to 1 do
        for j := 0 to 1 do
            for k := 0 to 1 do
                for m := 0 to 1 do
                    for n := 0 to 1 do
                        for o := 0 to 1 do
                            begin
                                A6[l].degP := 6;
                                A6[l].P[0] := i;
                                A6[l].P[1] := j;
                                A6[l].P[2] := k;
                        A6[l].P[3] := m;
                        A6[l].P[4] := n;
                        A6[l].P[5] := o;
                        A6[l].P[6] := 1;
                        l := l + 1
                    end
end;
procedure GenPolys (var A: Polys; q: degtype);
begin
    case q of
        1:
            GenPolys1(A);
        2:
            GenPolys2(A);
        3:
            GenPolys3(A);
        4:
            GenPolys4(A);
        5:
            GenPolys5(A);
        6:
            GenPolys6(A);
    end
end;
{************************************************************************}
procedure InitalisePoly (var E7: Polyrec; g7: degtype);
{generates the polynomial $E7(u)= e_0 + e_1u + .. + e_{g7-1}u^{g7-1}$ }
var
    i, j: degtype;
begin
    for i := 0 to g7 - 1 do
        begin
            for j := 0 to g7 - 1 do
                E7.poly[i, j] := 0;
            E7.poly[i, i] := 1
        end;
    E7.deg := g7 - 1
end;
procedure FindInverse (var E8: Polyrec; g8: degtype);
{computes the inverse of polynomial $E8(u)$,namely,$e_{g8-1} + e_{g8-1}u +. .+e_0u^{g8-1}$ }
{which is required for the wraparound calculation}
var
    i, j: degtype;
begin
    for i := 0 to g8 - 1 do
        begin
```

```
                for j := 0 to g8 - 1 do
                    E8.poly[i, j] := 0;
                E8.poly[i, g8 - 1 - i] := 1
            end;
        E8.deg := g8 - 1
end;
{***********************************************************************}
{The multiplication algorithms for  reduced polynomials of degree 0,..,5 now follow}
procedure multi0 (E0: Polyrec; g0: degtype; var ref0: multiptr);
{multi algorithm where reduced polynomials are of degree 0}
var
    i: degtype;
    q: multiptr;
begin
    new(q);
    ref0^.next := q;
    ref0 := q;
    for i := 0 to g0 - 1 do
        ref0^.m[i] := E0.poly[i, 0];
end;
procedure multi1 (E1: Polyrec; g1: degtype; var ref1: multiptr);
{multi algorithm where reduced polynomials are of degree 1}
var
    i: degtype;
    q: multiptr;
begin
    new(q);
    ref1^.next := q;
    ref1 := q;
    for i := 0 to g1 - 1 do
        ref1^.m[i] := E1.poly[i, 0];
    new(q);
    ref1^.next := q;
    ref1 := q;
    for i := 0 to g1 - 1 do
        ref1^.m[i] := E1.poly[i, 1];
    new(q);
    ref1^.next := q;
    ref1 := q;
    for i := 0 to g1 - 1 do
        ref1^.m[i] := (E1.poly[i, 0] + E1.poly[i, 1]) mod 2;
end;
procedure multi2A (E2: Polyrec; g2: degtype; var ref2: multiptr);
{multi algorithm where reduced polynomials are of degree 2 and reducing poly. P(u)=u^3}
var
    i: degtype;
    q: multiptr;
begin
    new(q);
    ref2^.next := q;
    ref2 := q;
    for i := 0 to g2 - 1 do
        ref2^.m[i] := E2.poly[i, 0];
    new(q);
    ref2^.next := q;
    ref2 := q;
    for i := 0 to g2 - 1 do
        ref2^.m[i] := E2.poly[i, 1];
    new(q);
    ref2^.next := q;
    ref2 := q;
    for i := 0 to g2 - 1 do
        ref2^.m[i] := E2.poly[i, 2];
    new(q);
    ref2^.next := q;
    ref2 := q;
    for i := 0 to g2 - 1 do
        ref2^.m[i] := (E2.poly[i, 0] + E2.poly[i, 1]) mod 2;
    new(q);
    ref2^.next := q;
    ref2 := q;
    for i := 0 to g2 - 1 do
        ref2^.m[i] := (E2.poly[i, 0] + E2.poly[i, 2]) mod 2;
end;
procedure multi2B (E2: Polyrec; g2: degtype; var ref2: multiptr);
{multi algorithm where reduced polynomials are of degree 2 and  reducing poly. P(u)≠u^3 }
var
    i: degtype;
    q: multiptr;
begin
    new(q);
    ref2^.next := q;
    ref2 := q;
    for i := 0 to g2 - 1 do
        ref2^.m[i] := E2.poly[i, 0];
    new(q);
    ref2^.next := q;
    ref2 := q;
    for i := 0 to g2 - 1 do
        ref2^.m[i] := E2.poly[i, 1];
    new(q);
    ref2^.next := q;
    ref2 := q;
    for i := 0 to g2 - 1 do
        ref2^.m[i] := E2.poly[i, 2];
    new(q);
    ref2^.next := q;
    ref2 := q;
    for i := 0 to g2 - 1 do
        ref2^.m[i] := (E2.poly[i, 0] + E2.poly[i, 1]) mod 2;
    new(q);
    ref2^.next := q;
    ref2 := q;
    for i := 0 to g2 - 1 do
        ref2^.m[i] := (E2.poly[i, 1] + E2.poly[i, 2]) mod 2;
    new(q);
    ref2^.next := q;
    ref2 := q;
    for i := 0 to g2 - 1 do
        ref2^.m[i] := (E2.poly[i, 0] + E2.poly[i, 2]) mod 2;
end;
procedure multi2 (E2: Polyrec; g2: degtype; var ref2: multiptr);
{decides which algorithm should be used for reduced polynomials of degree 2}
begin
    if A then
        multi2A(E2, g2, ref2)
    else
        multi2B(E2, g2, ref2);
end;
procedure multi3A (E3: Polyrec; g3: degtype; var ref3: multiptr);
{multi algorithm where reduced polynomials are of degree 3 and reducing poly. P(u)=u^4 }
var
    i: degtype;
    q: multiptr;
begin
    new(q);
    ref3^.next := q;
    ref3 := q;
    for i := 0 to g3 - 1 do
        ref3^.m[i] := E3.poly[i, 0];
    new(q);
    ref3^.next := q;
    ref3 := q;
    for i := 0 to g3 - 1 do
        ref3^.m[i] := E3.poly[i, 1];
    new(q);
    ref3^.next := q;
    ref3 := q;
```

```
        for i := 0 to g3 - 1 do
            ref3^.m[i] := E3.poly[i, 2];
        new(q);
        ref3^.next := q;
        ref3 := q;
        for i := 0 to g3 - 1 do
            ref3^.m[i] := E3.poly[i, 3];
        new(q);
        ref3^.next := q;
        ref3 := q;
        for i := 0 to g3 - 1 do
            ref3^.m[i] := (E3.poly[i, 0] + E3.poly[i, 1]) mod 2;
        new(q);
        ref3^.next := q;
        ref3 := q;
        for i := 0 to g3 - 1 do
            ref3^.m[i] := (E3.poly[i, 0] + E3.poly[i, 2]) mod 2;
        new(q);
        ref3^.next := q;
        ref3 := q;
        for i := 0 to g3 - 1 do
            ref3^.m[i] := (E3.poly[i, 1] + E3.poly[i, 2]) mod 2;
        new(q);
        ref3^.next := q;
        ref3 := q;
        for i := 0 to g3 - 1 do
            ref3^.m[i] := (E3.poly[i, 0] + E3.poly[i, 3]) mod 2;
    end;
    procedure multi3B (E3: Polyrec; g3: degtype; var ref3: multiptr);
    {multi algorithm where reduced polynomials are of degree 3 and reducing poly. P(u)≠u^4}
    var
        i: degtype;
        q: multiptr;
    begin
        new(q);
        ref3^.next := q;
        ref3 := q;
        for i := 0 to g3 - 1 do
            ref3^.m[i] := E3.poly[i, 0];
        new(q);
        ref3^.next := q;
        ref3 := q;
        for i := 0 to g3 - 1 do
            ref3^.m[i] := E3.poly[i, 1];
        new(q);
        ref3^.next := q;
    ref3 := q;
    for i := 0 to g3 - 1 do
        ref3^.m[i] := (E3.poly[i, 0] + E3.poly[i, 1]) mod 2;
    new(q);
    ref3^.next := q;
    ref3 := q;
    for i := 0 to g3 - 1 do
        ref3^.m[i] := E3.poly[i, 2];
    new(q);
    ref3^.next := q;
    ref3 := q;
    for i := 0 to g3 - 1 do
        ref3^.m[i] := E3.poly[i, 3];
    new(q);
    ref3^.next := q;
    ref3 := q;
    for i := 0 to g3 - 1 do
        ref3^.m[i] := (E3.poly[i, 2] + E3.poly[i, 3]) mod 2;
    new(q);
    ref3^.next := q;
    ref3 := q;
    for i := 0 to g3 - 1 do
        ref3^.m[i] := (E3.poly[i, 0] + E3.poly[i, 2]) mod 2;
    new(q);
    ref3^.next := q;
    ref3 := q;
    for i := 0 to g3 - 1 do
        ref3^.m[i] := (E3.poly[i, 1] + E3.poly[i, 3]) mod 2;
    new(q);
    ref3^.next := q;
    ref3 := q;
    for i := 0 to g3 - 1 do
        ref3^.m[i] := (E3.poly[i, 0] + E3.poly[i, 1] + E3.poly[i, 2] + E3.poly[i, 3]) mod 2;
end;
procedure multi3 (E3: Polyrec; g3: degtype; var ref3: multiptr);
{decides which algorithm should be used for reduced polynomials of degree 3}
begin
    if A then
        multi3A(E3, g3, ref3)
    else
        multi3B(E3, g3, ref3);
end;
procedure multi4A (E4: Polyrec; g4: degtype; var ref4: multiptr);
{multi algorithm where reduced polynomials are of degree 4 and reducing poly. P(u)=u^5}
var
    i: degtype;
    q: multiptr;
begin
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := E4.poly[i, 0];
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := E4.poly[i, 1];
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := (E4.poly[i, 0] + E4.poly[i, 1]) mod 2;
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := E4.poly[i, 2];
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := (E4.poly[i, 0] + E4.poly[i, 2]) mod 2;
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := E4.poly[i, 3];
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := (E4.poly[i, 0] + E4.poly[i, 3]) mod 2;
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := (E4.poly[i, 1] + E4.poly[i, 2]) mod 2;
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
```

```
        ref4^.m[i] := E4.poly[i, 4];
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := (E4.poly[i, 0] + E4.poly[i, 4]) mod 2;
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := (E4.poly[i, 1] + E4.poly[i, 3]) mod 2;
end;
procedure multi4B (E4: Polyrec; g4: degtype; var ref4: multiptr);
{multi algorithm where reduced polynomials are of degree 4 and reducing poly. P(u)≠u^5 }
var
    i: degtype;
    q: multiptr;
begin
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := E4.poly[i, 0];
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := E4.poly[i, 1];
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := (E4.poly[i, 0] + E4.poly[i, 1]) mod 2;
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := E4.poly[i, 2];
    new(q);
    ref4^.next := q;
    ref4 := q;
    for i := 0 to g4 - 1 do
        ref4^.m[i] := (E4.poly[i, 0] + E4.poly[i, 2]) mod 2;
    new(q);
    ref4^.next := q;
    ref4 := q;
```

```
        for i := 0 to g4 - 1 do
            ref4^.m[i] := E4.poly[i, 3];
        new(q);
        ref4^.next := q;
        ref4 := q;
        for i := 0 to g4 - 1 do
            ref4^.m[i] := (E4.poly[i, 1] + E4.poly[i, 3]) mod 2;
        new(q);
        ref4^.next := q;
        ref4 := q;
        for i := 0 to g4 - 1 do
            ref4^.m[i] := (E4.poly[i, 2] + E4.poly[i, 3]) mod 2;
        new(q);
        ref4^.next := q;
        ref4 := q;
        for i := 0 to g4 - 1 do
            ref4^.m[i] := (E4.poly[i, 0] + E4.poly[i, 1] + E4.poly[i, 2] + E4.poly[i, 3]) mod 2;
        new(q);
        ref4^.next := q;
        ref4 := q;
        for i := 0 to g4 - 1 do
            ref4^.m[i] := E4.poly[i, 4];
        new(q);
        ref4^.next := q;
        ref4 := q;
        for i := 0 to g4 - 1 do
            ref4^.m[i] := (E4.poly[i, 0] + E4.poly[i, 4]) mod 2;
        new(q);
        ref4^.next := q;
        ref4 := q;
        for i := 0 to g4 - 1 do
            ref4^.m[i] := (E4.poly[i, 1] + E4.poly[i, 4]) mod 2;
        new(q);
        ref4^.next := q;
        ref4 := q;
        for i := 0 to g4 - 1 do
            ref4^.m[i] := (E4.poly[i, 2] + E4.poly[i, 4]) mod 2;
        new(q);
        ref4^.next := q;
        ref4 := q;
        for i := 0 to g4 - 1 do
            ref4^.m[i] := (E4.poly[i, 3] + E4.poly[i, 4]) mod 2;
    end;
    procedure multi4 (E4: Polyrec; g4: degtype; var ref4: multiptr);
    {decides which algorithm should be used for reduced polynomials of degree 4}
    begin
    if A then
        multi4A(E4, g4, ref4)
    else
        multi4B(E4, g4, ref4);
end;
procedure multi5A (E5: Polyrec; g5: degtype; var ref5: multiptr);
{multi algorithm where reduced polys are of degree 5 and the reducing poly. P(u)=u^6 }
var
    i: degtype;
    q: multiptr;
begin
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := E5.poly[i, 0];
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := E5.poly[i, 1];
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := (E5.poly[i, 0] + E5.poly[i, 1]) mod 2;
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := E5.poly[i, 2];
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := (E5.poly[i, 0] + E5.poly[i, 2]) mod 2;
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := E5.poly[i, 3];
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := E5.poly[i, 4];
        new(q);
        ref5^.next := q;
        ref5 := q;
        for i := 0 to g5 - 1 do
            ref5^.m[i] := (E5.poly[i, 2] + E5.poly[i, 3]) mod 2;
        new(q);
        ref5^.next := q;
        ref5 := q;
        for i := 0 to g5 - 1 do
            ref5^.m[i] := (E5.poly[i, 1] + E5.poly[i, 3]) mod 2;
        new(q);
        ref5^.next := q;
        ref5 := q;
        for i := 0 to g5 - 1 do
            ref5^.m[i] := (E5.poly[i, 0] + E5.poly[i, 1] + E5.poly[i, 2] + E5.poly[i, 3]) mod 2;
        new(q);
        ref5^.next := q;
        ref5 := q;
        for i := 0 to g5 - 1 do
            ref5^.m[i] := (E5.poly[i, 0] + E5.poly[i, 4]) mod 2;
        new(q);
        ref5^.next := q;
        ref5 := q;
        for i := 0 to g5 - 1 do
            ref5^.m[i] := E5.poly[i, 5];
        new(q);
        ref5^.next := q;
        ref5 := q;
        for i := 0 to g5 - 1 do
            ref5^.m[i] := (E5.poly[i, 0] + E5.poly[i, 5]) mod 2;
        new(q);
        ref5^.next := q;
        ref5 := q;
        for i := 0 to g5 - 1 do
            ref5^.m[i] := (E5.poly[i, 1] + E5.poly[i, 4]) mod 2;
    end;
    procedure multi5B (E5: Polyrec; g5: degtype; var ref5: multiptr);
    {multi algorithm where reduced polys are of degree 5 and the reducing poly P(u)≠u^6 }
    var
        i: degtype;
        q: multiptr;
    begin
        new(q);
        ref5^.next := q;
        ref5 := q;
        for i := 0 to g5 - 1 do
    ref5^.m[i] := E5.poly[i, 0];
new(q);
ref5^.next := q;
ref5 := q;
for i := 0 to g5 - 1 do
    ref5^.m[i] := E5.poly[i, 1];
new(q);
ref5^.next := q;
ref5 := q;
for i := 0 to g5 - 1 do
    ref5^.m[i] := E5.poly[i, 2];
new(q);
ref5^.next := q;
ref5 := q;
for i := 0 to g5 - 1 do
    ref5^.m[i] := (E5.poly[i, 0] + E5.poly[i, 1]) mod 2;
new(q);
ref5^.next := q;
ref5 := q;
for i := 0 to g5 - 1 do
    ref5^.m[i] := (E5.poly[i, 1] + E5.poly[i, 2]) mod 2;
new(q);
ref5^.next := q;
ref5 := q;
for i := 0 to g5 - 1 do
    ref5^.m[i] := (E5.poly[i, 0] + E5.poly[i, 2]) mod 2;
new(q);
ref5^.next := q;
ref5 := q;
for i := 0 to g5 - 1 do
    ref5^.m[i] := E5.poly[i, 3];
new(q);
ref5^.next := q;
ref5 := q;
for i := 0 to g5 - 1 do
    ref5^.m[i] := E5.poly[i, 4];
new(q);
ref5^.next := q;
ref5 := q;
for i := 0 to g5 - 1 do
    ref5^.m[i] := E5.poly[i, 5];
new(q);
ref5^.next := q;
ref5 := q;
for i := 0 to g5 - 1 do
    ref5^.m[i] := (E5.poly[i, 3] + E5.poly[i, 4]) mod 2;
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := (E5.poly[i, 4] + E5.poly[i, 5]) mod 2;
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := (E5.poly[i, 3] + E5.poly[i, 5]) mod 2;
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := (E5.poly[i, 0] + E5.poly[i, 3]) mod 2;
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := (E5.poly[i, 1] + E5.poly[i, 4]) mod 2;
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := (E5.poly[i, 2] + E5.poly[i, 5]) mod 2;
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := (E5.poly[i, 0] + E5.poly[i, 1] + E5.poly[i, 3] + E5.poly[i, 4]) mod 2;
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := (E5.poly[i, 1] + E5.poly[i, 2] + E5.poly[i, 4] + E5.poly[i, 5]) mod 2;
    new(q);
    ref5^.next := q;
    ref5 := q;
    for i := 0 to g5 - 1 do
        ref5^.m[i] := (E5.poly[i, 0] + E5.poly[i, 2] + E5.poly[i, 3] + E5.poly[i, 5]) mod 2;
end;
procedure multi5 (E5: Polyrec; g5: degtype; var ref5: multiptr);
{decides which algorithm should be used for reduced polys of degree 5}
begin
    if A then
        multi5A(E5, g5, ref5)
    else
        multi5B(E5, g5, ref5);
end;
procedure Mult (E: Polyrec; g: degtype; var ref: multiptr);
{decides which multi algorithm should be used depending on the degree of  reduced polys}
begin
    case E.deg of
        0:
            multi0(E, g, ref);
        1:
            multi1(E, g, ref);
        2:
            multi2(E, g, ref);
        3:
            multi3(E, g, ref);
        4:
            multi4(E, g, ref);
        5:
            multi5(E, g, ref);
    end
end;
{***************************************************************************}
{This procedure enables us to get the generator matrix}
procedure ProduceMatrix (k1: degtype);
{The multi coefficients are used to obtain the generator matrix}
var
    i: 0..matrixmax;
    j: degtype;
    L1: integer;
begin
    multihead := multihead^.next;
    i := 0;
    while multihead <> nil do
        begin
            for j := 0 to k1 - 1 do
                Mx1[j, i] := multihead^.m[j];
            multihead := multihead^.next;
            i := i + 1;
        end;
    L1 := i;
    for j := 0 to k1 - 1 do
        begin
            for i := 0 to L1 - 1 do
                Write(F, Mx1[j, i] : 1);
            WriteLn(F);
        end;
```

```
end;
{**********************************************************************}
{The next three procedures determine whether or not two polynomials are coprime}
{-these procedures are necessary when finding the minimal reducing polynomial P(u)}
function polymod (x1, y1: Prec): Prec;
{computes y1 mod x1 where  are polynomials such that y1>x1}
var
    i, j: degtype;
    temp: Prec;
begin
    temp := y1;
    with temp do
        begin
            if (x1.degP = 0) and (x1.P[0] = 1) then
                begin
                    degP := 0;
                    P[0] := 0
                end
            else
                begin
                    while x1.degP ≤ degP do
                        begin
                            for i := 0 to x1.degP - 1 do
                                begin
                                    if x1.P[i] = 1 then
                                        P[degP - x1.degP + i] := (P[degP - x1.degP + i] +
                                                                        P[degP]) mod 2
                                end;
                            P[degP] := 0;
                            repeat
                                degP := degP - 1
                            until (degP = 0) or (P[degP] <> 0);
                        end
                end;
        end;
    polymod := temp
end;
function gcd (x2, y2: Prec): Prec;
{computes the gcd of polynomials x2 and y2 assuming y2>x2}
begin
    if (x2.degP = 0) and (x2.P[0] = 0) then
        gcd := y2
    else
        gcd := gcd(polymod(x2, y2), x2)
end;
function CoPrime (m, n: Prec): boolean;
```

```
{determines whether or not polynomials m and n are coprime where n>m}
var
    temp: Prec;
begin
    temp := gcd(m, n);
    if (temp.degP = 0) and (temp.P[0] = 1) then
        CoPrime := true
    else
        CoPrime := false
end;
{******************************************************************}
{The next section of procedures find the minimal reducing polynomial P(u)}
function Power (a, b: degtype): integer;{computes a^b}
begin
    if b = 0 then
        Power := 1
    else
        Power := (a * Power(a, (b - 1)))
end;
function CheckPoly (W: Prec; headW: PolyPtr): boolean;
{determines whether polynomial W is coprime to each of the polynomials in the list}
var
    OK: boolean;
    temp: PolyPtr;
begin
    OK := true;
    if (headW^.PolyP.P[0] = 0) and (headW^.PolyP.degP = 0) then
        OK := true
    else
        begin
            temp := headW;
            while (temp^.link <> nil) and (OK) do
                begin
                    if not CoPrime(temp^.PolyP, W) then
                        OK := false
                    else
                        temp := temp^.link;
                end
        end;
    CheckPoly := OK
end;
procedure ProcessPolysDeg (D1: integer; var head1, tail1: PolyPtr);
var
    count, i: integer;
    B: polys;
    temp: PolyPtr;
```

```
begin
    count := 0;
    GenPolys(B, pd);
    if pd = rd then
        begin
            while (count < Power(2, pd) - remove) and (pd + td ≤ D1) do
                begin
                    if CheckPoly(B[count + remove], head1) then
                        begin
                            for i := 0 to B[count + remove].degP do
                                tail1^.PolyP.P[i] := B[count + remove].P[i];
                            tail1^.PolyP.degP := B[count + remove].degP;
                            td := td + pd;
                            new(temp);
                            tail1^.link := temp;
                            tail1 := temp;
                            tail1^.link := nil
                        end;
                    count := count + 1;
                end
        end
    else
        begin
            while (count < Power(2, pd)) and (pd + td ≤ D1) do
                begin
                    if CheckPoly(B[count], head1) then
                        begin
                            for i := 0 to B[count].degP do
                                tail1^.PolyP.P[i] := B[count].P[i];
                            tail1^.PolyP.degP := B[count].degP;
                            td := td + pd;
                            new(temp);
                            tail1^.link := temp;
                            tail1 := temp;
                            tail1^.link := nil
                        end;
                    count := count + 1;
                end
        end
end;
procedure Update (D2: integer; var head2, tail2: PolyPtr);
var
    temp: PolyPtr;
begin
    if td <> D2 then
        begin
```

```
            if (td > D2) or (td + nd > D2) then
                begin
                    pd := rd;
                    remove := remove + 1;
                    head2 := head2^.link;
                    head2^.link := nil;
                    tail2 := head2;
                    new(temp);
                    tail2^.link := temp;
                    tail2 := temp;
                    tail2^.link := nil;
                    td := head2^.PolyP.degP;
                    nd := pd + 1;
                end
            else if td + nd ≤ D2 then
                begin
                    pd := pd + 1;
                    nd := pd + 1
                end;
            if remove = power(2, rd) - 1 then
                begin
                    pd := rd + 1;
                    rd := rd + 1;
                    td := head2^.PolyP.degP;
                    remove := 0
                end
        end
end;
procedure ComputePolyP (D3: integer; var first: PolyPtr);
{computes the minimal reducing polynomial P(u) of degree D3}
var
    i: integer;
    head, tail: PolyPtr;
begin
    new(head);
    head^.PolyP.P[0] := 0;
    head^.PolyP.degP := 0;
    new(tail);
    tail := head;
    pd := 1;
    rd := 1;
    td := 0;
    nd := 2;
    remove := 0;
    while td <> D3 do
        begin
```

```
            ProcessPolysDeg(D3, head, tail);
            Update(D3, head, tail);
        end;
    first := head;
end;
procedure InitialiseArray;
{initialises the reduce array which will store the minimal polys. when they are computed}
var
    i: integer;
begin
    for i := 1 to 50 do
        RA[i].RAPoly := nil
end;
procedure FindPolyP (D0: integer; var head0: POlyPtr);
var
    temp: polyptr;
    maxd: degtype;
begin
    if RA[D0].RAPoly = nil then
{if the minimal polynomial has not been determined, then it is found & stored in RA[D0]}
        begin
            ComputePolyP(D0, head0);
            RA[D0].RAPoly := head0;
            temp := head0;
            maxd := 0;
            if temp^.link = nil then
                begin
                    RA[D0].maxdeg := temp^.PolyP.degP
                end
            else
                begin
                    while temp^.link <> nil do
                        begin
                            if temp^.PolyP.degP > maxd then
                                maxd := temp^.PolyP.degP;
                                temp := temp^.link
                        end;
                    RA[D0].maxdeg := maxd;
                end
        end
    else
{ if the minimal reducing polynomial was found previously-simply get it from RA[D0]}
        head0 := RA[D0].RAPoly;
end;
procedure WritePolyList (first: PolyPtr);
```

```
{ $P(u) = \prod_{i=1}^{t} P_i(u)$ , then $P_1(u),..,P_t(u)$ are written in binary form}
var
    temp: PolyPtr;
    i: integer;
begin
    temp := first;
    if temp^.link = nil then
        begin
            with temp^ do
                begin
                    for i := 0 to PolyP.degP do
                        Write(F, PolyP.P[i] : 2);
                    WriteLn(F)
                end
        end
    else
        begin
        while temp^.link <> nil do
                begin
                    with temp^ do
                        begin
                            for i := 0 to PolyP.degP do
                                Write(F, PolyP.P[i] : 2);
                            WriteLn(F)
                        end;
                    temp := temp^.link
                end
        end
end;
procedure WraparoundPoly (var Ply1: Prec; s1: integer);
{ expresses $u^s$ in binary form}
var
    i: degtype;
begin
    with Ply1 do
        begin
            degP := s1;
            for i := 0 to s1 - 1 do
                P[i] := 0;
            P[s1] := 1
        end
end;
{************************************************************************}
```

```
procedure ReducePoly (var E2: Polyrec; g2: degtype; Ply2: Prec);
{computes E2(u) modulo Ply2}
var
    i, j: degtype;
begin
    with Ply2 do
        begin
            while degP <= E2.deg do
                begin
                    for i := 0 to degP - 1 do
                        begin
                            if P[i] = 1 then
                                begin
                                    for j := 0 to g2 - 1 do
                                        E2.poly[j, E2.deg - degP + i] :=
                                        (E2.poly[j, E2.deg - degP + i] + E2.poly[j, E2.deg])
                                                                                mod 2;
                                end
                        end;
                    for i := 0 to g2 - 1 do
                        E2.poly[i, E2.deg] := 0;
                    E2.deg := E2.deg - 1;
                end
        end;
end;
{*******************************************************************}
{the following procedures are used to find the actual minimum distance of the code}
function AddCodeWord (word1, word2: subcodearray; L2: degtype): subcodearray;
var
    i: degtype;
    temp: subcodearray;
begin
{each corresponding bit of the 2 codewords is added to obtain}
{  the corresponding bit of the new codeword}
    for i := 0 to L2 - 1 do
        temp[i] := (word1[i] + word2[i]) mod 2;
    AddCodeWord := temp;
end;
procedure Dist (word1: subcodearray; L1: degtype; var mindist1: degtype);
var
    i, count: integer;
begin
    count := 0;{count records the no of 1's found in word1 so far}
    for i := 0 to L1 - 1 do
        count := count + word1[i];
{now, count=distance of word1}
```

```
    if (0 < count) and (count < mindist1) then
{if distance of word1<mindist then mindist is updated}
        mindist1 := count;
end;
procedure FindCodeWord (word3: subcodeArray; i, K3, L3: degtype; var mindist3:
degtype);
{ this is a recursive procedure which ensures that all the codewords are found}
var
    temp: subcodearray;
    j: degtype;
begin
    while i < K3 do
        begin
            temp := AddCodeWord(word3, Mx1[i], L3);
            Dist(temp, L3, mindist3);
            j := i + 1;
            FindCodeWord(temp, j, K3, L3, mindist3);
            i := i + 1;
        end;
end;
procedure FindMinDist (L7: matrixrange; k7: degtype);
{finds the actual minimum distance of the code}
var
    row, ind, mindist: degtype;
    codeword: subcodearray;
begin
    row := 0;
    mindist := L7;
    while row < k7 do
        begin
            codeword := Mx1[row];
            Dist(codeword, L7, mindist);
            ind := row + 1;
            FindCodeWord(codeword, ind, k7, L7, mindist);
            row := row + 1
        end;
    WriteLn(F);
    WriteLn(F, 'The actual minimum distance is ', mindist : 3);
end;
{*****************************************************************}
procedure GetDims (var k3, kstep3, deg3, degstep3, limit3, N3: degtype);
{prompts the user to enter initial data}
begin
    WriteLn('Please enter a value for k');
    Read(k3);
    WriteLn('and a value for d:');
    Read(deg3);
    N3 := k3 + deg3 - 1;
    WriteLn('Now, enter kstep and dstep respectively');
    Read(kstep3, degstep3);
    WriteLn('and finally the limit');
    Read(limit3);
    WriteLn(F, 'k = ', k3 : 3, ' d = ', deg3 : 3, ' kstep = ', kstep3 : 3);
    WriteLn( ' dstep = ', degstep3 : 3, ' limit = ', limit3 : 3);
    WriteLn(F, '______________________________________________________');
    WriteLn(F, ' k= ',k3 : 3, ' d = ', deg3 : 3);
end;
procedure UpdateDims (var k4, kstep4, deg4, degstep4, N4: degtype);
{after data for (k ,d ) KM code has been produced,k:=k+kstep,d:=d+dstep}
{ and procedures repeated}
begin
    k4 := k4 + kstep4;
    deg4 := deg4 + degstep4;
    WriteLn(F, '______________________________________________________');
    WriteLn(F, 'k and d are increased to ', k4 : 3, ' and ', deg4 : 3, ' resp.');
    N4 := k4 + deg4 - 1;
end;
function min (a, b: integer): integer;
{finds the minimum of integers a and b}
begin
    if a < b then
        min := a
    else
        min := b
end;
procedure SetD (N4: degtype; var D4: degtype; var s4: wraprange);
{for each k and d, the degree of P(u) is initially set to k+d-1 and the number of wraps to 0}
begin
    WriteLn(F, 'We must have N = D + s = ', N4 : 3);
    s4 := 0;
    D4 := N4;
end;
function UseMultiA (X: Prec): boolean;
{if P(u)=u^c - then a different multi algorithm must be used}
var
    count: integer;
begin
    count := 0;
    while X.P[count] = 0 do
        count := count + 1;
    if count = X.degP then
        UseMultiA := true
    else
        UseMultiA := false;
end;
function MultiWrap (s7: integer): integer;
{gives the number of multis required for different wraparounds}
begin
    case s7 of
        0:
            MultiWrap := 0;
        1:
            MultiWrap := 1;
        2:
            MultiWrap := 3;
        3:
            MultiWrap := 5;
        4:
            MultiWrap := 8;
        5:
            MultiWrap := 11;
        6:
            MultiWrap := 14
    end
end;
function MultiComp (M: Prec): integer;
{gives the number of multis required for reducing polynomials }
begin
    if UseMultiA(M) then
        MultiComp := MultiWrap(M.degP)
    else
        begin
            case M.degP of
                0:
                    MultiComp := 0;
                1:
                    MultiComp := 1;
                2:
                    MultiComp := 3;
                3:
                    MultiComp := 6;
                4:
                    MultiComp := 9;
                5:
                    MultiComp := 14;
                6:
                    MultiComp := 18
            end
        end
end;
procedure FindCodeLength (var L5: matrixrange; D5: degtype; s5: wraprange);
{computes the length of code simply by adding the number of multis corresponding}
{ to each reducing polynomial}
var
    temp: PolyPtr;
    j: degtype;
begin
    L5 := 0;
    FindPolyP(D5, temp);
    if temp^.link = nil then
        L5 := MultiComp(temp^.PolyP)
    else
        begin
            while temp^.link <> nil do
                begin
                    L5 := L5 + MultiComp(temp^.PolyP);
                    temp := temp^.link
                end
        end;
    L5 := L5 + MultiWrap(s5);
end;
procedure DisplayResult (var L6: matrixrange; D6, k6: degtype; s6: wraprange;
                                                                head: PolyPtr);
begin
    FindCodeLength(L6, D6, s6);
    WriteLn(F, '****************************************');
    WriteLn(F, 'D =',D6 : 3,'s =', s6 : 3, 'length of code =', L6 : 4, 'with gen. matrix:');
    ProduceMatrix(k6);
    WriteLn(F);
    WriteLn(F, 'The reducing polys used (in binary form) are:');
    WritePolyList(head);
    FindMinDist(L6, k6);
end;
function CheckValid (D9, k9, deg9: degtype): boolean;
{program only has algorithms for multiplying reduced polynomials of the same degree}
{- if they are not of the same degree then code is not produced}
var
    valid: boolean;
begin
    valid := true;
    if RA[D9].maxdeg > min(k9, deg9) then
        valid := false;
    CheckValid := valid
end;
```

```
procedure ProcessPoly (D7, k7: degtype; s7: wraprange);
{will produce some the data for specified D,s and k}
var
    Pref, PP: PolyPtr;
    multiref: multiptr;
    Pu: Prec;
    E: Polyrec;
begin
    new(multihead);
    new(multiref:);
    multihead := nil;
    multiref: := multihead;
    FindPolyP(D7, PP);
    new(Pref);
    Pref := PP;
    if Pref^.link = nil then
        begin
            InitalisePoly(E, k7);
            ReducePoly(E, k7, Pref^.PolyP);
            if UseMultiA(Pref^.PolyP) then
                A := true
            else
                A := false;
            Mult(E, k7, multiref:);
            Pref := Pref^.link
        end
    else
        begin
            while Pref^.link <> nil do
                begin
                    InitalisePoly(E, k7);
                    ReducePoly(E, k7, Pref^.PolyP);
                    if UseMultiA(Pref^.PolyP) then
                        A := true
                    else
                        A := false;
                    Mult(E, k7, multiref:);
                    Pref := Pref^.link
                end;
        end;
    if s7 <> 0 then
        begin
            A := true;
            FindInverse(E, k7);
            WraparoundPoly(Pu, s7);
            ReducePoly(E, k7, Pu);
            Mult(E, k7, multiref:);
        end
    else
        A := false;
    multiref:^.next := nil;
    DisplayResult(len, D7, k7, s7, PP);
end;
procedure StateInvalid (D8: degtype; s8: wraprange);
{states that data cannot be produced since reduced polys. would be of different degrees}
begin
    WriteLn(F, 'The program was not able to compute the KMcode');
    WriteLn(F, 'for D= ', D8 : 3, 'and s= ', s8 : 3);
    WriteLn(F, 'An algorithm for multiplying polys of different degrees is needed');
    WriteLn(F, '************************');
end;
{*********************************************************************}
procedure GetCodeDims (var SL: subcodePtr; DP: degtype; sP: wraprange);
{the dims of each subcode are required to compute the burst error-detection capability}
var
    tempPP: PolyPtr;
    leader, q: subcodeptr;
begin
    new(SL);
    new(leader);
    leader := SL;
    FindPolyP(DP, tempPP);
    if tempPP^.link = nil then
        begin
            leader^.sub.len := MultiComp(tempPP^.PolyP);
            leader^.sub.kfield := tempPP^.PolyP.degP
        end
    else
        begin
            while tempPP^.link <> nil do
                begin
                    leader^.sub.len:=MultiComp(tempPP^.PolyP);
                    leader^.sub.kfield := tempPP^.PolyP.degP;
                    new(q);
                    leader^.next := q;
                    leader := q;
                    tempPP := tempPP^.link
                end;
            if sP <> 0 then
                begin
                    leader^.sub.len := MultiWrap(sP);
                    leader^.sub.kfield := sP;
                    new(q);
                    leader^.next := q;
                    leader := q
                end;
            leader^.next := nil
        end;
end;
procedure CheckThisCycle (SL1: subcodeptr; var GotBound: boolean; CA:
                                                    subcodeArray; dim1: degtype);
var
    i, NonZero, start, count: integer;
    nonCodeword: Boolean;
    SL: subcodePtr;
begin
    nonCodeword := false;
    NonZero := 0;
{nonZero records the number of nonzero subblocks}
    start := 1;
    SL := SL1;
    if SL^.next = nil then
        begin
            count := 0;
{count records the number of nonzero bits in the current subcode}
            for i := start to start + SL^.sub.len - 1 do
                count := count + CA[i];
            if count <> 0 then
                NonZero := NonZero + SL^.sub.kfield;
            if (0 < count) and (count < SL^.sub.kfield) then
{this subblock is nonzero yet it has less then required no of 1's- it will be detected}
                NonCodeWord := true;
            start := start + SL^.sub.len;
            SL := SL^.next
        end
    else
        begin
            while SL^.next <> nil do
                begin
                    count := 0;
                    for i := start to start + SL^.sub.len - 1 do
                        count := count + CA[i];
                    if count <> 0 then
                        NonZero := NonZero + SL^.sub.kfield;
                    if (0 < count) and (count < SL^.sub.kfield) then
                        NonCodeWord := true;
                    start := start + SL^.sub.len;
                    SL := SL^.next
                end;
            end;
        if (NonZero >= dim1) and (not (NonCodeword)) then
    {number of nonzero subblocks is greater than dim and each block has sufficient 1's}
    {- error will not be detected}
            gotbound := true;
    end;
    procedure CheckAllCycles (SL2: subcodeptr; var gotbound2: boolean; error2, len2,
                                            dim2: degtype; var CA2: subCodeArray);
    var
        ind, j: integer;
    begin
        for ind := 1 to len2 do
            CA2[ind] := 0;
        j := 1;
        while (j <= len2 - error2 + 1) and (not (gotbound2)) do
            begin
                for ind := j to j + error2 - 1 do
                    CA2[ind] := 1;
                CheckThisCycle(SL2, gotbound2, CA2, dim2);
                CA2[j] := 0;
                j := j + 1;
            end;
    end;
    procedure FindErrorBurst (D, len3, dim3: degtype; s: wraprange);
    {determines the burst error-detection capability of the code}
    var
        error, length: degtype;
        SC: subcodeptr;
        CA: subcodeArray;
        gotbound: boolean;
    begin
        gotbound := false;
        GetCodeDims(SC, D, s);
        error := 1;
        while (error <= len3) and (not (gotbound)) do
            begin
                CheckAllCycles(SC, gotbound, error, len3, dim3, CA);
                error := error + 1;
            end;
        WriteLn(F);
        WriteLn(F, 'The burst error detection capability is', error - 2);
    end;
    {*****************************************************************************}
```

```
procedure ProcessData (kk, kkstep, dd, ddstep, lim, NN: degtype);
{gets initial input, produces the required data, updates the variables until limit is reached}
var
    D: degtype;
    s: wraprange;
    temp: PolyPtr;
begin
    while NN < lim do
        begin
            SetD(NN, D, s);
            repeat
                FindPolyP(D, temp);
                if CheckValid(D, kk, dd) then
                    begin
                        ProcessPoly(D, kk, s);
                        FindErrorBurst(D, len, dd, s)
                    end
                else
                    StateInvalid(D, s);
                D := D - 1;
                s := s + 1;
            until s > min(6, min(kk, dd));
            UpdateDims(kk, kkstep, dd, ddstep, NN);
        end;
    WriteLn(F, 'N greater than limit');
end;

{*******************************************************************}
{Main Program}
begin
    ShowText;
    ReWrite(F, 'Results');
    InitialiseArray;
    GetDims(k, kstep, dim, dimstep, limit, N);
    ProcessData(k, kstep, dim, dimstep, limit, N);
    WriteLn('The program has finished running. The results can be found in a file')
    WriteLn('called Results - use 'miniWRITER' to open it ');
    Close(F);
end.
```

STATEMENT OF ORIGINALITY

This thesis is the author's own work, except where it is explicitly stated otherwise. In particular, the following are original :

CHAPTER 3

Example 3.1.1
Example 3.1.2
Example 3.1.3

CHAPTER 5

Example 5.1.1
Example 5.1.2
Example 5.2.1
Example 5.2.2
Example 5.3

CHAPTER 6

Example 6.2.1
Example 6.2.2
Example 6.3.1
Example 6.4.1
Example 6.6.1
Example 6.6.2
Example 6.7.1
Example 6.8.1
Example 6.8.2

CHAPTER 7

7.1 - Generalising how KM codes may be employed in GH-ARQ schemes
Example 7.1.1
Example 7.2.1
Example 7.4.1

APPENDIX A - Showing (3.6) $\approx$ (3.7) for any k and d.

APPENDIX B - Derivation of non-commutative algorithms for multiplying polynomials of degree 4 or 5 (i.e Multi 4A, Multi 4B, Multi 5A, Multi 5B).

APPENDIX D - KM Codes Program.

GLASGOW UNIVERSITY LIBRARY