Multi-h Synchronisation for Codes with Long Constraint-Length

Prepared by:

Eugéne Nel,

BSc(Eng) *Cape Town*

Prepared for:

The Department of Electrical and Electronic Engineering at the University of Cape Town in partial fulfillment of the requirements for the degree of MSc(Eng)

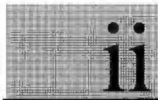
© University of Cape Town 1997

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Page



Declaration of Authenticity

I hereby declare that the work contained in this document is my own original work and that every attempt has been made to indicate and reference any exceptions or external sources.

I also certify that, prior to the submission of this dissertation, the work presented here has never been published in any form, nor has it been submitted to any educational institution for the purposes of obtaining any degree or other qualification.

Signature of Author..

Eugéne J. Nel
Department of Electrical Engineering
June 16, 1997

Page

iii Acknowledgments

A fair number of people were involved in helping me bring this project to completion. Foremost among them was *Dr. Robin Braun*, who firstly selected me for the project and gave much help and advice throughout.

Many thanks also to *Garth Airey*, who partnered much of the research and gave inspiration when mine was lacking.

Credit is also due to *Elanix* for their *Systemview©* electronic system simulation software which made it possible to do much in little time.

I also wish to thank *Datafusion* for sponsoring my Masters degree.

Lastly, I wish to express heartfelt thanks to my family and friends for all their advice and entertainment.

Eugéne Nel

Cape Town

February, 1997

Page

iv Abstract

This dissertation investigates the synchronisation of Multi-h signals with a long constraint length. However, Multi-h codes exhibiting a long constraint length invariably also exhibit a large denominator q. Since traditional methods for synchronising Multi-h signals utilises a q-th power law device, where the frequency spectrum of the qth power of the signal renders all the necessary frequencies for synchronisation, we suspect that a large q could be detrimental.

When simulated, it turns out that the qth power law device fails to deliver distinct (and useable) frequency components at a q size of about 8. Unfortunately, the most useful codes have denominators starting at a size of q equals 32. This called for a novel approach to synchronisation.

One device that shows much potential is the new Massey-Hodgart coherent MSK demodulator. In a significant departure from standard quadrature structures, this MSK demodulator uses matched filter detection with a pair of reference signals at the two MSK signaling frequencies; an optimal maximum-likelihood bit decision is then formed over two bit intervals. The reference signals are recovered by a pair of decision-switched Costas loops, which are tightly integrated with the demodulator structure.

The goal was to modify the Massey-Hodgart MSK demodulator into a Multi-h synchroniser that contained matched filter detection for all the frequencies in the Multi-h signal. The reference frequencies would still be decision switched and recovered by Costas loops.

V List of Illustrations

Fig	Title	Page
2.1	Multi-h CPFSK in relation to other modulation schemes . .	3
2.2	Comparison between FSK and CPFSK waveforms . . .	5
3.1	Phase Trellis Diagram for $\{h_1, h_2\} = \{^1/_4, ^2/_4\}$ . . .	11
4.1	Flowchart of Cuthbert's [1] PBIL Search Algorithm . . .	16
4.2	Flow diagram for the modified PBIL Search Algorithm . .	20
4.3	3D surface plot of the new PBIL search results . . .	22
4.4	BER Curves for the *h*-sets of sizes 4 and 6 compared to that of MSK	24
5.1	Power Spectral Densities for the *h*-sets of sizes 4 and 2 . .	27
5.2	Power Spectral Densities for the *q*th powers of the *h*-sets of sizes 4 and 2	28
5.3	Finding θ_{best} by using a pair of correlators	30
5.4	Block Diagram of the entire synchroniser	32
5.5	Block Diagram of a single correlator from Figure 5.2 . .	33
5.6	Baud and Superbaud Recovery	36
5.7	The Receiver Input (1)	37
5.8	The Rectified Signal at (2)	37
5.9	The Signal at (3) after Lowpass Filtering, Biasing and Rectification .	37
5.10	The Superbaud Rate (4)	38
5.11	The Signal at (5) after Lowpass Filtering and in between Thresholding	38
5.12	The Baud Rate (6)	38

Page

vi List of Tables

Table	Title	Page
4.1	Cuthbert's PBIL *h*-sets	17
4.2	Theoretical performance limits for *h*-set sizes 2, 3, and 4	17
4.3	Cuthbert's corrected results for d^2_{min}, as calculated by the author	18
4.4	Results obtained from the new, refined PBIL algorithm	21
4.5	Comparison between new PBIL results and theoretical maxima	23

Table of Contents

Title Page i

Declaration of Authenticity ii

Acknowledgments iii

Abstract iv

List of Illustrations v

List of Tables vi

Chapter

1 **Introduction** 1
1.1 Background 1
1.2 Objectives of the Thesis 1
1.3 Organisation of the Thesis 2

2 **Theory of Multi-h CPFSK** 3
2.1 Multi-h CPFSK in relation to other modulation schemes 3
2.2 Arriving at a mathematical formulation for Multi-h. 4
2.3 Satellite Channels and Multi-h 7
2.3.1 Criteria for evaluating modulation schemes . 7
2.3.2 Is Multi-h a good choice for satellite channels? 8

3 **Decoding of CPFSK Signals** 10
3.1 Trellis Diagrams 10
3.2 The Viterbi Algorithm 11
3.3 Maximum Likelihood Detection 12
3.4 Probability of Error for the Correlator Receiver . 13
3.5 Total Probability of Error for Viterbi Decoding . 13

4 **Finding good *h*-sets with PBIL** 14
4.1 Desired Features of *h*-sets 14
4.2 Using Population-Based Incremental Learning (PBIL) to find good *h*-sets10 14
4.3 Discussion of *h*-sets obtained from PBIL . . 23

5 **Multi-h Signal Synchronisation** 26
5.1 A present method of synchronising Multi-h signals . 26
5.2 The Massey-Hodgart MSK Demodulator . . 29
5.3 Adapting the Massey-Hodgart Demodulator for Multi-h synchronisation 31
5.4 Baud/Superbaud Recovery 35
5.5 Injected Carrier to facilitate synchronisation . 39

6 **Summary Chapter** 40

Appendix

A **PBIL Search Algorithm Code** 43

B **Viterbi Decoder Code** 55

Bibliography 64

Chapter 1 Introduction

1.1 Background

Buzz words such as 'Information Age', 'Personal Empowerment' and 'Global Village' best describe why we are currently experiencing explosive growth in the construction of modern wireless digital communication networks. This area has seen, amongst others, the advent of digital cellular telephony, and the current thrust is towards mobile satellite communications systems with global coverage. The interest in these new communications systems has in turn prompted significant advances in high speed digital signal processors, RF hardware, sophisticated data compression and error correction algorithms and also in efficient new digital modulation techniques.

Satellite communication channels place demanding constraints on the choice of a digital modulation scheme, and one of the many new modulation schemes proposed for use on these channels is Multi-h CPFSK. (Multi-h CPFSK is a specific case of CPFSK where the modulation index changes every bit period to be equal to the next value of a fixed set of indices. For reasons explained later, these indices are fractions, all of which have the same denominator, q.)

To date demodulators have been developed for only the simplest of Multi-h codes. Since the increased efficiency of more complex codes may prove them to be well worth implementing, it is of considerable interest to find a method of synchronising such complex Multi-h signals that is both robust and reliable in practice.

1.2 Objectives of the Thesis

Recently, J. Cuthbert [1] investigated the existence of Multi-h codes that would give significant coding gains over even such efficient modulation schemes as MSK. He found a number of such codes with phenomenal coding gains, yet two problems regarding the demodulation of these codes were immediately apparent. Firstly, there was a tendency for the efficiency of a code to be proportional to its constraint length.

This roughly translates to the more efficient the code, the more complex the synchroniser and decoder at the receiver needs to be. Secondly, his codes all had a denominator q of 360. Since traditional Multi-h synchronisers facilitated a qth power law device, it is intuitively obvious that a new method of synchronising had to be found. The objective of this thesis then was to find such a new method of synchronisation.

However, after some initial work it became apparent that all was not well with the codes that needed to be synchronised. Apart from the fact that it seemed impossible to synchronise codes with both a large denominator and a long constraint-length, the codes themselves were in fact not worth synchronising. On closer investigation it turned out that due to calculation errors the codes had but meager coding gains over MSK - and was certainly not worth the substantial added complexity of Multi-h demodulation.

Another (and possibly more important) objective to this thesis thus came into existence. Not only do we need to find a method of synchronising the long constraint-length Multi-h codes, we first need to find such long constraint-length Multi-h codes worthwhile of synchronising.

1.3 Organisation of the Thesis

The chapters in the thesis all follow in logical order. Chapter 1 sets the scene with the introduction and some background information and Chapter 2 gives the necessary background Multi-h theory. Chapter 3 defines and explains all the necessary details for finding the new long constraint-length codes in Chapter 4. Chapter 5 contains all the synchronisation information, while Chapter 6 gives a summary of all the foregoing, and makes suggestions for further investigation.

Chapter

2 Theory of Multi-h CPFSK

In this chapter we introduce Multi-h CPFSK and explore its raison d'être.

2.1 Multi-h CPFSK in relation to other modulation schemes

Before describing and formulating Multi-h CPFSK mathematically, it is instructive to view its relationship to other well known modulation schemes. Figure 2.1 (adapted from [3]) shows how Multi-h can be derived from other more generalised modulation schemes. Note: whenever we refer to 'Multi-h' we mean Multi-h CPFSK, as to Multi-h CPM [4].

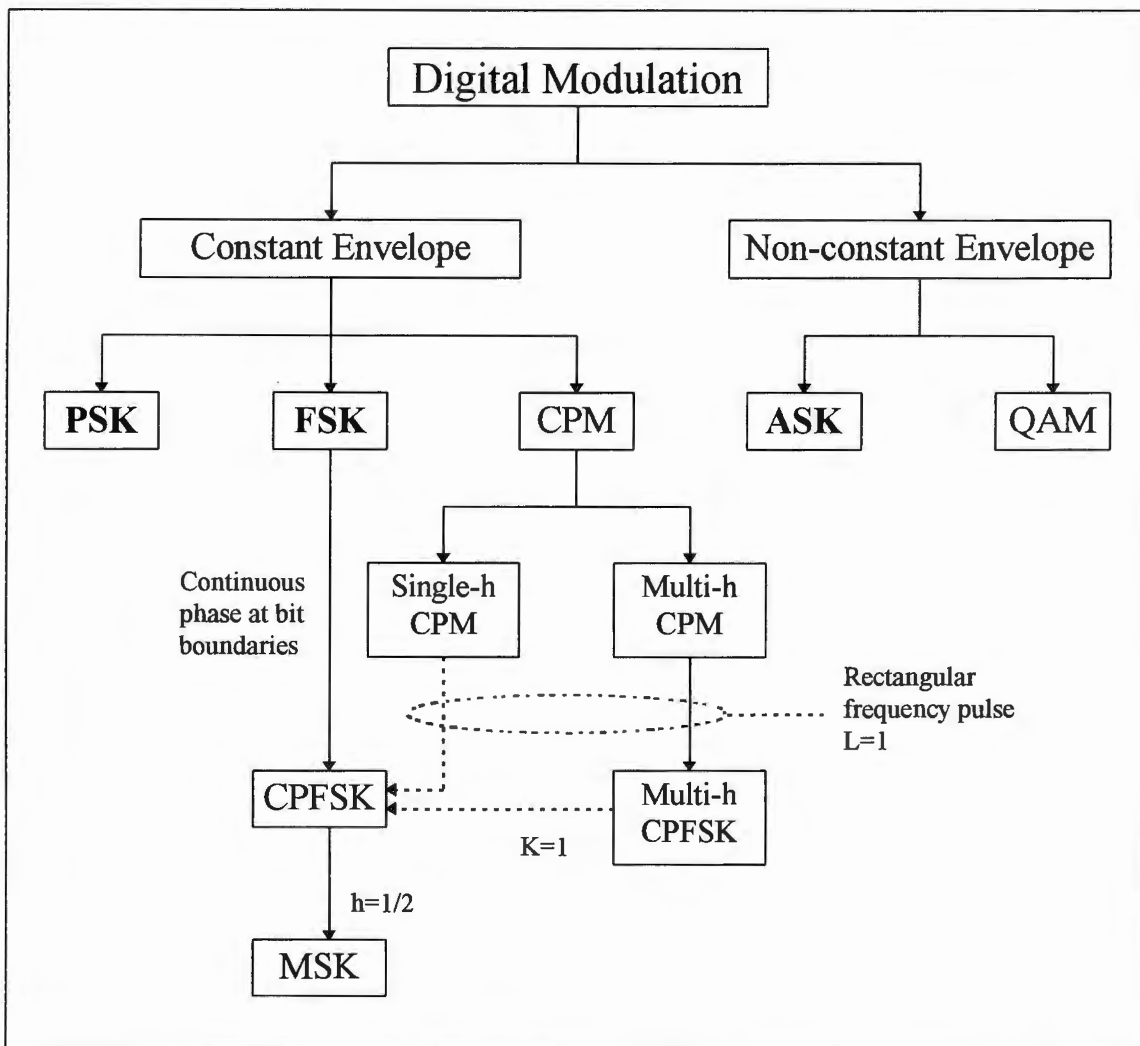


Fig 2.1: Multi-h CPFSK in relation to other modulation schemes. h is the modulation index, L is the pulse length in number of bit periods and K is the number of modulation indices for the code.

On the first level of the tree in Figure 2.1 a distinction is made between constant and non-constant envelope modulation schemes. In general a modulated signal retains its constant envelope only while that signal is unfiltered. Only continuous-phase signals retain a constant envelope after bandlimiting, while any signal with phase discontinuities will have a non-constant envelope after it is filtered. Schemes that guarantee a constant envelope have certain advantages over others. These advantages will be discussed in Section 2.3.1.

On the second level of the tree the three most basic modulation schemes are ordered: A sinusoidal signal can be modulated in amplitude, phase or frequency resulting in Amplitude Shift Keying (ASK), Frequency Shift Keying (FSK), and Phase Shift Keying (PSK) schemes. All the other schemes are variations and/or combinations of these fundamental schemes.

Finally, the tree shows where Multi-h CPFSK fits in. It is derived via CPM and has CPFSK as a special case. In reaching a mathematical formulation for Multi-h CPFSK it is most intuitive to start with FSK and proceed via CPFSK.

2.2 Arriving at a mathematical formulation for Multi-h CPFSK

First we consider **FSK**, which is a class of digital modulation in which the two binary symbols, '0' and '1', are represented by two different signaling frequencies. The signal is switched between these two frequencies in every bit interval depending on the binary input data. The modulated waveform, *s(t)*, for FSK is given by

$$s(t)_{FSK} = \sqrt{\frac{2E_b}{T_b}} \cos\left(2\Pi\left(f_c + x_k \Delta f\right)t\right) \tag{2.1}$$

where E_b is the bit energy, T_b is the bit period, f_c is the carrier frequency in hertz, Δf is the offset frequency and x_k is related to the current data bit by

$$x_k = -1 \quad \text{for binary '0'}$$

$$x_k = +1 \quad \text{for binary '1'} \tag{2.2}$$

Here we can also introduce the concept of the modulation index, *h*, which is the ratio of the difference between the two signaling frequencies and the bit period:

$$h = \frac{2\Delta f}{T_b} \tag{2.3}$$

CPFSK is the same as FSK in that it represents the binary data with sinusoids of different frequencies. However, in CPFSK the modulator is constructed in such a way as to ensure phase continuity of the modulated waveform at the bit boundaries. An example of an FSK waveform (with T_b = 1, f_c = 2.925, h = 1.65) and the equivalent CPFSK waveform is shown in Figure 2.2. We refer the reader to [5] for more information on continuous phase modulation.

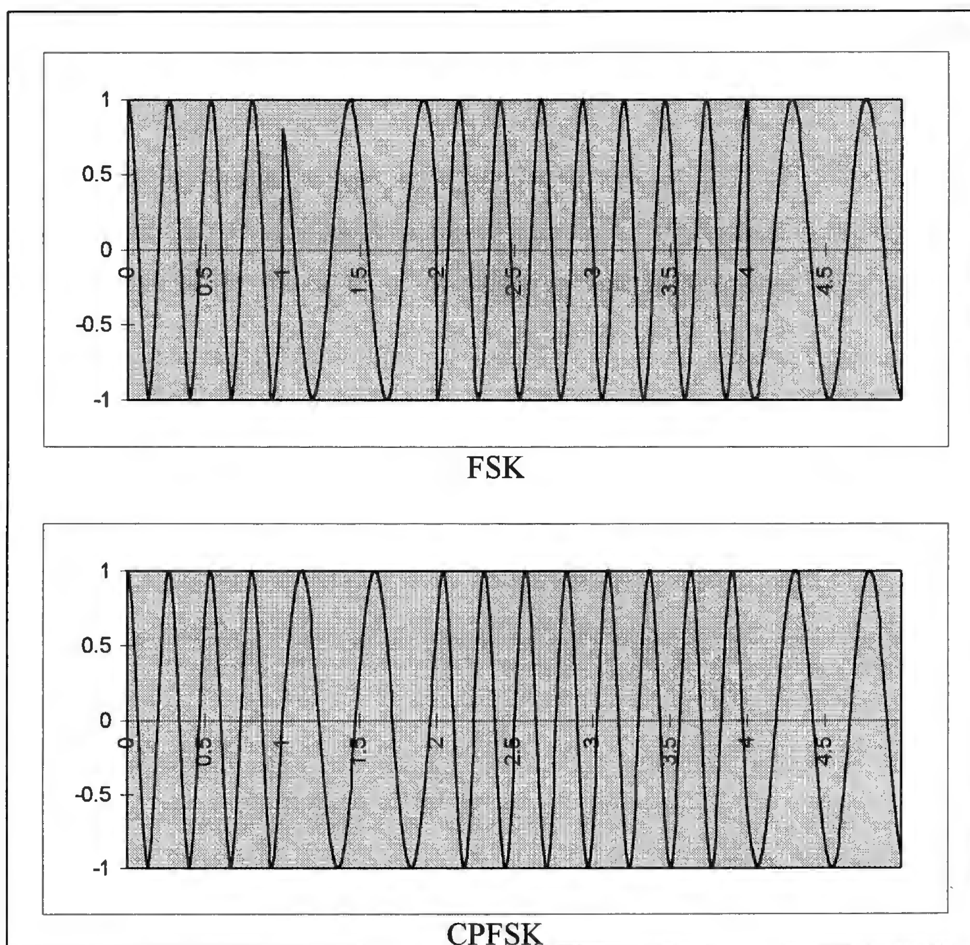


Fig 2.2: Comparison between FSK and CPFSK waveforms for T_b = 1, f_c = 2.925, h = 1.65

Multi-h CPFSK is a specific case of CPFSK where the modulation index, *h*, changes every bit period to be equal to the next value of a fixed set of indices. To keep the system manageable, the most common form of Multi-h is one in which the indices are chosen cyclically from a fixed set of arbitrary size *N* (the quantity *N+1* is referred to as the constraint length of the code).

Mathematically, Multi-h can be expressed as:

$$s(t)_{multi-h} = \sqrt{\frac{2E_b}{T_b}} \cos\left(2\Pi\left(f_c + \frac{x_k h_i T_b}{2}\right)t + \theta_k\right) \tag{2.4}$$

where h_i is the specific modulation index for bit x_k, chosen as

$$h_i \in \{h_1, h_2, \ldots, h_N\}$$
$$i = k \bmod N \tag{2.5}$$

and θ_k represents the accumulated phase up to the beginning of bit period *k*. This quantity is necessary to ensure smooth phase transitions between the successive CPFSK bit periods, and is equal to

$$\theta_k = \sum_{n=0}^{k-1} 2\Pi\left(f_c + \frac{x_n h_i T_b}{2}\right)T_b \tag{2.6}$$

The phase continuity adds an amount of redundancy into the signal. This redundancy can be exploited by the demodulator to help it in its bit decisions, thereby increasing Multi-h's performance in terms of the signal-to-noise ratio required to maintain a given probability of error at the demodulator.

2.3 Satellite Channels and Multi-h (adapted from [3])

Multi-h has been designed for use in an environment with a low signal-to-noise ratio. In this section we shall investigate one such case, being the satellite communications channel.

To understand why Multi-h is attractive for use on satellite channels it is instructive to review the constraints these channels place on the choice of modulation scheme, and then to determine how Multi-h meets these constraints.

2.3.1 Criteria for evaluating modulation schemes

In general we can determine the merits of digital modulation schemes in terms of the following criteria:

Bandwidth efficiency

As the demand for wireless communication services increases, the frequency spectrum becomes more and more congested. Most wireless communication systems are forced to operate in a limited slice of the frequency spectrum and as a result bandwidth efficient modulation schemes are required since they transmit as much information as possible in the given bandwidth.

Power efficiency

The number of bit errors made in receiving a modulated signal depends on the transmitted signal power and on the noise power introduced by the channel. Clearly, modulation schemes that minimise the probability of bit error for a given signal-to-noise ratio are preferred.

Synchronisation

One must recover the carrier phase, frequency and bit timing information from the signal when synchronising the receiver. The complexity of doing this depends on the modulation scheme, but the most efficiently synchronised schemes are those that are *self-synchronising*. Such schemes have characteristics that enable one to recover the carrier and clock from the modulated signal itself, without the need for a discrete signal

component at the carrier frequency, which would be wasteful of transmitter power and channel bandwidth.

Envelope characteristics

In systems where the available power is limited, it is common to use non-linear power amplifiers because of their high efficiency. Passing a signal with amplitude fluctuations through a non-linearity produces out-of-band sidelobes, which would interfere with adjacent channels. The use of such amplifiers thus calls for modulation schemes with a constant envelope.

System complexity

In the implementation of a digital communications system the overall system complexity depends on the requirements in terms of each of the various factors mentioned above. In general, systems that comply well to these requirements are much more complex to implement than systems with less stringent requirements. It is unfortunately true that more often than not the cost of implementing a good modulation scheme is the major reason that it is, in fact, not implemented.

2.3.2 Is Multi-h a good choice for satellite channels?

The most restrictive aspect of operating systems on a satellite is that the available power is usually severely limited. It is thus important that satellites should employ a modulation scheme that is power efficient. Due to the power constraints non-linear amplifiers are invariably used, further requiring that only modulation schemes with a constant envelope be used. This immediately limits the choice to one of the CPM schemes. Since a satellite channel is wireless, another important consideration is bandwidth efficiency.

Multi-h fares excellently in terms of both power efficiency and bandwidth efficiency and it also has a constant envelope. However, its improved performance goes hand-in-hand with improved circuit complexity and it is relatively complex to demodulate – even for simple codes.

Although Multi-h's characteristics make it ideal for use on a satellite channel (or any other low signal-to-noise ratio channel) from a *performance* consideration, its implementation complexity is a problem. Many satellites in orbit today use such inefficient schemes such as FSK quite simply because it is easier to add a little more power to your system than to have a system much more complex than is absolutely necessary.

Currently, a modulation scheme such as MSK is probably better suited to a *practical* implementation on satellite channels, since it trades off performance for a much simpler circuit complexity. (However, as a point of interest MSK is a special, simpler case of Multi-h.)

In this sense Multi-h is quite possibly ahead of its time.

Chapter

3 Decoding of CPFSK Signals

This chapter explores the decoding of CPFSK signals in the presence of Additive White Gaussian Noise (AWGN), thereby introducing the *minimum squared Euclidean distance* d^2_{min}, which is necessary when the results of chapter 4 is discussed. The Viterbi Algorithm is also discussed, since it is used both for finding h-sets as well as for maximum likelihood detection (see 3.3) at the receiver.

In any practical environment the signal transmitted through the communications channel will become corrupted by noise. In the simulation environment such noise is best modelled by AWGN. Additive noise will result in transmission errors. The purpose of the CPFSK decoder is to minimise the probability of error in the received signal.

3.1 Trellis Diagrams

Since every successive bit in a CPFSK signal relies on the frequency and phase of the bits preceding it, a certain amount of memory is introduced into the transmitted sequence of bits. The memory is expressed mathematically by the term θ_k in (2.4) and ensures more reliable decoding as the decoder can now consider a sequence of bits instead of one bit at a time. The concept of memory in CPFSK is best illustrated by a phase trellis diagram which plots the excess phase (with respect to the carrier frequency) of every possible transmitted sequence of bits starting at time $t = 0$. Each path in the trellis diagram originates from a common point with an excess phase of zero.

In Figure 3.1 we consider a specific set of h values $\{h_1, h_2\} = \{^1/_4, ^2/_4\}$ and its resultant phase trellis diagram.

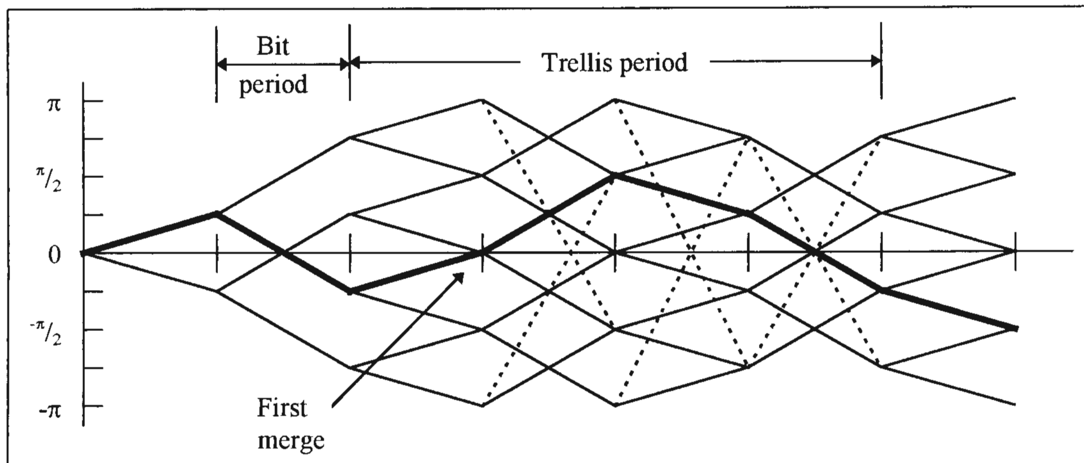


Fig 3.1: Phase Trellis Diagram for $\{h_1, h_2\} = \{^1/_4, ^2/_4\}$

The path shown in bold in Figure 3.1 corresponds to the bit pattern 1011000. Note that this path is unique until the end of the third bit period, where it merges with the path for the bit sequence 010. This means that the decoder has three bit periods instead of one in which to decide which of the two bit sequences are more likely to have been sent.

3.2 The Viterbi Algorithm

The Viterbi algorithm is an optimal method of finding the most likely transmitted path in the trellis diagram, i.e. it performs maximum likelihood detection on the convolutional code represented in the trellis diagram [6]. For the application of the Viterbi algorithm to the detection of Multi-h signals, the reader is referred to [7].

The Viterbi algorithm finds the most likely transmitted path by making a decision at every point in the trellis diagram where paths merge. It keeps track of all the possible paths starting at the previous merge and discards all but the most likely one ending at the next merge. It makes its decision as to which path is the most likely with the help of a quantity defined as the path metric. Each of the paths through he trellis has its own unique path metric, indicating how closely that path approximates the path actually received.

3.3 Maximum Likelihood Detection [4.1]

It is important that the path metric is calculated in such a way that guarantees optimal detection of the received signal, i.e. it must minimise the probability of error [4] when choosing between two possible paths.

It can be shown that the probability of error is a minimum when one consistently chooses the path with the smallest squared Euclidean distance from the received path [1], that is, when one chooses the squared Euclidean distance as the path metric. The squared Euclidean distance between two signals $s_1(t)$ and $s_2(t)$ is defined as

$$D^2 = \int (s_1(t) - s_2(t))^2 dt \qquad (3.1)$$

Equivalently,

$$D^2 = \int \left(s_1{}^2(t) - 2s_1(t)s_2(t) + s_2{}^2(t)\right) dt \qquad (3.2)$$

This is the quantity to be minimised in the detection process. Now, if $s_1(t)$ is the received signal and $s_2(t)$ is the path being considered, the quantity $s_1{}^2(t)$ may be ignored for minimisation purposes because it is constant for all the paths being considered. Also, $s_2{}^2(t)$ may be discarded if we have a constant modulation envelope, which we do have in CPFSK.

We are left with the term $-\int 2s_1(t)s_2(t)dt$ which means we have to maximise the quantity $\int s_1(t)s_2(t)dt$ which equals the correlation value between the two signals.

3.4 Probability of Error for the Correlation Receiver [4.2]

In practice we prefer finding the maximum for the correlation $\int s_1(t)s_2(t)dt$, since the quantity D^2 is difficult to calculate.

When one does this, the probability of choosing an incorrect path becomes

$$P_e = Q\left(\sqrt{\frac{D^2}{2N_0}}\right) \tag{3.3}$$

In general, one considers the normalised squared Euclidean distance

$$d^2 = \frac{D^2}{2E_b} \tag{3.4}$$

only. The probability of error is then

$$P_e = Q\left(\sqrt{\frac{E_b.d^2}{N_0}}\right) \tag{3.5}$$

3.5 Total probability of Error for Viterbi Decoding

For Viterbi decoding, the probability of error is dominated by the minimum squared Euclidean distance d^2_{min} among all possible paths in the phase trellis which originate at a common node and end at a subsequent merge.

In terms of Viterbi decoding the probability of error thus becomes

$$P_e \approx Q\left(\sqrt{\frac{E_b.d^2_{\min}}{N_0}}\right) \tag{3.6}$$

It should be noted at this point that the signal to noise ratio (E_b/N_0) required to give a particular P_e for CPFSK codes is generally referenced to the standard set by Minimum Shift Keying (MSK) which has a d^2_{min} of 2. Thus, any code with a d^2_{min} of greater than 2 will achieve improved power efficiency (or coding gain) over MSK.

Chapter

4 Finding good h-sets with PBIL

Since $Q(x)$ in (3.6) is a monotonically decreasing function with respect to x, the overall probability of error can be minimised by maximising d^2_{min}. As d^2_{min} in Multi-h depends on the elements and size (constraint length) of the h-set, the aim is to find values for both these quantities that will guarantee a high d^2_{min}. It can be shown that longer constraint-lengths generally result in higher values of d^2_{min}, although the law of diminishing returns apply. The reader is referred to [2] for an in-depth discussion on this topic.

4.1 Desired Features of *h*-sets [1]

Some of the features desired to ensure practical h-sets are summarised below:

- The indices h_i should be confined to $0 < h_i < 1$ to create codes that exhibit minimum bandwidth requirements and high power efficiency.
- The indices should be chosen as rational, i.e. $h_i = p_i/q$, where p_i, $q \in Z$ so as to make implementation manageable.

4.2 Using Population-Based Incremental Learning to find good *h*-sets

There are infinitely many practical h-sets, as per 4.1. However, very few of these are feasible when we consider physical implementation using available technology. Thus we define what we term to be a 'good' h-set – i.e. an h-set that is not only practical but also feasible for physical implementation. A good h-set will then have the following properties:

- Large d^2_{min} to ensure minimum probability of error.
- Small denominator q to allow ease of synchronisation and decoding.
- Small set size to keep modulator/demodulator complexity within manageable limits.

Recently, J.Cuthbert [1] set out to find such codes using a Population Based Incremental Learning (PBIL) algorithm. The PBIL algorithm is a variation on the

Genetic Algorithm [8], and is an efficient numerical method for approximating global maxima of multidimensional functions.

Basically, the PBIL algorithm works on the assumption that one will obtain a better parameter set when merging two good parameter sets. The first parameter sets are constructed randomly and each is then tested for its 'fitness'. The fittest sets are kept and merged to form new sets which, hopefully, will perform better than its two constituent sets. Random mutations are also introduced during this 'evolution' of sets in order to produce the best parameter set possible,

A detailed flowchart of [1]'s PBIL search algorithm illustrates the process in Figure 4.1.

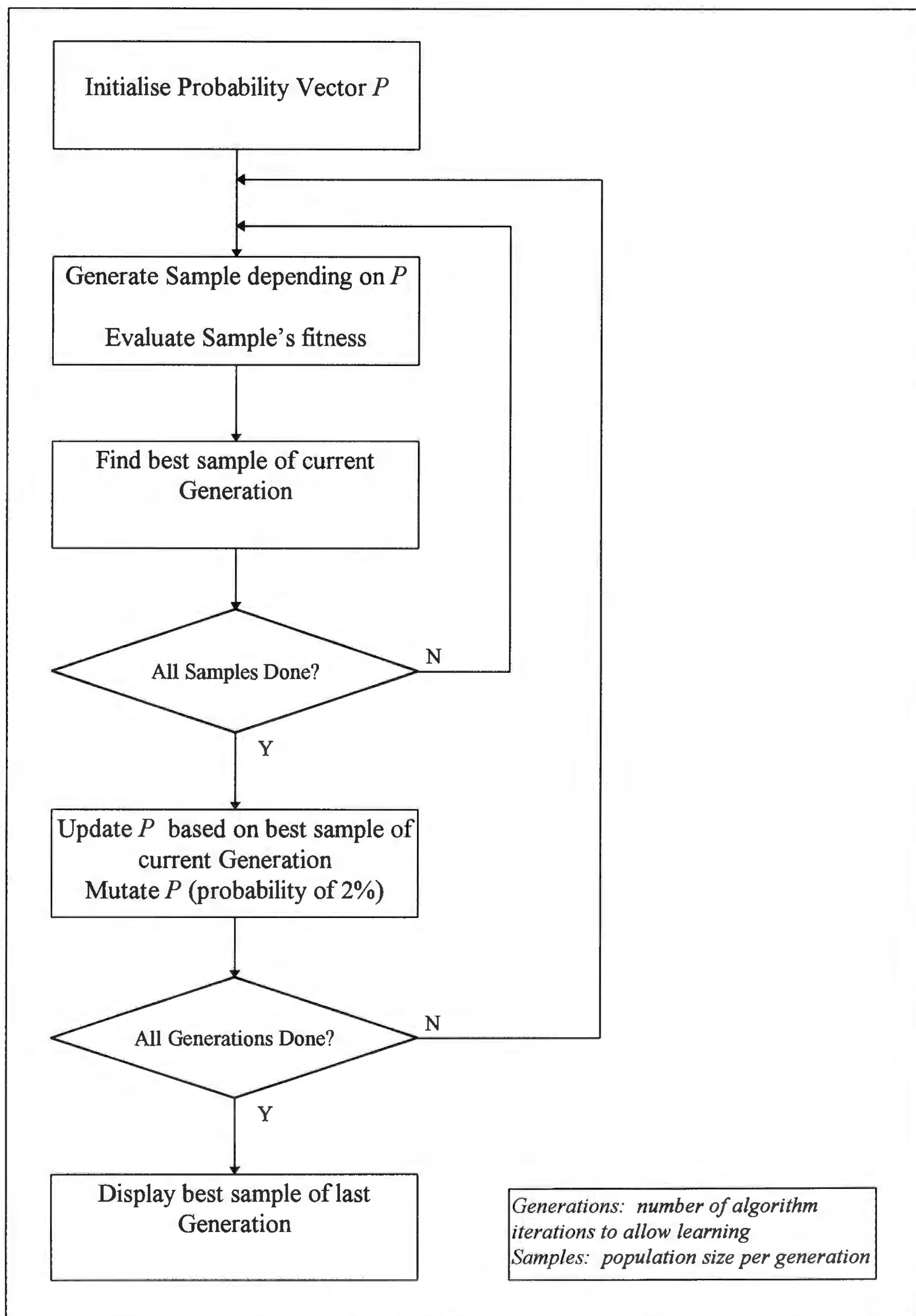


Fig 4.1: Flowchart of Cuthbert's [1] PBIL Search Algorithm

[1]'s results are set out in Table 4.1. Note that p_i in Table 4.1 represents the numerator of h_i for symmetric *h*-sets. [1] also considered unsymmetric *h*-sets with two sets of p_i – p_i^{+1} which is used to encode a binary '1' and p_i^{-1} for a binary '0'. The important result here is d^2_{min} (shown in bold).

Set Size	q	p_i^{+1} p_i^{-1} p_i (symmetric)	d^2_{min}	Gain over MSK in dB
3	360	331, 184, 60 57, 208, 331	**6.083**	4.831
3	360	195, 194, 197	**6.083**	4.831
4	360	79, 84, 213, 100 302, 304, 169, 286	**8.054**	6.050
4	360	182, 202, 207, 200	**7.924**	5.979
5	360	150, 196, 49, 18, 227 257, 208, 359, 342, 160	**9.883**	6.938
5	360	207, 176, 198, 209, 194	**9.764**	6.886
6	360	316, 253, 211, 90, 328, 206 56, 123, 199, 308, 83, 211	**11.656**	7.655
6	360	180, 178, 207, 160, 164, 224	**10.194**	7.073
7	360	18, 114, 148, 114, 142, 102, 337 340, 287, 278, 246, 311, 180, 43	**12.151**	7.836
7	360	249, 191, 246, 217, 195, 175, 180	**10.580**	7.234
8	360	18, 114, 148, 114, 250, 129, 102, 337 340, 287, 278, 246, 20, 215, 180, 43	**12.000**	7.782

Table 4.1: Cuthbert's PBIL *h*-sets

However, Anderson, Aulin & Sundberg [2] have done exhaustive searches for *h*-set sizes of one, two, three, and four with a q of 1000 and found the following theoretical limits as far as maximisation of d^2_{min} is concerned:

Set Size	q	p_i (symmetric)	d^2_{min}	Gain over MSK in dB
1	1000	715	**2.43**	0.85
2	1000	500, 500	**4.00**	3.00
3	1000	620, 686, 714	**4.88**	3.87
4	1000	730, 550, 730, 550	**5.69**	4.54

Table 4.2: Theoretical performance limits for *h*-set sizes 2, 3, and 4

Now, it is intuitively obvious that if we compare two optimal codes with different denominators q, then we can expect that the code with the larger denominator will

exhibit the larger d^2_{min} (since more phases are attainable before the first inevitable merge, see Figure 4.3).

However, from the above two tables we see that [1] quotes values for d^2_{min} for *h*-set sizes of three and four that are supposedly above the theoretical limit found by [2], although [1]'s denominator is smaller than [2]'s.

After careful investigation it was established that Cuthbert was employing an inadequate method of determining d^2_{min}, and that the maximum values quoted by [2] are indeed correct. Cuthbert's fault lies in that he only considered the first inevitable merge in the phase trellis diagram when calculating d^2_{min}. However, it can be shown [1] that the first inevitable merge does not necessarily produce the lowest d^2_{min} in the code, and that one needs to consider the merges during the subsequent few symbol periods as well. (The reader will notice that the reference to this correct way of determining d^2_{min} is in fact Cuthbert. We thus assume his faulty results were due to an oversight.)

This not only means that [1]'s values for d^2_{min} are incorrect, but as a result his PBIL 'fitness factor' for determining optimal *h*-sets is in error. (I.e. the codes under consideration are probably not optimal.) When a more appropriate technique for determining d^2_{min} is used, [1]'s results reduce to the following (for the symmetric case):

Set Size	q	p_i (symmetric)	d^2_{min}	Gain over MSK in *dB*
3	360	195, 194, 197	**3.84**	2.83
4	360	182, 202, 207, 200	**3.91**	2.91
5	360	207, 176, 198, 209, 194	**3.93**	2.93
6	360	180, 178, 207, 160, 164, 224	**3.57**	2.52
7	360	249, 191, 246, 217, 195, 175, 180	**3.96**	2.97

Table 4.3: Cuthbert's corrected results for d^2_{min}, as calculated by the author

Since a PBIL search algorithm should be able to find values for d^2_{min} which approach the theoretical maxima when correctly implemented, it was decided to revisit Cuthbert's PBIL search algorithm in an attempt to find good *h*-sets for the symmetrical case. (Only the symmetrical case was considered due to the added complexity of the unsymmetrical case.)

For the purposes of the PBIL search d^2_{min} was calculated by Viterbi decoding a random bitstream and noting the differences in path metrics between the bitstream and all paths merging with it in the phase trellis.

The flow diagram in Figure 4.2 shows the following changes to the PBIL algorithm: Firstly, experimental results suggested that it is best to mutate the probability vector only when it was obviously stuck on a local maximum. This was facilitated by checking whether a generated sample was equal to the best sample found thus far. If so, a mutation counter was incremented and the current sample replaced by a completely *random* one. At the end of the generation a mutation was forced if the mutation counter exceeded a given threshold.

Another change is that the Probability Vector P is now updated with the best sample found from *all* generations instead of just from the current generation. This is in fact a fundamental change away from the actual PBIL algorithm. The rationale behind this change was that, since the algorithm was already modified to recover quickly when stuck on a local maximum, it was in our best interests to reach the local maximum as quickly as possible. (When P is updated by the best sample from only the current generation, it had the effect of 'wandering' around the local maximum before finally reaching it.)

Results from the modified PBIL algorithm are listed in Table 4.4.

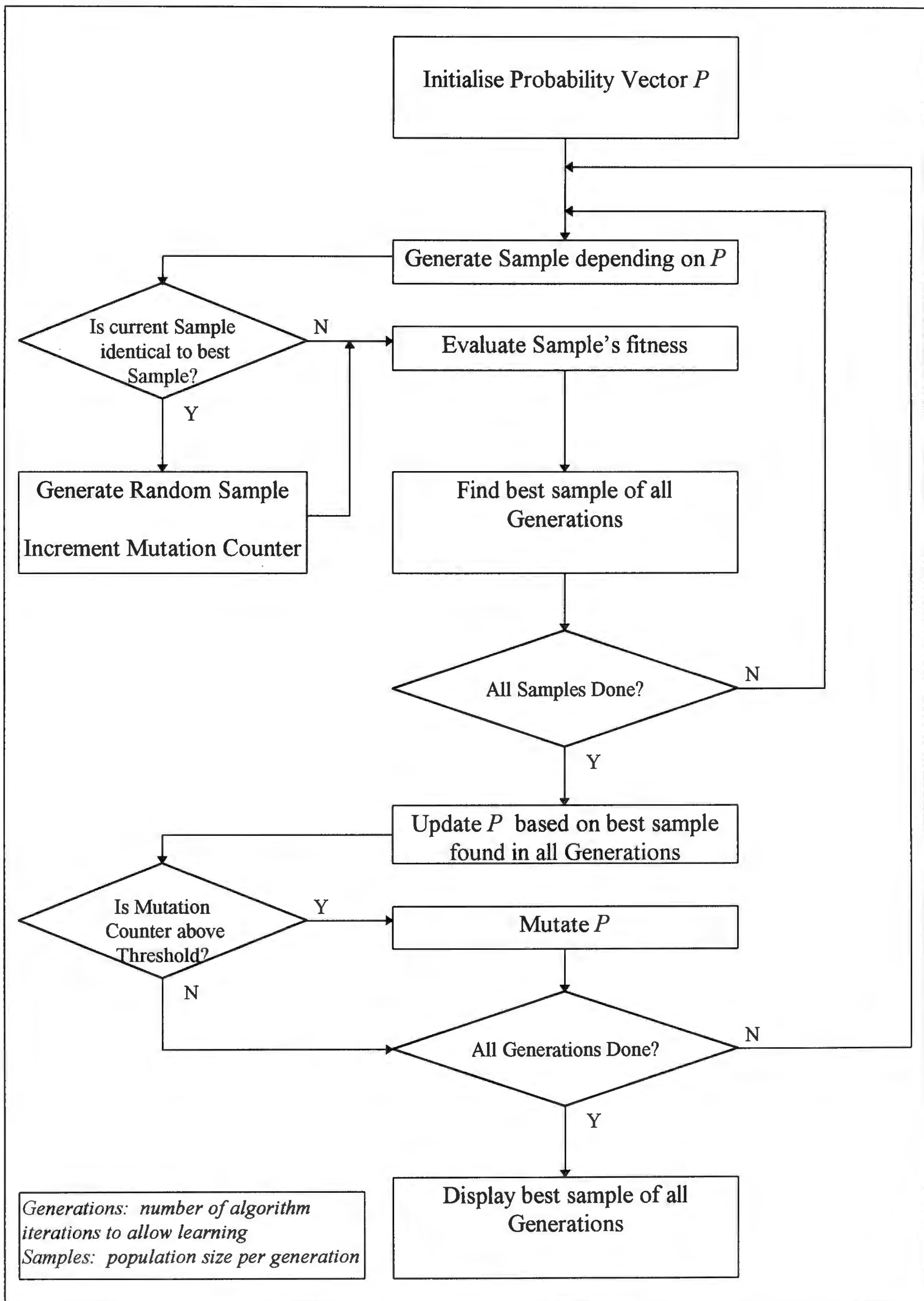


Fig 4.2: Flow diagram for the modified PBIL Search Algorithm

Set Size	q	p_i (symmetric)	d^2_{min}	Gain over MSK in dB
2	8	5, 4	3.55	2.49
	16	9, 8	3.76	2.74
	32	17, 16	3.88	2.88
	64	33, 32	3.94	2.94
	128	65, 64	3.97	2.98
	256	129, 128	3.98	2.99
3	8	5, 4, 6	3.79	2.78
	16	11, 12, 10	4.34	3.36
	32	23, 22, 21	4.57	3.59
	64	44, 43, 45	4.85	3.85
	128	89, 91, 82	4.87	3.86
	256	163, 177, 183	4.87	3.86
4	8	4, 6, 4, 5	3.75	2.73
	16	10, 11, 8, 12	4.65	3.66
	32	24, 20, 23, 18	5.16	4.12
	64	37, 48, 39, 47	5.40	4.31
	128	72, 94, 74, 95	5.58	4.46
	256	187, 144, 188, 146	5.68	4.53
5	8	5, 4, 5, 4, 6	3.55	2.49
	16	10, 8, 10, 12, 9	4.07	3.09
	32	22, 20, 25, 16, 24	5.03	4.01
	64	44, 32, 48, 41, 50	5.26	4.20
	128	69, 99, 81, 100, 86	5.52	4.41
	256	200, 153, 195, 137, 171	5.57	4.45
6	8	6, 5, 4, 6, 5, 4	3.79	2.78
	16	12, 11, 10, 12, 11, 10	4.34	3.36
	32	26, 23, 18, 25, 22, 20	4.97	3.95
	64	33, 42, 51, 33, 50, 43	5.57	4.45
	128	102, 64, 100, 78, 97, 70	5.93	4.72
	256	152, 190, 155, 201, 162, 199	6.09	4.84
7	8	6, 4, 5, 4, 6, 5, 4	3.75	2.73
	16	12, 10, 11, 8, 12, 10, 11	4.34	3.36
	32	26, 23, 22, 20, 24, 16, 21	4.96	3.94
	64	42, 47, 36, 42, 48, 32, 52	5.35	4.27
	128	105, 89, 72, 99, 81, 102, 68	5.79	4.62
	256	153, 192, 142, 200, 169, 132, 193	6.14	4.87
8	8	4, 5, 6, 4, 5, 4, 6, 5	3.84	2.83
	16	10, 11, 8, 12, 10, 11, 8, 12	4.65	3.66
	32	23, 20, 22, 24, 17, 26, 22, 24	4.95	3.94
	64	37, 48, 39, 47, 37, 48, 39, 47	5.40	4.31
	128	84, 97, 81, 101, 69, 107, 81, 103	5.73	4.57
	256	132, 194, 166, 207, 180, 147, 200, 169	6.09	4.84

Table 4.4: Results obtained from the new, refined PBIL algorithm

As a graphical aid, the values for d^2_{min} in Table 4.4 are presented below in a 3D surface plot. Note the tendency for d^2_{min} to increase with longer constraint-length and larger denominator q.

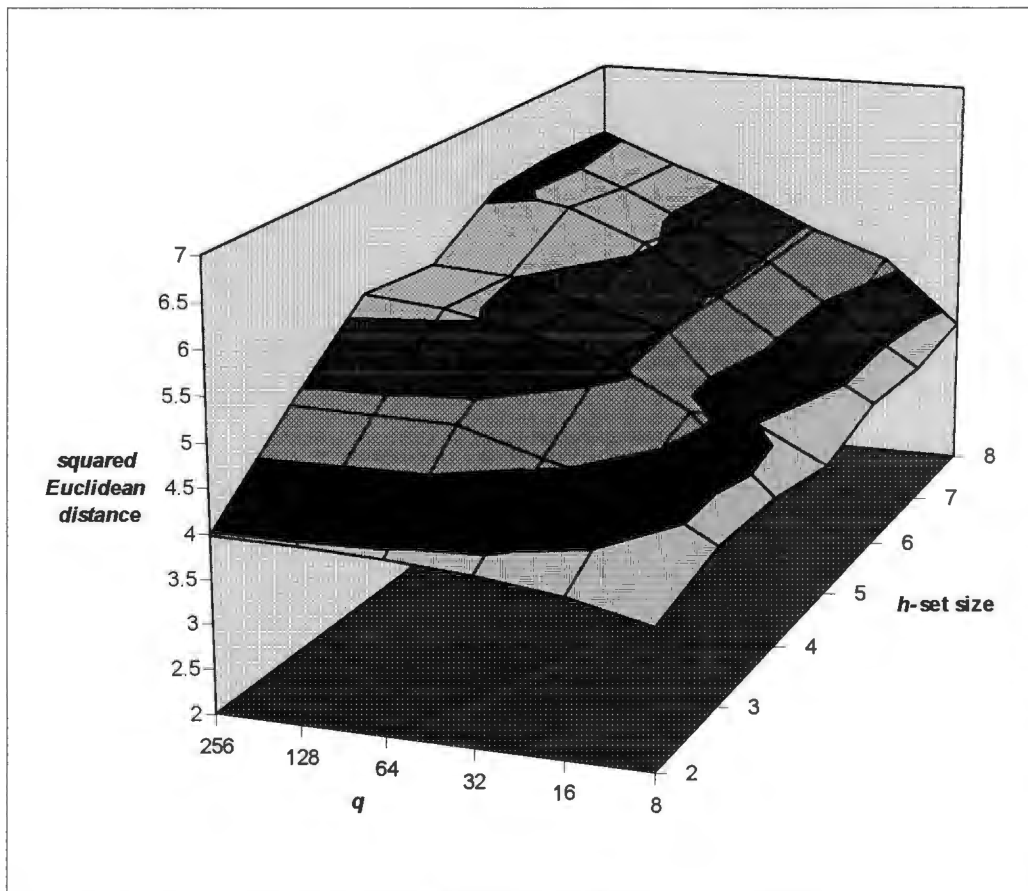


Fig 4.3: 3D surface plot of the new PBIL search results

4.3 Discussion of *h*-sets obtained from PBIL

In Table 4.5 below the normalised results for h-set sizes 2, 3, and 4 with a denominator of 256 from Table 4.4 are compared to the theoretical maximum values listed in Table 4.3.

	PBIL Results		Theoretical Maxima	
h-set size	h_i (symmetric)	d^2_{min}	h_i (symmetric)	d^2_{min}
2	0.50, 0.50	3.98	0.50, 0.50	4.00
3	0.64, 0.69, 0.71	4.87	0.62, 0.69, 0.71	4.88
4	0.73, 0.56, 0.73, 0.57	5.68	0.73, 0.55, 0.73, 0.55	5.69

Table 4.5: Comparison between new PBIL results and theoretical maxima

Since the PBIL results approximate the theoretical maxima to within 0.5% for the h-set sizes of 2, 3, and 4, it is reasonable to assume that the results for the larger h-set sizes are also close to their respective (and as yet unknown) maxima.

What remains to be seen in chapter 5 is whether the codes in Table 4.4 are conducive to use in a practical environment, i.e. whether they can be synchronised at the receiver.

For both synchronisation and decoding, receiver complexity is directly proportional to denominator size, as this determines phase resolution of the modulated signal. That means that even though the values for d^2_{min} resulting from a large denominator are invariably better than those from a small one, the increased receiver complexity may outweigh the gain in performance.

With these factors in mind we shall examine the synchronisation of those PBIL results deemed most useful – those with a small denominator size as well as large d^2_{min}. Two codes from Table 4.4 which fall into this category are for h-set size 4 with denominator 32, and h-set size 6 with denominator 64. The BER curves for these two codes, derived with a Viterbi decoder with random input and varying signal to noise ratios, are shown in Figure 4.4 overleaf. (The program listing for this Viterbi decoder, which has been implemented in the 'C' programming language has been included in Appendix B.)

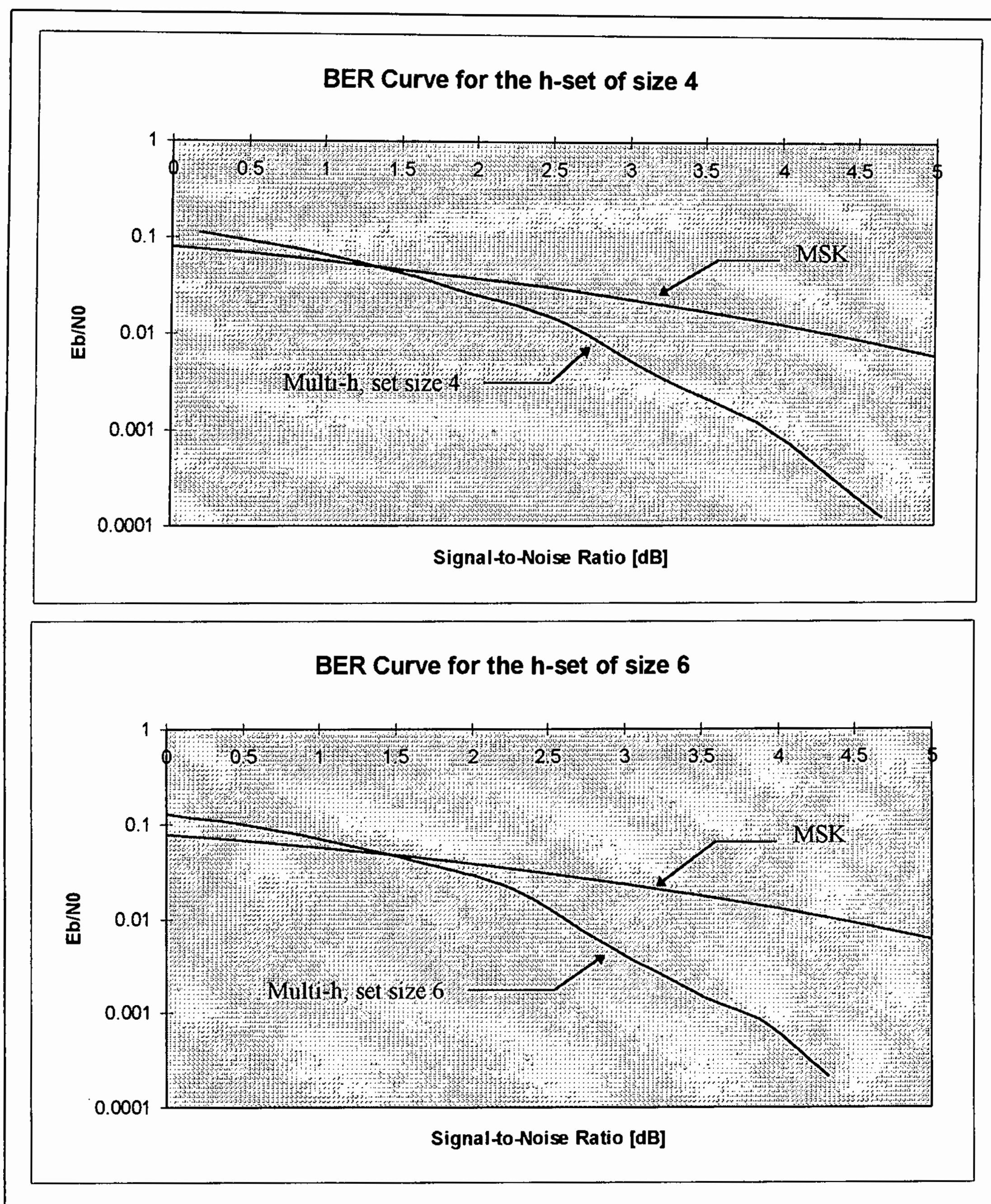


Fig 4.4: BER Curves for the h-sets of sizes 4 and 6 compared to that of MSK

When the foregoing BER curves were constructed via the Viterbi decoding simulation it became apparent that the code of h-set size 6 have characteristics that makes it very difficult to decode.

Although it is not the purpose of this work to explain the reasons why one code is more difficult to decode than another, we will explore the reason for this specific case. When we consider the elements (24,20,23,18) of the code with 4 elements in its h-set, we notice that the elements 24 and 23 are numerically close, as are 20 and 18. When

we now consider the elements (33,42,51,33,50,43) of the code with h-set size 6 we notice that 33 equals 33, 51 is almost equal to 50, as is 42 to 43.

Now, when we consider that these elements determine the paths in the phase trellis, we realise that some of the paths in the trellis for the 4 h-set will stay relatively close together due to the set elements being relatively close numerically. However, since the h-set with 6 elements have 3 sets of elements that are almost equal, it was found that the paths in its phase trellis diagram stay very close together in some cases for up to 5 times longer than with the code of size 4.

This of course means that the Viterbi decoder needs much more memory and will take much longer to successfully decode the code of size 6. Since we can conclude that this specific code is prohibitive to decode, we shall only be investigating the synchronisation of the code with denominator q equals 32 and h-set size of 4.

Chapter

5 Multi-h Signal Synchronisation

In this chapter we shall investigate the receiver synchronisation of the following code from Table 4.4:

- Set Size **4**, Denominator **32**, Set Elements **24**, **20**, **23**, **18**.

Three levels of synchronisation are required in the demodulation of Multi-h CPFSK signals. These are:

1. carrier phase synchronisation,
2. symbol or baud timing, and
3. superbaud timing (time between the *h*-set repeating itself)

5.1 A present method of synchronising Multi-h signals

One such method by which we can easily acquire all this information is to pass the signal through a qth power law device, where q is the denominator of the *h*-set. The spectrum of the output has discrete spectral components at specific frequencies surrounding the carrier frequency multiplied by q. These adjacent spectral components can be manipulated to yield the baud and superbaud timing, as well as the carrier phase synchronisation. (For a detailed description of this procedure see [4.1].)

However, from the outset it is intuitively obvious that such a method is not feasible for the code we need to investigate. Since our code has a denominator of 32 we need to employ a device that provides the 32nd power of the spectrum with frequency spikes at 32 times the carrier frequency!

Even though such a system would never work in practice (due to the enormous computational effort involved), we investigated using simulations designed with the Systemview© software package. In this simulation no noticeable power was present at the 32nd harmonic, which would suggest that such a method is not practical.

Upon further investigation it became clear that there are no usable spectral components at q times their carrier frequency for codes with a constraint length greater than 4. Also, there are in fact no noticeable components at the 32nd harmonic of any code, regardless of constraint length.

Since we seek to investigate codes with long constraint-lengths - and since the best of these codes have large denominators q, this method is of no use to us. Below in Figure 5.1 we show the power spectral densities of the h-set of size 4 with q 32, along with that of an arbitrary code of size 2 with q equals 4. The carrier frequency is 10Hz in both cases.

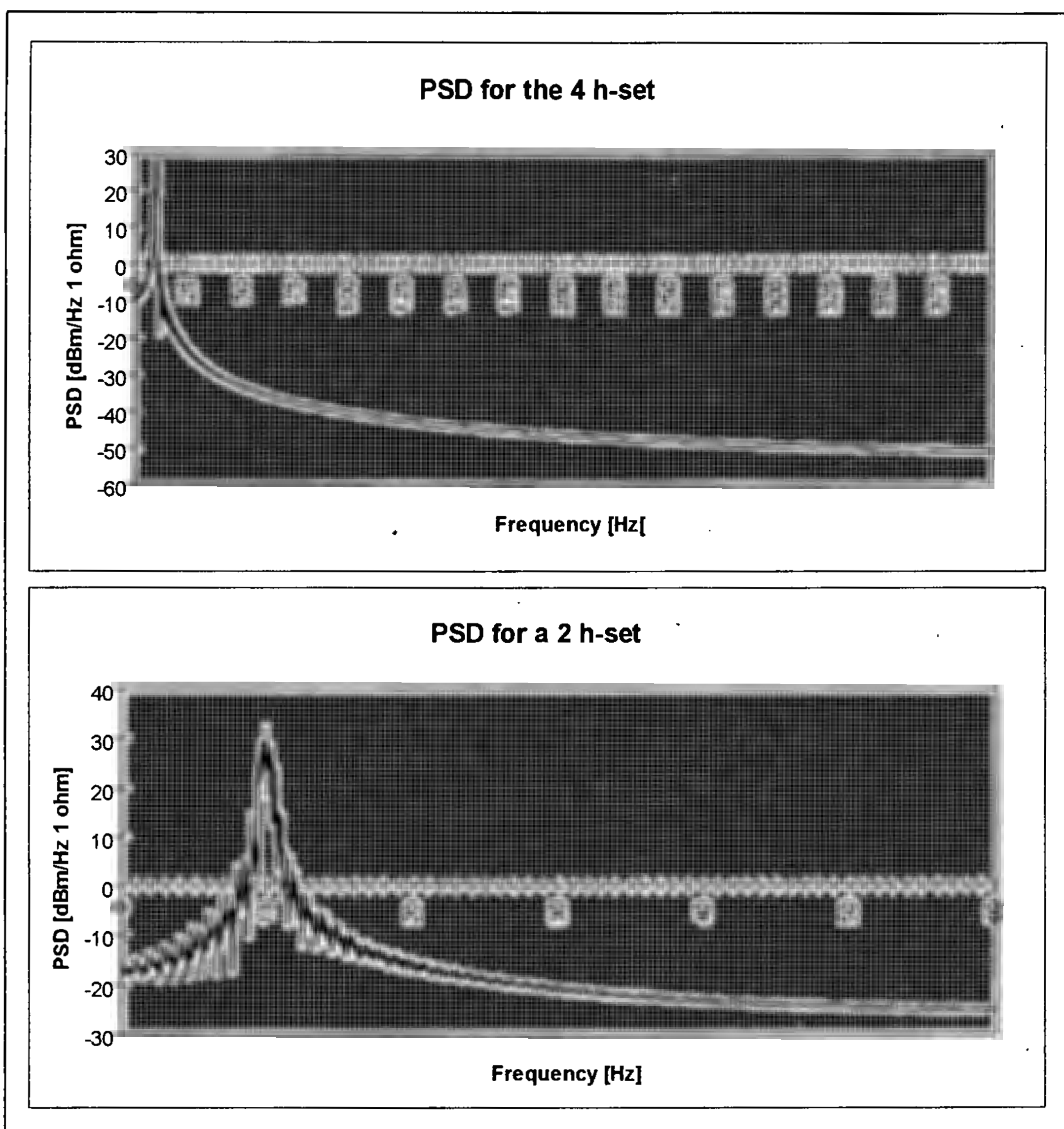


Fig 5.1: Power Spectral Density for the h-set of size 4 along with that for an h-set of size 2

In Figure 5.2 below we show the power spectral densities of the qth powers of the two codes from Figure 5.1. Note the discrete spectral components at 40Hz for the code of size 2, while the other code have no discrete spectral components at 320Hz.

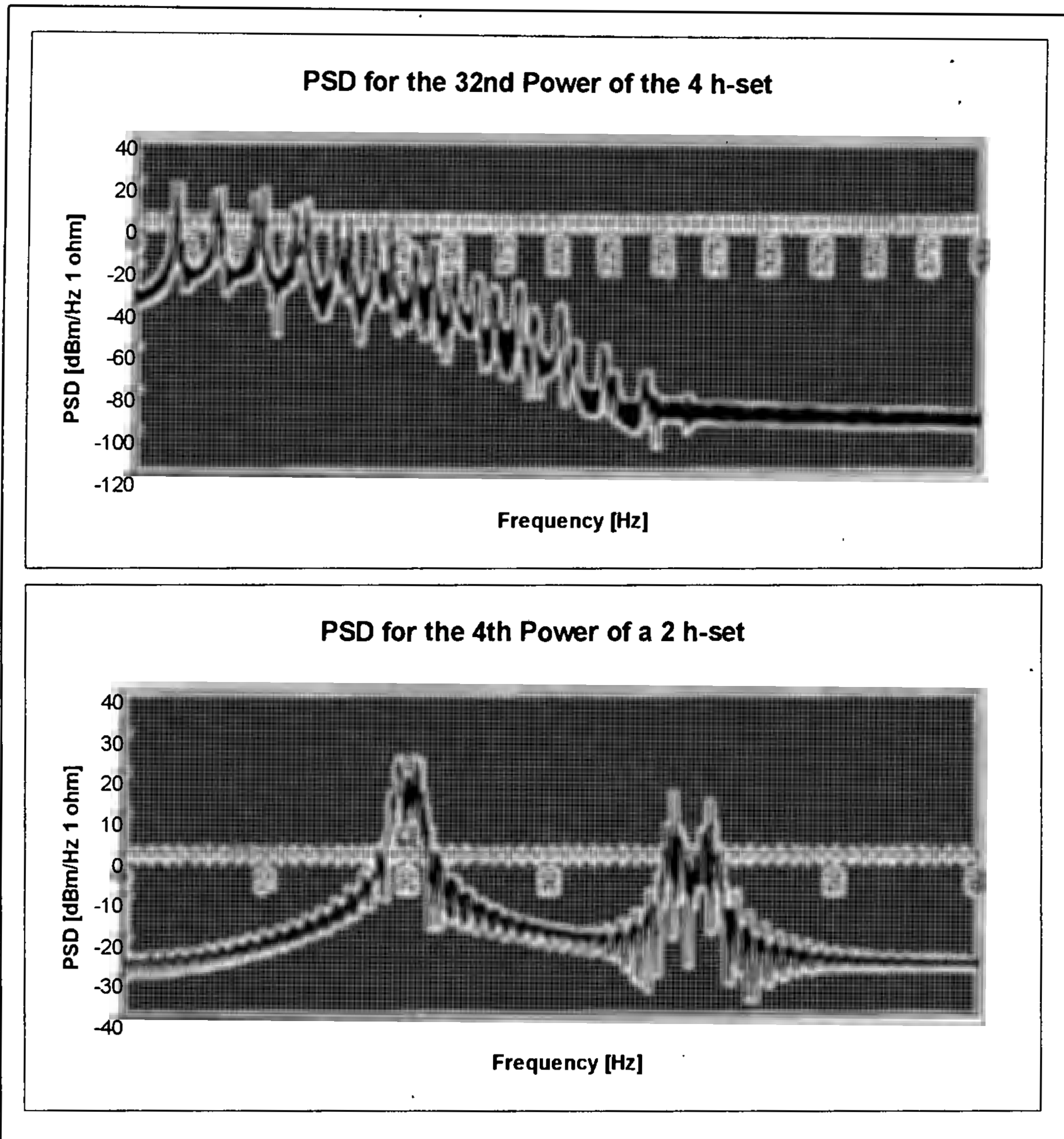


Fig 5.2: Power Spectral Densities for the qth powers of the h-sets of sizes 4 and 2

5.2 The Massey-Hodgart MSK Demodulator

Due to the complexity of the h-sets we need to investigate, it is extremely difficult to derive the carrier frequency (with correct phase!) from the constituent frequencies. A hypothesis was developed that it may be more feasible to work with the constituent frequencies themselves instead of attempting to derive the carrier frequency.

A device that facilitates this and was proved to work in practice for MSK is the Massey-Hodgart MSK Demodulator. [10] contains a detailed description of this demodulator, and [3] contains a working DSP implementation of such a system.

The Massey-Hodgart Demodulator uses two Costas loops (a form of phase-locked loop) to track the two MSK signaling frequencies independently. A significant departure from the standard form of a Costas loop is that the loops used in the Massey-Hodgart demodulator are *decision-switched.* Although each loop tracks only one of the MSK signaling frequencies, the input is an MSK signal consisting of symbols at *either* frequency. Clearly, the phase-error estimates will be meaningless when the incorrect frequency symbol is present at the input of a loop, and thus invalid estimates must be ignored.

The decision-switching of the Costas loops is done by a data sequence that has a one-to-one correspondence with the frequency of the received MSK symbols. Basically, during the first bit period that a symbol arrives at the demodulator, it is decided which frequency is present. During the second bit period the corresponding Costas loop is connected and the delayed symbol is used to keep that loop phase-synchronised.

A proof that the Costas loop is an optimal way of estimating a signal with unknown phase [9] is presented next. Thereafter we discuss the adaptation of the Massey-Hodgart demodulator for Multi-h synchronisation.

Proof: Costas loop is optimal at estimating signals with unknown phase

Consider a signal of known form

$$s(t,\theta) = a_c \sin(2\pi f_c t + \theta) \qquad 0 \le t \le T \tag{5.1}$$

where the amplitude a_c and frequency f_c are known. The Problem is to estimate the unknown phase θ. From (5.1) we find that the maximum likelihood estimate θ_{best} of the phase θ is the solution of

$$\int_0^T [x(t) - a_c \sin(2\pi f_c t + \theta_{best})] \cos(2\pi f_c t + \theta_{best}) dt = 0 \tag{5.2}$$

We assume that $2f_c T = k$ where k is an integer. Then the integral of the second term in (5.2) is zero, so that the phase estimate θ_{best} is the solution of

$$\int_0^T x(t) \cos(2\pi f_c t + \theta_{best}) dt = 0 \tag{5.3}$$

Expanding the cosine term in the integrand in (5.3) and rearranging terms, we get

$$\cos\theta_{best} \int_0^T x(t) \cos(2\pi f_c t) dt = \sin\theta_{best} \int_0^T x(t) \sin(2\pi f_c t) dt \tag{5.4}$$

Solving for θ_{best} yields the desired estimate:

$$\theta_{best} = \tan^{-1} \left[\frac{\int_0^T x(t) \cos(2\pi f_c t) dt}{\int_0^T x(t) \sin(2\pi f_c t) d} \right] \tag{5.5}$$

The operations in (5.5) may be performed by using a pair of correlators or filters matched to $cos(2\pi f_c t)$ and $sin(2\pi f_c t)$, as indicated in Figure 5.3.

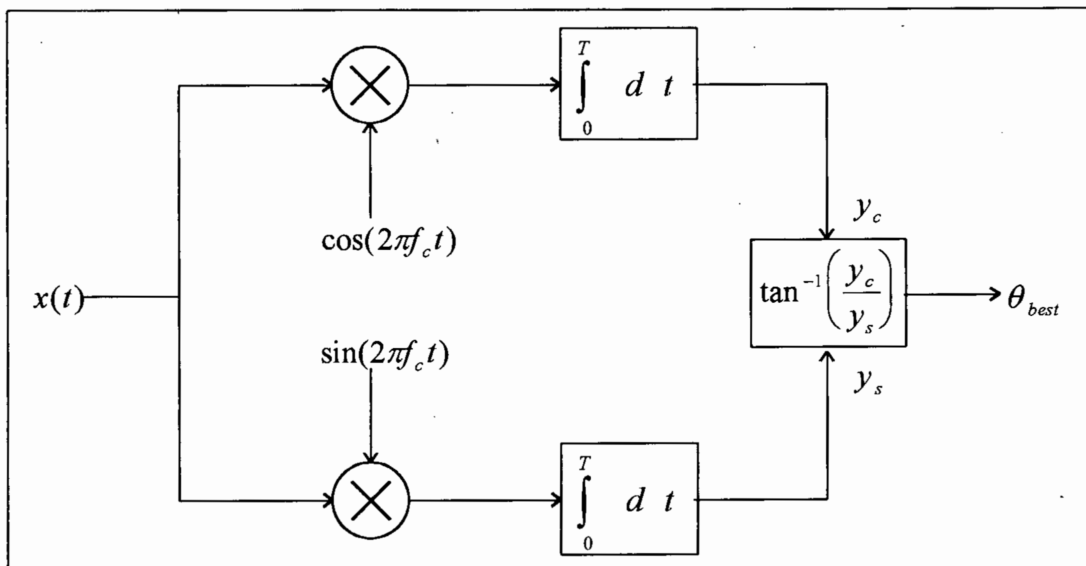


Fig 5.3: Finding Θ_{best} by using a pair of correlators

5.3 Adapting the Massey-Hodgart MSK Demodulator for Multi-h Synchronisation

Since we know that the Massey-Hodgart demodulator works very well with MSK, the task now is to adapt this system to work with a modulation scheme substantially more complex than MSK.

Since each Costas loop tracks a single frequency, we need 8 Costas loops for the *h*-set of size 4 (we have 8 frequencies) instead of just 2 as for MSK (which has 2 frequencies). The decision logic that switches the Costas loops also needs to be redesigned. The characteristics of MSK ensures a simple and clever implementation of this decision-switching (see [3]). However, our Multi-h signals have no such convenient characteristics.

In Figure 5.4 overleaf the system is shown from transmission to the receiver's Viterbi Decoder interface. The Viterbi Decoder accepts the correlation values from all the correlators, obtained by multiplying the received signal with both the in-phase and quadrature correlator signals.

The correlator modules contain the adapted Massey-Hodgart demodulator. The final adapted synchroniser is shown in Figure 5.5. The synchronisers basically work as follows: First the incoming signal is multiplied by the in-phase and quadrature correlator frequencies and integrated. If the sum of these integrals are above a predetermined threshold the received frequency is that of the correlator and the Costas loop for that correlator is connected.

However, the incoming signal is usually of different phase to the correlator frequency. For the *h*-set of size 4 the denominator is 32. Since we need to cater for negative frequencies as well, the phase of the incoming signal can have any of 64 different phases. The 4 quadrant arctan in the diagram outputs the real phase difference between the correlator and received frequencies. The 6 bit quantiser then calculates the phase difference between the correlator frequency and the closest permissible phase of the

received frequency. This phase difference is fed into the Costas loop as the error to be corrected.

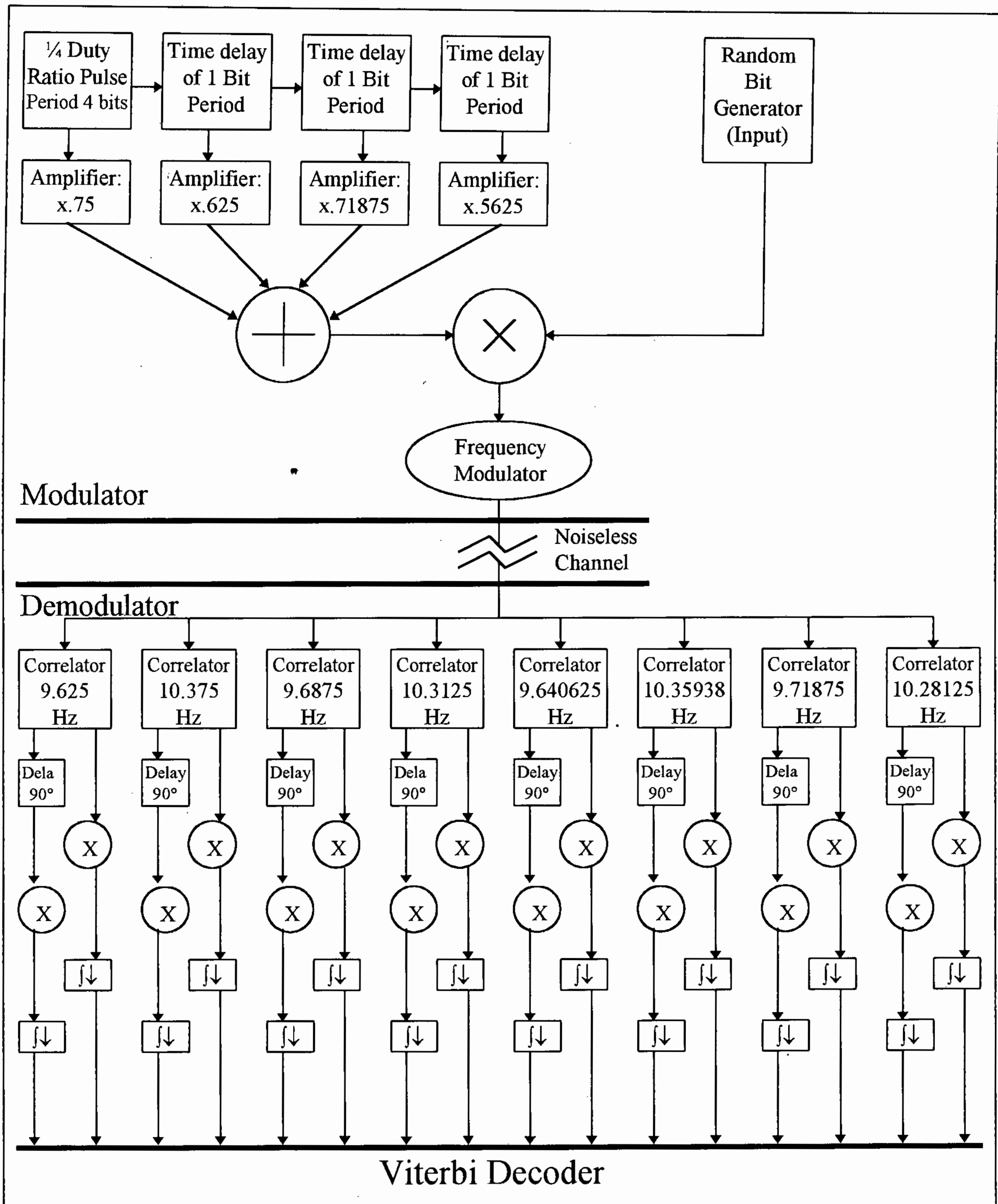


Fig 5.4: Block Diagram of the entire synchroniser.
NOTES:
The multipliers after the correlator modules have the received signal as their second input.
∫↓ represents an 'integrate and dump' module.
The Viterbi Decoder receives two values from each of the correlators: The first is the quadrature correlation with the received signal, and the second the in-phase correlation.

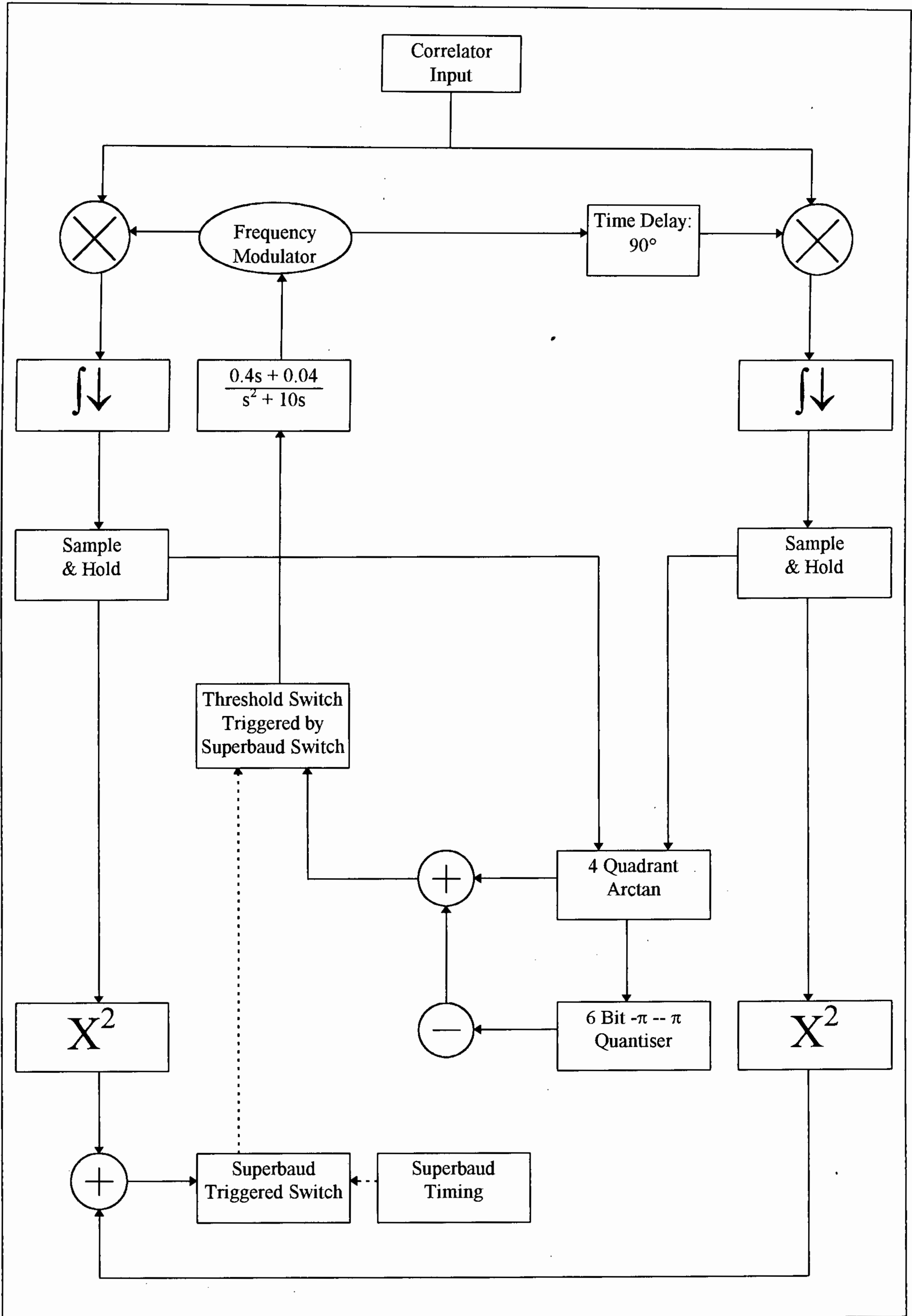


Fig 5.5: Block Diagram of a single correlator from Figure 5.2.

The filter preceding the frequency modulator in Figure 5.5 is of enormous importance. It must neither under- nor overreact to any given stimulus, else the Costas loop will be incorrectly tuned. The filter values presented in Figure 5.5 work for a near continuous input of a specific frequency. However, it does not work when the input is very spurious, as it must be for Multi-h. (This is also the reason for the use of a 'noiseless channel' in Figure 5.4. There is no sense in adding noise when the simulation does not work even in its absence.)

Unfortunately, the Systemview© platform on which the simulations were created necessitated much time and effort in 'fine tuning' the filter, and time constraints only allowed the testing of a few dozen filter configurations. It is our belief that the current filter works best of all those considered, although it is still inadequate to the task at hand. It must be noted that a Systemview© implementation of the Massey-Hodgart demodulator for MSK, as well as an implementation of the adapted demodulator for the demodulation of an arbritary 2 *h*-set worked exactly as it should, so we can conclude that the problem was indeed with the spurious nature of the 4 *h*-set's input. (The results from the above two working simulations are not fully documented as they do not satisfy the criterion of showing the synchronisation of a code with *long* constraint-length.)

It is unfortunate that, although the functionality of the system has been proved in theory, we were unable to make it work in practice. However, it is our belief that the system has enough merit to warrant implementation in a more flexible environment for further investigation.

5.4 Baud/Superbaud recovery

Since every bit period has its own different frequency associated with it, the obvious way to recover the baud rate is to generate a pulse every time a different frequency to the one currently received is detected. For the superbaud rate the same principle applies, except that you want a pulse only when you detect the frequencies from only one specific correlator.

The flow diagram in Figure 5.6 illustrates the baud/superbaud recovery process used for the *h*-set of size 4. Figures 5.7-12 shows the input to the receiver and output from the various numbered points on the flow diagram. Specifically, Figure 5.10 shows the recovered superbaud rate and Figure 5.12 the recovered baud rate. Although the two delays between the original and recovered clock rates apparent from these two graphs are different, they can be readily made equal. They merely reflect the different processing times necessary for the two clock recovery procedures.

The recovery process is shown in the absence of additive noise for purposes of clarity. Although reasonably robust to the effects of additive noise, the clock recovery system was in no way fortified against the effects of such noise. The main reason for this is that we preferred spending available time on improving the frequency synchroniser part of the adapted Massey-Hodgart demodulator, this having been a significantly more complex and important part of the problem at hand.

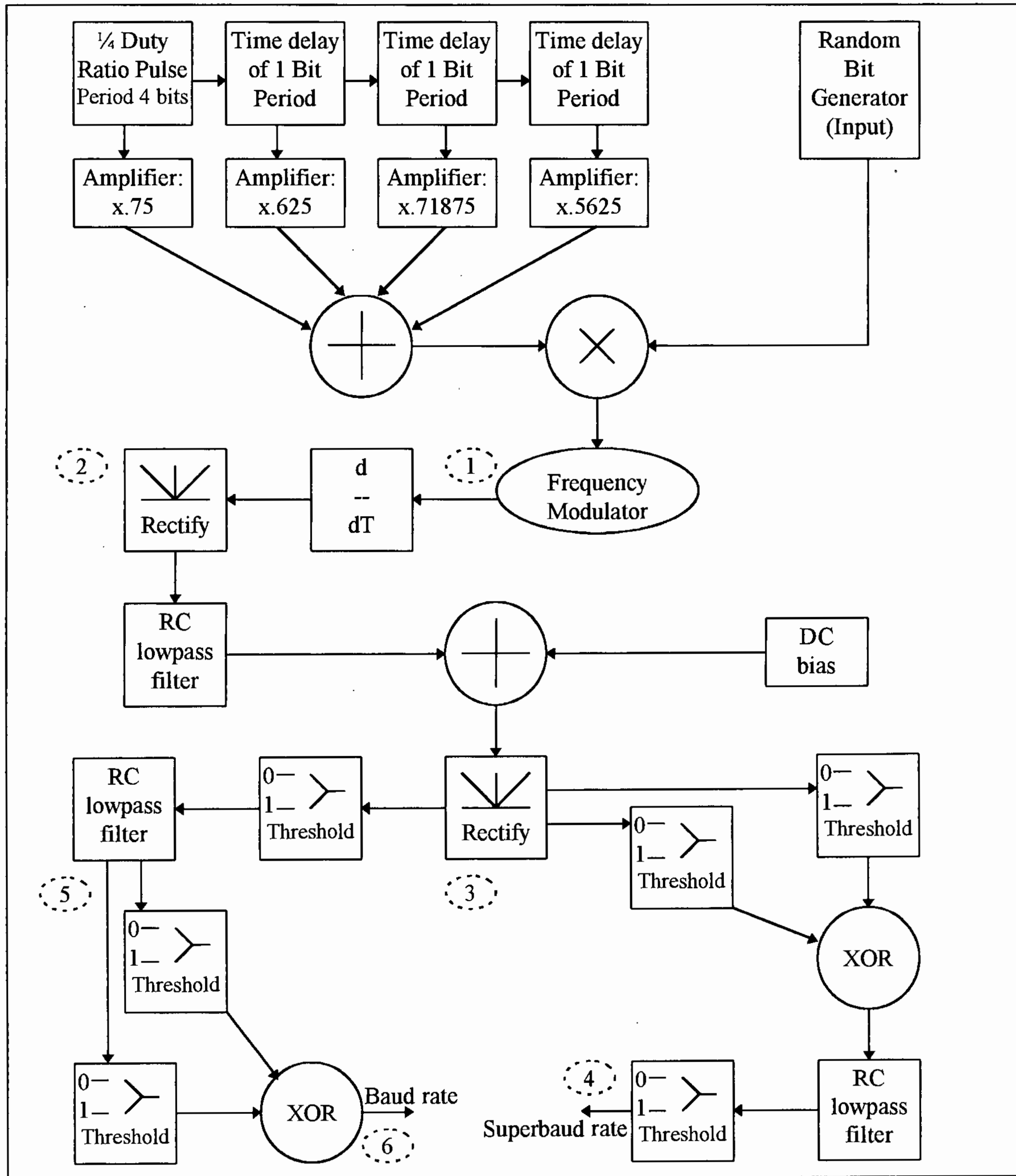


Fig 5.6: Baud and Superbaud Recovery

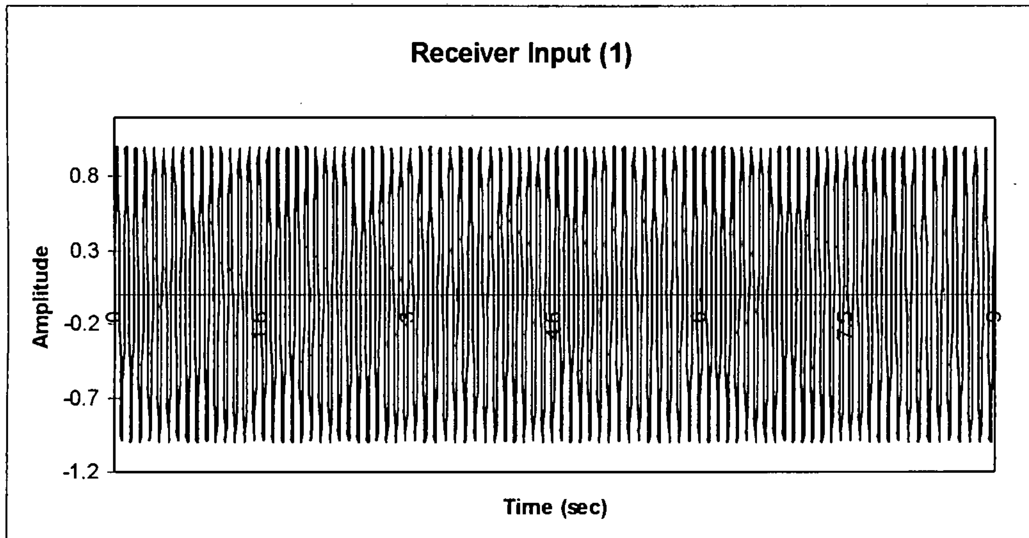


Fig 5.7: The Receiver Input (1)

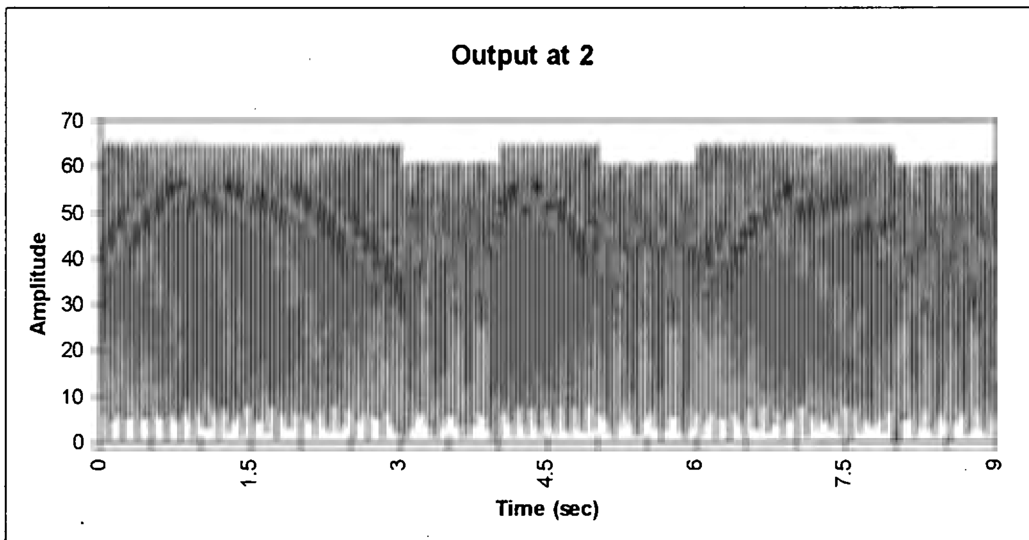
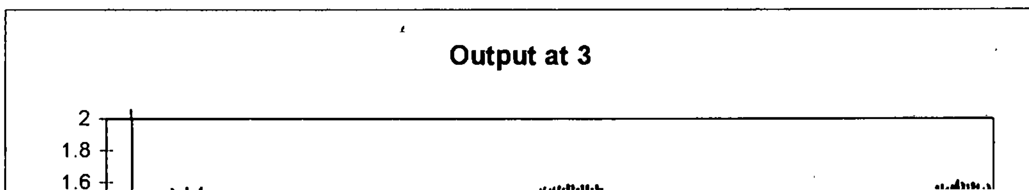


Fig 5.8: The Rectified Signal at (2)



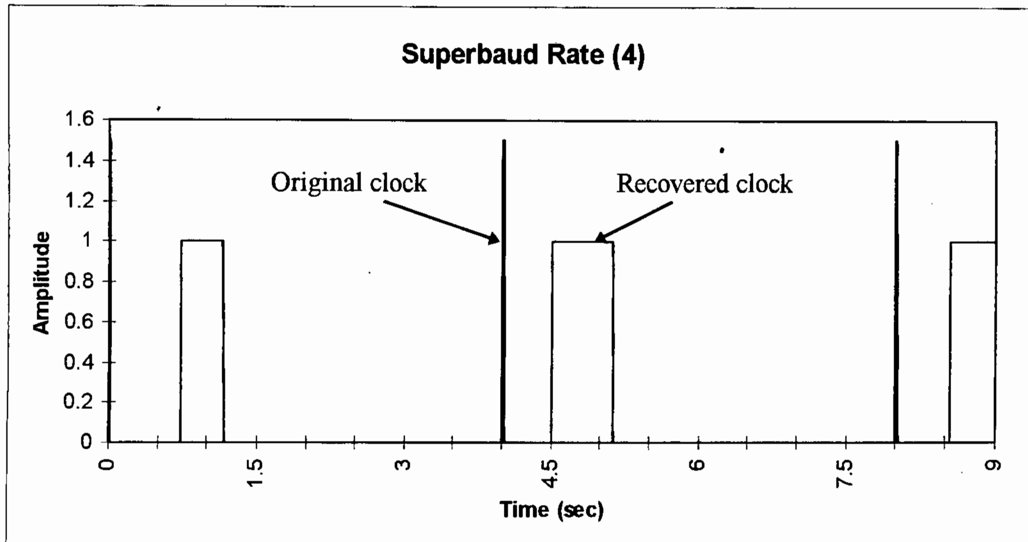


Fig 5.10: The Superbaud Rate (4)

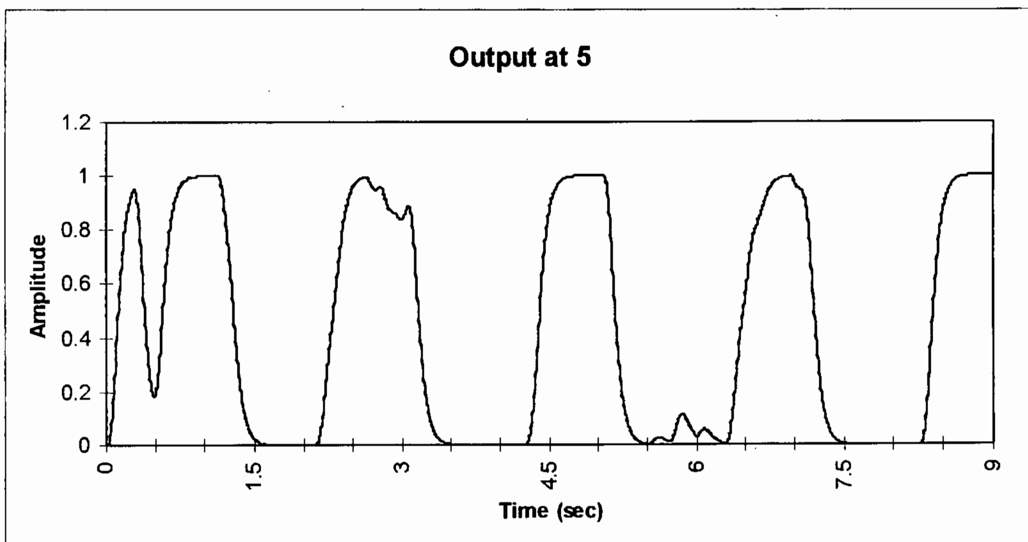


Fig 5.11: The Signal at (5) after Lowpass Filtering and in between Thresholding

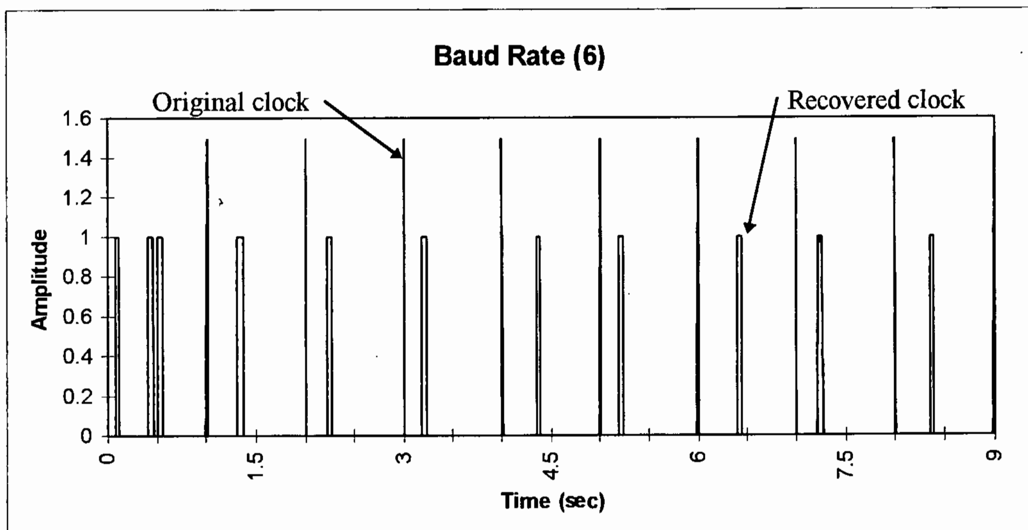


Fig 5.12: The Baud Rate (6)

5.5 Injected Carrier to facilitate synchronisation

Another alternative carrier recovery method was considered that would no doubt be easier to implement than the complicated Massey-Hodgart derivative. The argument here is that one can send the carrier itself without any modulation every x number of symbol periods in order to maintain carrier phase synchronisation.

Although we have little doubt that this method will work, it is very wasteful - whenever the carrier by itself is transmitted no data is transferred, resulting in power wastage. Also, if we transmit the phase-corrected carrier abruptly every x bits, we have phase discontinuities which results in a non-constant envelope - something we wish to avoid. If, on the other hand, we use a number of symbol periods to lead up to the carrier transmission with continuous phase to preserve the envelope, we waste even more power.

As a result, the process of carrier phase synchronisation by carrier injection was not investigated in detail in this work. The purpose was, after all, to find a novel way to synchronise Multi-*h* codes. Carrier injection does not fall into that category.

Chapter

6 Summary Chapter

The original purpose of this dissertation was to synchronise the very efficient Multi-h codes found recently by J. Cuthbert [1]. These codes have both a large denominator and long constraint-length, which meant that existing techniques for synchronising Multi-h signals, such as using a q-th power law device, would not work.

After some initial investigation it became clear that the codes in question were not as efficient as claimed. Since this project had no basis without efficient long constraint-length Multi-h codes, it was imperative to find such codes.

The method (Population-Based Incremental Learning) whereby the inefficient codes were found was revisited, and it was established that a calculation error resulted in the wrong codes being output - the method itself was not at fault. After a number of adjustments to the PBIL Search Algorithm a new set of codes was generated. The codes with lower constraint-lengths in this set had minimum squared Euclidean distances d^2_{min} close to theoretical limits calculated by [2], and it was hypothesized that the codes with longer constraint-lengths would also be close to their (as yet unknown) maxima.

Of all the codes found, the most promising one was that of constraint-length 5 and denominator 32. It was not the most efficient code from the set, but promised the best performance/implementation complexity tradeoff.

When we consider the synchronisation of Multi-h codes we see that we need three levels of synchronisation: we need to reestablish the baud rate, the superbaud rate and carrier synchronisation at the receiver. Both the baud and superbaud rates were extracted from the received signal by differentiating between the various frequencies in the received signal.

Maintaining carrier synchronisation proved to be a far harder task. The only way that one can accurately *reconstruct* the carrier with correct phase at the receiver is when one knows the output of the Viterbi decoder about 200 bit periods later. This is certainly a non-causal solution to the problem, and some possible alternative methods were examined.

The first method relies on the fact that, since we cannot reconstruct the carrier at the receiver, we may as well send it across the communications channel in unmodulated form every x bit periods to enable a phase-locked loop at the receiver to maintain synchronisation. However, this is wasteful of power, and especially so when one still attempts to conserve constant envelope while doing so. Since we wanted to find a *novel* way of synchronisation, this method was not investigated further.

The method that was investigated in a fair amount of depth is that of the adapted Massey-Hodgart MSK Demodulator. The Massey-Hodgart MSK Demodulator works on the principle of acquiring both the transmitted frequencies, instead of reestablishing the carrier at the transmitter that gave rise to those frequencies in the first place. This is a sensible approach, especially for a system such as Multi-h where it is impossible to tell what the carrier is just by examining the received signals.

However, for the promising *h*-set of size 4 the Massey-Hodgart derivative would require 8 different phase-locked loops for the 8 constituent frequencies. Since the various phase-locked loops are only trained when their specific frequency is received, it was feared that the potentially long time between updates for a specific phase-locked loop may give rise to errors.

These fears proved to be well-founded. The individual phase-locked loops works very well when receiving continuous input, but fail when they receive spurious input. It is unfortunate that the platform on which the system was implemented did not readily allow for fine tuning the operation of the complete system. Time constraints would also not allow porting the system to another platform.

It is believed that the Massey-Hodgart derivative has enough merit to warrant further investigation on a platform more conducive to fine tuning. The Massey-Hodgart Demodulator works well for MSK and we have no reason to believe that it could not also work for long constraint-length Multi-h given the correct environment.

Having said that, we should note that Multi-h codes with long constraint-length and large denominator (that is, all the efficient ones) have a debilitating drawback. These codes necessitate a very complex receiver structure.

The reality is that industry would much rather implement a sub-optimal code than a very efficient one that is complex. We fear that long constraint-length Multi-h codes may prove too complex to implement unless a far simpler receiver structure than the one proposed here can be developed.

Appendix

A PBIL Search Algorithm Code

```
/*****************************************************************************/
/*                                                                           */
/*  Modified PBIL search algorithm to find good symetrical h-sets. The       */
/*  fitness function for the search algorithm is the minimum squared         */
/*  euclidean distance of the h-set.                                         */
/*                                                                           */
/*  d^2(min) is calculated for each h-set by Viterbi decoding a modulated    */
/*  random bit sequence, and recording the minimum accumulated squared       */
/*  distance for all paths in the phase trellis which merge with the         */
/*  transmitted path.                                                        */
/*                                                                           */
/*****************************************************************************/


#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>



/*****************************************************************************/
/*  Data structure definition for struct NODE. This structure holds          */
/*  information for a node in the phase trellis diagram.                     */
/*****************************************************************************/


typedef struct node
{
  int  occupied;                        /*  Flag set if node is occupied     */
  double metric;                        /*  Path metric for node             */
} NODE;



/**************************** Seed shift register ***************************/
/*  Function to produce a random seed for the PN generator shift register.  */
/*                                                                           */
/*   int *sreg:  pointer to shift register                                   */
/*****************************************************************************/


void seedsreg (int *sreg)
{
  int counter;

  randomize();

  for (counter = 0;counter < 20;counter++)
    sreg[counter] = random(2);

  sreg[0]  = 1;
}
```

```
/****************************** Bit generator ******************************/
/*  Function to produce a random bit using the PN generator. Function     */
/*  returns the random bit and advances the shift register. The shift     */
/*  register is a 20 stage register with feedback taps 20 and 3. It       */
/*  produces a pseudonoise sequence of period 1048575.                    */
/*                                                                        */
/*   int *sreg:  pointer to shift register                                */
/**************************************************************************/

int bitgen (int *sreg)
{
  int bit;                              /*  Random bit                     */
  int temp;

  bit = sreg[19];

  temp = (sreg[19]+sreg[2]) % 2;

  sreg[19] = sreg[18];
  sreg[18] = sreg[17];
  sreg[17] = sreg[16];
  sreg[16] = sreg[15];
  sreg[15] = sreg[14];
  sreg[14] = sreg[13];
  sreg[13] = sreg[12];
  sreg[12] = sreg[11];
  sreg[11] = sreg[10];
  sreg[10] = sreg[9];
  sreg[9] = sreg[8];
  sreg[8] = sreg[7];
  sreg[7] = sreg[6];
  sreg[6] = sreg[5];
  sreg[5] = sreg[4];
  sreg[4] = sreg[3];
  sreg[3] = sreg[2];
  sreg[2] = sreg[1];
  sreg[1] = sreg[0];
  sreg[0] = temp;

  return (bit);
}


/***************************** Initialise node ****************************/
/*  Function to initialise the array of nodes. All node metrics are set   */
/*  to zero. All occupied flags are set to zero except for node zero.     */
/*                                                                        */
/*   NODE *node:  pointer to array of nodes                               */
/*   int hd:  denominator of h-set                                        */
/**************************************************************************/

void initnode (NODE *node,int hd)
{
  int counter;

  for (counter = 0;counter < 2*hd;counter++)
  {
    node[counter].occupied = 0;
    node[counter].metric = 0;
  }
  node[0].occupied = 1;
}
```

```
/******************************* Correlate ******************************/
/*  Function to calculate the correlations (over a single bit period)    */
/*  between the transmitted path and cos and sin waves for the two       */
/*  possible modulation frequencies.                                      */
/*                                                                        */
/*   int *h:  Pointer to h-set                                            */
/*   double hd:  Denominator of h-set                                     */
/*   int i:  h-set index                                                  */
/*   int bit:  Current transmitted bit                                    */
/*   int bitnode:  Starting node for current bit                          */
/*   double *corrcos0:  Pointer to correlation with '0' frequency cos     */
/*   double *corrsin0:  Pointer to correlation with '0' frequency sin     */
/*   double *corrcos1:  Pointer to correlation with '1' frequency cos     */
/*   double *corrsin1:  Pointer to correlation with '1' frequency sin     */
/*************************************************************************/


void correlate (int *h,double hd,int i,int bit,int bitnode,
             double *corrcos0,double *corrsin0,
             double *corrcos1,double *corrsin1)
{
  if (bit+1)
  {
    *corrcos0 = (sin((2*h[i] + bitnode)/hd*M_PI) -
              sin(bitnode/hd*M_PI)) / (2*h[i]/hd*M_PI);
    *corrsin0 = (sin((2*h[i] + bitnode+hd/2)/hd*M_PI) -
              sin((bitnode+hd/2)/hd*M_PI)) / (2*h[i]/hd*M_PI);
    *corrcos1 = cos(bitnode/hd*M_PI);
    *corrsin1 = cos((bitnode+hd/2)/hd*M_PI);
  }
  else
  {
    *corrcos0 = cos(bitnode/hd*M_PI);
    *corrsin0 = cos((bitnode+hd/2)/hd*M_PI);
    *corrcos1 = (sin((-2*h[i] + bitnode)/hd*M_PI) -
              sin(bitnode/hd*M_PI)) / (-2*h[i]/hd*M_PI);
    *corrsin1 = (sin((-2*h[i] + bitnode+hd/2)/hd*M_PI) -
              sin((bitnode+hd/2)/hd*M_PI)) / (-2*h[i]/hd*M_PI);
  }
}
```

```
/****************************** dmin generate ******************************/
/*  Function to calculate the minimum squared euclidean distance for a     */
/*  given h-set. Function returns the minimum squared euclidean distance.  */
/*  In the case of the minimum squared distance being less than the best   */
/*  minimum squared distance, the function aborts the calculation and sets */
/*  the crash flag.                                                        */
/*                                                                         */
/*   int *sreg:  Pointer to shift register                                 */
/*   int *h:  Pointer to h-set                                             */
/*   double hd:  Denominator of h-set                                      */
/*   int setsize:  Size of h-set                                           */
/*   double bestdmin:  Best minimum squared euclidean distance             */
/*   int *crash:  Pointer to crash indicator                               */
/***************************************************************************/


double dmingen (int *sreg,int *h,double hd,int setsize,double bestdmin,
             int *crash)
{
  NODE *node;                       /*  Pointer to an array of nodes   */
  NODE *temp;                       /*  Pointer to a temporary array   */
  NODE *swap;                       /*  Pointer used for swapping      */

  int snode0;                       /*  Starting node for a binary '0' */
  int snode1;                       /*  Starting node for a binary '1' */
  double corr0;                     /*  Path metric for '0' path       */
  double corr1;                     /*  Path metric for '1' path       */
  double corrcos0;                  /*  Correlation with '0' cos       */
  double corrsin0;                  /*  Correlation with '0' sin       */
  double corrcos1;                  /*  Correlation with '1' cos       */
  double corrsin1;                  /*  Correlation with '1' sin       */
  double dmin;                      /*  Minimum squared distance       */

  int bit;                          /*  Current transmitted bit        */
  int bitnode;                      /*  Starting node for current bit  */

  int i;                            /*  h-set index                    */
  int bitcount;                     /*  Bit counter                    */
  int nodecount;                    /*  Node counter                   */
  int numnodes;                     /*  Number of nodes in node array  */


  node = (NODE *)malloc(2*hd*sizeof(NODE));
  temp = (NODE *)malloc(2*hd*sizeof(NODE));

  initnode (node,hd);

  numnodes = 2*hd;
  bitnode = 0;
  dmin = 100;
  *crash = 0;

  for (bitcount = 0;bitcount < 1000 && dmin > bestdmin+0.001;
       bitcount++)
  {
    i = fmod (bitcount,setsize);
    bit = bitgen (sreg) * 2 - 1;

    correlate (h,hd,i,bit,bitnode,&corrcos0,&corrsin0,&corrcos1,&corrsin1);


/********/
```

```
    gotoxy (1,1);
    cprintf ("%d   %lf  ",bitcount,dmin);

/********/

    bitnode = fmod (bitnode + bit*h[i] + numnodes,numnodes);


    for (nodecount = 0;nodecount < numnodes;nodecount++)
    {
      snode0 = fmod(nodecount + h[i] + numnodes,numnodes);
      snode1 = fmod(nodecount - h[i] + numnodes,numnodes);

      corr0 = node[snode0].metric + cos(snode0/hd*M_PI)*corrcos0
                              - sin(snode0/hd*M_PI)*corrsin0;
      corr1 = node[snode1].metric + cos(snode1/hd*M_PI)*corrcos1
                              - sin(snode1/hd*M_PI)*corrsin1;


      if ((corr0 > corr1 || !node[snode1].occupied) &&
         node[snode0].occupied)
      {
       if (node[snode1].occupied && node[snode0].occupied)
       {
         if (nodecount == bitnode && corr0-corr1 < dmin)
           dmin = corr0-corr1;
       }
       temp[nodecount].occupied = 1;
       temp[nodecount].metric = corr0;
      }

      else if ((corr1 >= corr0 || !node[snode0].occupied) &&
              node[snode1].occupied)
      {
       if (node[snode0].occupied && node[snode1].occupied)
       {
         if (nodecount == bitnode && corr1-corr0 < dmin)
           dmin = corr1-corr0;
       }
       temp[nodecount].occupied = 1;
       temp[nodecount].metric = corr1;
      }

      else
      {
       temp[nodecount].occupied = 0;
       temp[nodecount].metric = 0;
      }
    }
    swap = node;
    node = temp;
    temp = swap;
  }

  if (bitcount < 1000)
    *crash = 1;

  free (node);
  free (temp);

  return (dmin);
}
```

```
/************************** Random number generate ************************/
/*  Function to generate a random number between 0 and 1, using a 10 bit   */
/*  sequence of random bits from the PN generator. Function returns the    */
/*  random number.                                                         */
/*                                                                         */
/*   int *sreg:  Pointer to shift register                                 */
/**************************************************************************/


float randnumgen (int *sreg)
{
  float randnum;                        /*  Random number between 0 and 1  */

  randnum = bitgen (sreg) * 512 +
            bitgen (sreg) * 256 +
            bitgen (sreg) * 128 +
            bitgen (sreg) * 64 +
            bitgen (sreg) * 32 +
            bitgen (sreg) * 16 +
            bitgen (sreg) * 8 +
            bitgen (sreg) * 4 +
            bitgen (sreg) * 2 +
            bitgen (sreg);

  randnum += 0.5;
  randnum /= 1024;

  return (randnum);
}


/*********************** Initialise probablity vector *********************/
/*  Function to initialise the PBIL probablity vector to 0.5.              */
/*                                                                         */
/*   float *pvector:  Pointer to PBIL probablity vector                    */
/*   int samplesize:  Size of PBIL sample                                  */
/**************************************************************************/


void initpvector (float *pvector,int samplesize)
{
  int counter;

  for (counter = 0;counter < samplesize;counter++)
  {
    pvector[counter] = 0.5;
  }
}
```

```
/************************* Learn probablity vector ************************/
/*  Function to implement learning of the PBIL probablity vector, using a  */
/*  PBIL sample from which to learn. Learning rate is 0.05 and this        */
/*  decreases asymptotically as the probability vector moves from 0.5      */
/*  towards 0 or 1.                                                        */
/*                                                                         */
/*   int *sample:  Pointer to PBIL sample                                  */
/*   int *pvector:  Pointer to PBIL probablity vector                      */
/*   int samplesize:  Size of PBIL sample                                  */
/**************************************************************************/


void learnpvector (int *sample,float *pvector,int samplesize)
{
  int counter;
  float adjust;                    /*  Adjustment to vector           */

  for (counter = 0;counter < samplesize;counter++)
  {
     adjust = (1-pvector[counter]-sample[counter]) * 0.1;
     if (adjust > 0.05)
       adjust = 0.05;
     if (adjust < -0.05)
       adjust = -0.05;
     pvector[counter] += adjust;
  }
}



/************************* Mutate probablity vector ***********************/
/*  Function to implement mutation of the PBIL probablity vector. Each     */
/*  bit of the probability vector is mutated by an amount 0.25 in a random */
/*  direction.                                                             */
/*                                                                         */
/*   int *sreg:  Pointer to shift register                                 */
/*   float *pvector:  Pointer to PBIL probability vector                   */
/*   int samplesize:  Size of PBIL sample                                  */
/**************************************************************************/


void mutatepvector (int *sreg,float *pvector,int samplesize)
{
  int counter;
  float randnum;

  for (counter = 0;counter < samplesize;counter++)
  {
    randnum = randnumgen (sreg);

    if (randnum > 0.5)
      pvector[counter] += 0.25;
    else
      pvector[counter] -= 0.25;
  }
}
```

```
/************************** Clip probablity vector ************************/
/*  Function to clip the PBIL probablity vector to the region (0,1).      */
/*                                                                        */
/*   float *pvector:  Pointer to PBIL probability vector                  */
/*   int samplesize:  Size of PBIL sample                                 */
/**************************************************************************/


void clippvector (float *pvector,int samplesize)
{
  int counter;

  for (counter = 0;counter < samplesize;counter++)
  {
    if (pvector[counter] > 1)
      pvector[counter] = 1;
    if (pvector[counter] < 0)
      pvector[counter] = 0;
  }
}



/***************************** Sample generate ***************************/
/*  Function to generate a sample bit sequence using the PBIL probablity  */
/*  vector.                                                               */
/*                                                                        */
/*   int *sreg:  Pointer to shift register                                */
/*   int *sample:  Pointer to PBIL sample                                 */
/*   float *pvector:  Pointer to PBIL probability vector                  */
/*   int samplesize:  Size of PBIL sample                                 */
/**************************************************************************/


void samplegen (int *sreg,int *sample,float *pvector,int samplesize)
{
  int counter;
  float randnum;

  for (counter = 0;counter < samplesize;counter++)
  {
    randnum = randnumgen (sreg);

    if (randnum > pvector[counter])
      sample[counter] = 1;
    else
      sample[counter] = 0;
  }
}
```

```
/****************************** h-set generate *****************************/
/*  Function to generate the h-set values from a PBIL sample bit sequence. */
/*                                                                          */
/*   int *h:  Pointer to h-set                                              */
/*   int *sample:  Pointer to PBIL sample                                   */
/*   int hd:  Denominator of h-set                                          */
/*   int setsize:  Size of h-set                                            */
/*   int samplesize:  Size of PBIL sample                                   */
/***************************************************************************/


void hgen (int *h,int *sample,int hd,int setsize,int samplesize)
{
  int multiplier;
  int counter;
  int temp;

  temp = samplesize/setsize;

  for (counter = 0;counter < samplesize;counter++)
  {
    if (!fmod(counter,temp))
    {
      h[counter/temp] = 0;
      multiplier = 1;
    }
    h[counter/temp] += multiplier*sample[counter];
    multiplier *= 2;
  }

  gotoxy (1,3);
  cprintf ("currh:   ");
  for (counter = 0;counter < setsize;counter++)
  {
    h[counter] = fmod(h[counter]+hd*7/8,hd) + 1;
    cprintf ("h%d = %d   ",counter,h[counter]);
  }
}



/****************************** Compare h-set ******************************/
/*  Function to compare two h-sets. Function returns a 1 if h-sets are     */
/*  the same, or a 0 if h-sets are different.                              */
/*                                                                          */
/*   int *h1:  Pointer to h-set 1                                           */
/*   int *h2:  Pointer to h-set 2                                           */
/*   int setsize:  Size of h-set                                            */
/***************************************************************************/


int compareh (int *h1,int *h2,int setsize)
{
  int counter;
  int same = 0;                     /*  Flag set if h-sets are equal   */

  for (counter = 0;counter < setsize;counter++)
  {
    if (h1[counter] == h2[counter])
      same += 1;
  }

  if (same == setsize)
```

```
    same = 1;
  else
    same = 0;

  return (same);
}


/******************************** Copy array ********************************/
/*  Function to copy an array of ints from source to destination.           */
/*                                                                           */
/*   int *dest:  Pointer to destination array                                */
/*   int *source:  Pointer to source array                                   */
/*   int arraysize:  Size of array                                           */
/*****************************************************************************/


void copyarray (int *dest,int *source,int arraysize)
{
  int counter;

  for (counter = 0;counter < arraysize;counter++)
     dest[counter] = source[counter];
}


/*********************************** Main **********************************/
/*  Main procedure                                                          */
/****************************************************************************/


void main()
{
  int sreg1[20];                       /*  PN generator 1 shift register  */
  int sreg2[20];                       /*  PN generator 2 shift register  */

  int setsize;                         /*  Size of h-set                  */
  int *h;                              /*  Pointer to h-set array         */
  int *besth;                          /*  Pointer to best h-set array    */
  int hd;                              /*  Denominator of h-set           */

  float *pvector;                      /*  Pointer to probability vector  */
  float *stdpvector;                   /*  Pointer to standard 0.5 vector */
  int *sample;                         /*  Pointer to PBIL sample         */
  int *bestsample;                     /*  Pointer to best PBIL sample    */
  double dmin;                         /*  Minimum squared distance       */
  double bestdmin;                     /*  Best minimum squared distance  */
  int crash;                           /*  Indicator for dmingen crash    */

  int samplesize;                      /*  Size of PBIL sample            */

  int nums;                            /*  No of samples per generation   */
  int numg;                            /*  Number of PBIL generations     */

  int gcount;                          /*  Generation counter             */
  int scount;                          /*  Sample counter                 */
  int samecount;                       /*  Counter h-sets same as best h  */
  int mutatecount;                     /*  Counter for mutates            */

  clrscr();
```

```
printf ("\n\n\n\n");
printf ("Enter size of h set:          ");
scanf ("%d",&setsize);
printf ("Enter denominator of h set:  ");
scanf ("%d",&hd);
printf ("\nEnter number of samples per PBIL generation:   ");
scanf ("%d",&nums);
printf ("\nEnter number of PBIL generations:              ");
scanf ("%d",&numg);


hd = log(hd+1)/log(2);
samplesize = hd*setsize;
hd = pow(2,hd);


printf ("\n\n");
printf ("\nsetsize    =  %d",setsize);
printf ("\nhd         =  %d",hd);
printf ("\nnums       =  %d",nums);
printf ("\nnumg       =  %d",numg);
printf ("\n\n\n");


getch();
clrscr();

pvector = (float *)malloc(samplesize*sizeof(float));
stdpvector = (float *)malloc(samplesize*sizeof(float));
sample = (int *)malloc(samplesize*sizeof(int));
bestsample = (int *)malloc(samplesize*sizeof(int));

h = (int *)malloc(setsize*sizeof(int));
besth = (int *)malloc(setsize*sizeof(int));


seedsreg (sreg1);
seedsreg (sreg2);
initpvector (pvector,samplesize);
initpvector (stdpvector,samplesize);

for (scount = 0;scount < setsize;scount++)
{
  besth[scount] = 0;
}

bestdmin = 0;
mutatecount = 0;

for (gcount = 0;gcount < numg;gcount++)
{
  samecount = 0;
  for (scount = 0;scount < nums;scount++)
  {
    gotoxy (1,18);
    cprintf ("oddballs:  %d  ",samecount);
    samplegen (sreg2,sample,pvector,samplesize);
    hgen (h,sample,hd,setsize,samplesize);
    if (compareh(h,besth,setsize))
    {
     samplegen (sreg2,sample,stdpvector,samplesize);
     hgen (h,sample,hd,setsize,samplesize);
     samecount++;
```

```
      }
      dmin = dmingen (sreg1,h,hd,setsize,bestdmin,&crash);
      if (!crash)
      {
       bestdmin = dmin;
       gotoxy (55,8);
       cprintf ("last update:  %d ",gcount);
       copyarray (bestsample,sample,samplesize);
       copyarray (besth,h,setsize);
      }
      gotoxy (1,8);
      cprintf ("generation:  %d       best dmin  =  %lf",
             gcount,bestdmin);
    }
    learnpvector (bestsample,pvector,samplesize);
    if (samecount > nums/10-1)
    {
      mutatepvector (sreg2,pvector,samplesize);
      mutatecount++;
      gotoxy (1,20);
      cprintf ("mutates:  %d",mutatecount);
    }
    clippvector (pvector,samplesize);
    gotoxy (1,10);
    cprintf ("besth:   ");
    for (scount = 0;scount < setsize;scount++)
    {
      cprintf ("h%d = %d   ",scount,besth[scount]);
    }
  }

  getch();

  free (pvector);
  free (stdpvector);
  free (sample);
  free (bestsample);
  free (h);
  free (besth);
}
```

Appendix

B Viterbi Decoder Code

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <math.h>
#include <time.h>


#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)


int FREQ;
int SAMPLES;
int TRUNCDEPTH;

float N0;


/*************************************************************************/
/*  Data structure definition for struct NODE. This structure holds      */
/*  information for a node in the phase trellis diagram.                 */
/*************************************************************************/


typedef struct node
{
  int  occupied;                      /*  Flag set if node is occupied   */
  double metric;                      /*  Path metric for node           */
  char path[500];                     /*  Path bit sequence              */
} NODE;
```

```
/******************************** Random 1 ********************************/
/*  Function to produce a random number between 0.0 and 1.0. Function     */
/*  returns the random number. Initialises if called with idum a negative */
/*  number.                                                               */
/*                                                                        */
/*   long *sreg:  pointer to seed                                         */
/**************************************************************************/


float ran1 (long *idum)
{
  int j;
  int k;
  static long iy=0;
  static long iv[NTAB];
  float temp;

  if (*idum <= 0 || !iy)
  {
    if (-(*idum) < 1)
      *idum = 1;
    else
      *idum = -(*idum);
    for (j = NTAB+7;j >= 0;j--)
    {
      k = (*idum)/IQ;
      *idum = IA*(*idum-k*IQ)-IR*k;
      if (*idum < 0)
       *idum += IM;
      if (j < NTAB)
       iv[j] = *idum;
    }
    iy = iv[0];
  }
  k = (*idum)/IQ;
  *idum = IA*(*idum-k*IQ)-IR*k;
  if (*idum < 0)
    *idum += IM;
  j = iy/NDIV;
  iy = iv[j];
  iv[j] = *idum;
  if ((temp=AM*iy) > RNMX)
    return RNMX;
  else
    return temp;
}
```

```
/**************************** Gaussian deviate ****************************/
/*  Function to produce a normally distributed random number with zero    */
/*  mean and unit variance. Function returns the random number.            */
/*                                                                         */
/*  Source: Numerical Recipes in C - Press, Teukolsky, Vetterling,        */
/*          Flannery                                                       */
/*                                                                         */
/*   long *sreg:  pointer to seed                                          */
/***************************************************************************/


float gasdev (long *idum)
{
  float ran1 (long *idum);
  static int iset = 0;
  static float gset;
  float fac, rsq, v1, v2;

  if (iset == 0)
  {
    do
    {
      v1 = 2.0*ran1(idum)-1.0;
      v2 = 2.0*ran1(idum)-1.0;
      rsq = v1*v1+v2*v2;
    }
    while (rsq >= 1.0 || rsq == 0.0 );

    fac = sqrt (-2.0*log(rsq)/rsq);
    gset = v1*fac;
    iset = 1;
    return v2*fac;
  }
  else
  {
    iset=0;
    return gset;
  }
}


/*************************** Seed shift register ***************************/
/*  Function to produce a random seed for the PN generator shift register. */
/*                                                                         */
/*   int *sreg:  pointer to shift register                                 */
/***************************************************************************/


void seedsreg (int *sreg)
{
  int counter;

  randomize();

  for (counter = 0;counter < 20;counter++)
    sreg[counter] = random(2);

  sreg[0]   = 1;
}
```

```
/****************************** Bit generator *****************************/
/*  Function to produce a random bit using the PN generator. Function      */
/*  returns the random bit and advances the shift register. The shift      */
/*  register is a 20 stage register with feedback taps 20 and 3. It       */
/*  produces a pseudonoise sequence of period 1048575.                     */
/*                                                                         */
/*   int *sreg:  pointer to shift register                                 */
/***************************************************************************/


int bitgen (int *sreg)
{
  int bit;                              /*  Random bit                     */
  int temp;

  bit = sreg[19];

  temp = (sreg[19]+sreg[2]) % 2;

  sreg[19] = sreg[18];
  sreg[18] = sreg[17];
  sreg[17] = sreg[16];
  sreg[16] = sreg[15];
  sreg[15] = sreg[14];
  sreg[14] = sreg[13];
  sreg[13] = sreg[12];
  sreg[12] = sreg[11];
  sreg[11] = sreg[10];
  sreg[10] = sreg[9];
  sreg[9] = sreg[8];
  sreg[8] = sreg[7];
  sreg[7] = sreg[6];
  sreg[6] = sreg[5];
  sreg[5] = sreg[4];
  sreg[4] = sreg[3];
  sreg[3] = sreg[2];
  sreg[2] = sreg[1];
  sreg[1] = sreg[0];
  sreg[0] = temp;

  return (bit);
}



/***************************** Initialise node ****************************/
/*  Function to initialise the array of nodes. All node metrics are set    */
/*  to zero. All occupied flags are set to zero except for node zero.      */
/*                                                                         */
/*   NODE *node:  pointer to array of nodes                                */
/*   int hd:  denominator of h-set                                         */
/***************************************************************************/


void initnode (NODE *node,int hd)
{
  int counter1;
  int counter2;

  for (counter1 = 0;counter1 < 2*hd;counter1++)
  {
    node[counter1].occupied = 0;
    node[counter1].metric = 0;
```

```
    for (counter2 = 0;counter2 < TRUNCDEPTH;counter2++)
      node[counter1].path[counter2] = 0;
  }
  node[0].occupied = 1;
}



/******************************** Correlate ********************************/
/*  Function to calculate the correlations (over a single bit period)      */
/*  between the transmitted path and cos and sin waves for the two         */
/*  possible modulation frequencies.                                       */
/*                                                                         */
/*   int *h:  Pointer to h-set                                             */
/*   double hd:  Denominator of h-set                                      */
/*   int i:  h-set index                                                   */
/*   int bit:  Current transmitted bit                                     */
/*   int bitnode:  Starting node for current bit                           */
/*   long *idum:  Seed for Gaussian noise generator                        */
/*   double *corrcos0:  Pointer to correlation with '0' frequency cos      */
/*   double *corrsin0:  Pointer to correlation with '0' frequency sin      */
/*   double *corrcos1:  Pointer to correlation with '1' frequency cos      */
/*   double *corrsin1:  Pointer to correlation with '1' frequency sin      */
/***************************************************************************/


void correlate (int *h,double hd,int i,int bit,int bitnode,long *idum,
             double *corrcos0,double *corrsin0,
             double *corrcos1,double *corrsin1)
{
  double wave;
  double phase;
  float noiseampl;

  int counter;

  *corrcos0 = 0;
  *corrsin0 = 0;
  *corrcos1 = 0;
  *corrsin1 = 0;

  for (counter = 0;counter < SAMPLES;counter++)
  {
    phase = (FREQ*hd*2 + bit*h[i]) * (double)counter/SAMPLES;
    noiseampl = sqrt(N0*SAMPLES/2.0);
    wave = cos((phase+bitnode)/hd*M_PI)*sqrt(2) + gasdev(idum)*noiseampl;

    phase = (FREQ*hd*2 - h[i]) * (double)counter/SAMPLES;
    *corrcos0 += wave * cos(phase/hd*M_PI);
    *corrsin0 += wave * sin(phase/hd*M_PI);

    phase = (FREQ*hd*2 + h[i]) * (double)counter/SAMPLES;
    *corrcos1 += wave * cos(phase/hd*M_PI);
    *corrsin1 += wave * sin(phase/hd*M_PI);
  }
  *corrcos0 /= SAMPLES;
  *corrsin0 /= SAMPLES;
  *corrcos1 /= SAMPLES;
  *corrsin1 /= SAMPLES;
}
```

```
/******************************** Copy path ********************************/
/*  Function to copy a node path from source to destination.                */
/*                                                                          */
/*   int *dest:  Pointer to destination node                                */
/*   int *source:  Pointer to source node                                   */
/***************************************************************************/


void copypath (char *temp,char *path)
{
  int counter;

  for (counter = 0;counter < TRUNCDEPTH;counter++)
     temp[counter] = path[counter];
}



/*********************************** Main **********************************/
/*  Main procedure                                                          */
/***************************************************************************/


void main()
{
  long idum;                           /*  Gaussian number generator seed */
  int sreg[20];                        /*  PN generator shift register    */

  int setsize;                         /*  Size of h-set                  */
  int *h;                              /*  Pointer to h-set array         */
  int hd;                              /*  Denominator of h-set           */

  NODE *node;                          /*  Pointer to an array of nodes   */
  NODE *temp;                          /*  Pointer to a temporary array   */
  NODE *swap;                          /*  Pointer used for swapping      */

  int snode0;                          /*  Starting node for a binary '0' */
  int snode1;                          /*  Starting node for a binary '1' */
  double corr0;                        /*  Path metric for '0' path       */
  double corr1;                        /*  Path metric for '1' path       */
  double corrcos0;                     /*  Correlation with '0' cos       */
  double corrsin0;                     /*  Correlation with '0' sin       */

  double corrcos1;                     /*  Correlation with '1' cos       */
  double corrsin1;                     /*  Correlation with '1' sin       */
  double biggest;                      /*  Biggest path metric            */
  int outnode;                         /*  Node with biggest path metric  */

  int bit;                             /*  Current transmitted bit        */
  int bitnode;                         /*  Starting node for current bit  */
  char sentpath[500];                  /*  Transmitted path               */

  int i;                               /*  h-set index                    */
  int bitcount;                        /*  Bit counter                    */
  int nodecount;                       /*  Node counter                   */
  int numbits;                         /*  Number of bits to decode       */
  int numnodes;                        /*  Number of nodes in node array  */
  int errorcount;                      /*  Number of errors               */


  FILE *fpo;
```

```
  if (!(fpo = fopen ("viterbi.txt","w")))
  {
    fprintf (stderr,"ERROR: couldn't open file viterbi.txt for writing\n");
    exit (1);
  }


/*************/

  clrscr();

  printf ("\n\n\n\n");
  printf ("Enter number of bits to decode:      ");
  scanf ("%d",&numbits);
  printf ("Enter modulation center frequency:  ");
  scanf ("%d",&FREQ);
  printf ("Enter number of samples per bit:     ");
  scanf ("%d",&SAMPLES);
  printf ("Enter viterbi truncation depth:      ");
  scanf ("%d",&TRUNCDEPTH);
  printf ("Enter noise PSD, N0:    ");
  scanf ("%f",&N0);

  printf ("\n");

  printf ("Enter h-set denominator:    ");
  scanf ("%d",&hd);
  printf ("Enter h-set size:           ");
  scanf ("%d",&setsize);


  printf ("\n");

  printf ("\nnumbits = %d",numbits);
  printf ("\nFREQ = %d",FREQ);
  printf ("\nSAMPLES = %d",SAMPLES);
  printf ("\nTRUNCDEPTH = %d",TRUNCDEPTH);
  printf ("\nN0 = %f",N0);

  printf ("\n");

  printf ("\nhd = %d",hd);
  printf ("\nsetsize = %d",setsize);


  h = (int *)malloc(setsize*sizeof(int));


  printf ("\n\n\n");

  for (i = 0;i < setsize;i++)
  {
    printf ("Enter h[%d]:    ",i);
    scanf ("%d",&h[i]);
  }

  printf ("\n");

  for (i = 0;i < setsize;i++)
  {
    printf ("\nh[%d] = %d",i,h[i]);
  }
```

```
  getch();

/*************/


  node = (NODE *)malloc(2*hd*sizeof(NODE));
  temp = (NODE *)malloc(2*hd*sizeof(NODE));


  printf ("\n\n\n");

  randomize();
  idum = random(10) - 10;

  seedsreg (sreg);
  initnode (node,hd);


  numnodes = 2*hd;
  bitnode = 0;
  errorcount = 0;


  for (bitcount = 0;bitcount < numbits;bitcount++)
  {
    i = fmod (bitcount,setsize);
    bit = bitgen (sreg) * 2 - 1;

    sentpath[fmod(bitcount,TRUNCDEPTH)] = (bit+1)/2;

    correlate (h,hd,i,bit,bitnode,&idum,&corrcos0,&corrsin0,
               &corrcos1,&corrsin1);


    bitnode = fmod (bitnode + bit*h[i] + numnodes,numnodes);

    for (nodecount = 0;nodecount < numnodes;nodecount++)
    {
      snode0 = fmod(nodecount + h[i] + numnodes,numnodes);
      snode1 = fmod(nodecount - h[i] + numnodes,numnodes);

      corr0 = node[snode0].metric + cos(snode0/(double)hd*M_PI)*corrcos0
                                  - sin(snode0/(double)hd*M_PI)*corrsin0;
      corr1 = node[snode1].metric + cos(snode1/(double)hd*M_PI)*corrcos1
                                  - sin(snode1/(double)hd*M_PI)*corrsin1;


      if ((corr0 > corr1 || !node[snode1].occupied) &&
         node[snode0].occupied)
      {
       temp[nodecount].occupied = 1;
       temp[nodecount].metric = corr0;
       copypath (temp[nodecount].path,node[snode0].path);
       temp[nodecount].path[fmod(bitcount,TRUNCDEPTH)] = 0;
      }
      else if ((corr1 >= corr0 || !node[snode0].occupied) &&
               node[snode1].occupied)
      {
       temp[nodecount].occupied = 1;
       temp[nodecount].metric = corr1;
```

```
        copypath (temp[nodecount].path,node[snode1].path);
        temp[nodecount].path[fmod(bitcount,TRUNCDEPTH)] = 1;
       }
       else
       {
        temp[nodecount].occupied = 0;
        temp[nodecount].metric = 0;
       }
     }
     swap = node;
     node = temp;
     temp = swap;



     cprintf ("\n\r%d",bitcount);

     if (bitcount+1 > TRUNCDEPTH-1)
     {
       biggest = NULL;
       outnode = 0;
       for (nodecount = 0;nodecount < 2*hd;nodecount++)
       {
        if (node[nodecount].occupied)
        {
          if (biggest == NULL || node[nodecount].metric > biggest)
          {
            biggest = node[nodecount].metric;
            outnode = nodecount;
          }
        }
       }
       cprintf ("    %d",node[outnode].path[fmod(bitcount+1,TRUNCDEPTH)]);

       if (node[outnode].path[fmod(bitcount+1,TRUNCDEPTH)] !=
          sentpath[fmod(bitcount+1,TRUNCDEPTH)])
       {
        errorcount++;
       }
       cprintf ("     errors: %d",errorcount);
     }
   }


   printf ("\n\n");
   printf ("\nNumber of bits:   %d",numbits-TRUNCDEPTH+1);
   printf ("\nNumber of errors: %d",errorcount);
   printf ("\n\nBER  =  %f\n",(float)errorcount/(numbits-TRUNCDEPTH+1));

   getch();


   free (node);
   free (temp);
   free (h);

   fclose (fpo);
}
```

Page

64 Bibliography

[1] **Cuthbert, J.** *High Performance Multi-h CPFSK Modulator and Demodulator Design using Population-Based Incremental Learning Search Methods.* PhD thesis, Department of Electrical Engineering, University of Cape Town, December 1996.

[2] **Anderson, JB. Aulin, T. and Sundberg, C.** *Digital Phase Modulation.* Chapter 3, Plenum Press, 1986

[3] **Böhm, B.** *DSP Implementation and Performance Testing of the Massey-Hodgart MSK Demodulator.* MSc thesis, Department of Electrical Engineering, University of Cape Town, June 1995.

[4] **Ziemer, RE. and Peterson, RL.** *Introduction to Digital Communication.* Macmillan Publishing Company, 1992. (General Reference)

[4.1] Chapter 4 of the above

[4.2] Chapter 3 of the above

[5] **Sundberg, C.** *Continuous Phase Modulation.* IEEE Communications Magazine, Vol. 24, pp. 25-38, April 1986.

[6] **Viterbi, A.J.** *Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm.* IEEE Transactions on Information Theory, Vol. IT-13, pp. 260-269, April 1967.

[7] **Ziemer, RE. and Peterson, RL.** *Digital Communications and Spread Spectrum Systems.* Macmillan Publishing Company, 1985.

[8] **Baluja, S.** *Population-based Incremental Learning: A Method for Integrating Genetic Search-based Function Optimization.* Technical report, School of Computer Science, Carnegie Mellon University, June 1994.

[9] **Haykin, S.** *Digital Communications.* pp. 102-103, John Wiley & Sons, 1988.

[10] **Hodgart, M.S. and Schoonees, J.A.** *A Robust MSK Demodulator through DSP,* 1992 IEEE South African Symposium on Communications and Signal Processing, Cape Town.