Decoding Algorithms for Continuous Phase Modulation

Werner Frederick Kleyn

4th June 2002

Acknowledgements

I would like to thank Professor Robin Braun for his supervision of this research as well as the Department of Electrical Engineering at the University of Cape Town. Also thank you to Dr Daniel Mashao who acted as internal supervisor during the latter stage of this research.

Tellumat Pty (Ltd) provided financial assistance for which I am very grateful.

There were times when I had to neglect my family and friends in order to complete this research. Thank you for your patience and encouragement.

Finally, I wish to thank the LORD for His grace and the encouragement I received from Him without I would not have been able to complete this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Continuous Phase Modulation (CPM) possesses characteristics that make it very attractive for many applications. Efficient non-linear power amplifiers can be used in the transmitters of constant envelope CPM schemes. CPM also allows for the use of simple limiters in the demodulator rather than linear receivers with gain control. These characteristics not only increases the life of the power source, but it improves circuit reliability since less heat is generated. In some applications, such as satellite transmitters, where power and circuit failure is very expensive, CPM is the most attractive choice.

Bandwidth efficiency, also, is very attractive, and improves as the order of the scheme increases (together with reduction in modulation index). Still further improvement is obtained through pulse-shaping which normally result in partial response schemes as opposed to full-response (CPFSK) schemes.

The inherent memory or coding gain of CPM increases the minimum distance, which is a figure of merit for a scheme's error performance. The length of the inherent memory is the *constraint length* of the scheme. Successful extraction of this inherent memory result in improved power efficiency. By periodic variation of the modulation index as in multi-h CPFSK, a sub class of CPM, coding gain or inherent memory can be significantly improved.

CPM demodulation is also less sensitive to fading channels than some other comparable systems. Well-known schemes such as GSM digital mobile systems, DECT and Iridium all use some form of CPM to transport their information. These implementations are normally pulse-shaped FSK or MSK and are used for the reasons above, except that their receivers do not always exploit the inherent memory.

Unfortunately, though, when one wants to exploit the inherent memory of higher level CPM schemes, all these attractive characteristics are offset by the complexity of the receiver structures which increases exponentially in complexity as the order or constraint length is increased.

Optimum receivers for binary CPFSK were first described by Osborne and Luntz [19] in 1974 and their research was later extended by Schonhoff [26] to include M-*ary* CPFSK. These receivers evaluate likelihood functions after observing the received signal for a certain number of symbol intervals, say N, then calculate a set of likelihood parameters on which a likelihood ratio test regarding the first symbol is based. These receivers are complex and impractical but does provide valuable insight. This is called maximum likelihood sequence estimation (MLSE).

Another way to do MLSE would be to correlate all possible transmitted sequences (reference signals at the demodulator) over a period of N symbol intervals with the received sequence. The

first symbol of the reference sequence with which the received sequence has the largest correlation, is decoded as the most likely symbol. The number of reference sequences required at the receiver grow very fast as the observation period increases. Up to now, only the lowest order CPM schemes have feasible optimal receiver structures.

The only practical solution thus far for the MLSE of higher order schemes is the use of software implementations of which the Viterbi algorithm is the most popular. Through recursive or sequential processing of data per interval, the number of matched filters required can be reduced. However, for schemes beyond a certain order and constraint length, the Viterbi algorithm's consumption of computational resources reduces it's feasibility.

Research into CPM is focused mainly on the quest for simpler demodulators and decoders or lower order schemes with better coding gain. In order to gain further insight into CPM, research is approached from different angles.

Most significantly, in 1980 Massey [16] proposed a simple and practical MSK maximum likelihood demodulator. His starting point was to separate the continuous-phase coding/decoding and memoryless modulation/demodulation functions of the MSK receiver/transmitter and later [17] recommended that this principle should be extended to CPM schemes in general. Rimoldi [21] was the first to present a generalized method of separating the continuous-phase coding and memoryless modulation functions of a CPM modulator. This decomposition approach has two important implications. The memoryless modulator may be cascaded with the channel and a memoryless demodulator (a demodulator operating over one symbol interval) to form a discrete memoryless channel. This channel can then be studied with well-established theory. The second implication is that the continuous-phase encoder can be studied separately as a time-invariant linear sequential circuit, using the same theory as that developed for convolutional encoders. This encoder can be combined with an outside convolutional encoder to increase the minimum distance of the overall scheme, while ensuring that the continuous-phase requirement of the transmitted waveform is met.

Schoonees [27] did a likelihood ratio analysis of CPFSK. In his research he analyzed Massey's MSK receiver, based on the maximum likelihood receiver of Schonhoff [26], and discovered that the likelihood ratio test of MSK simplify to the same decoder as proposed by Massey [16]. He then extended this analysis to higher order CPFSK schemes and found that a similar simplification does not exist for schemes other than MSK. The conclusion is that no simplified optimum receivers exist for higher order CPFSK schemes.

Various reduced complexity decoders have been suggested. Reduced State Trellis decoding [8, 12, 30] is a method that tries to apply the Viterbi algorithm on the most promising part of the demodulator output.

Recently, some research has been done on the feasibility of using neural networks as decoders [18, 32] for CPM. Poor results were obtained as the error performance seems to be very sensitive to noise. The performance of these kinds of decoders are suboptimal, the architecture complex, but it is believed that a neural net-based implementation will become practical as VLSI digital and/or analogue neural network chips become feasible components.

1.2 Problem definition and Motivation

Power and bandwidth efficient CPM modulation schemes are well-defined, but feasible optimum decoders[1] for these schemes does not exist [27], except for MSK.

The complexity of software decoders have become more manageable in recent years with the advent of powerful digital signal processing integrated circuits. However, system cost is still largely

[1]The implementation that process the information received from the "memoryless" demodulator.

determined by complexity. The quest will always be for simplicity, which will lead to better power efficiency, cost effectiveness and ease of implementation. So any research that lead to more insight into CPM decoding, and performance results for possible simpler decoders, will be of value.

The use of neural networks implementations in decoding CPM has not been fully researched, especially its behavior in the presence of noise. The Fano algorithm is a sequential decoding algorithm of which the performance, when applied to CPM, is relatively unknown. Rimoldi [21] showed that MSK can be decoded optimally with only one symbol delay. Applying his method shows that 4-*ary* CPFSK need more than one symbol delay to be decoded optimally. This is contradictory to intuition when studying 4-*ary* CPFSK state trellis and will need investigation before comparing the results of any new algorithm with the optimum one. In summary then:

1. Neural network decoders seems to be very sensitive to Gaussian noise, leading to poor error performance. Investigate this sensitivity and compare Neural Net-based decoder with an optimum decoder.

2. Investigate the performance of the Fano algorithm as a decoder of CPM.

3. Investigate the optimum observation periods for CPFSK.

1.3 Objectives and Method

Most of this research is based on 4-*ary* CPFSK and MSK with the emphasis on effective software decoding algorithms for 4-*ary* CPFSK.

Since error performance is an important figure of merit, we will establish a reliable error bound for 4-*ary* CPFSK against which we can compare other decoders. The first objective will be to conduct a search for optimum Fano algorithm parameters, that is, parameters that will minimize error events and catastrophic failure.

Up to now, research on neural network trellis decoding was applied to higher level CPM schemes. We will compare the performance of Viterbi-decoded 4-*ary* CPFSK and MSK with those of a Neural Net-based ones.

In this research I will assume white Gaussian noise as the only channel perturbation, that is, no fading and band-limiting etc. Simulations will be performed using a custom written C program.

Chapter 2

System description

2.1 Introduction

In this chapter we establish a baseline against which we will interpret the results of Chapters 3 and 4. The continuous phase modulated signal is described, as well as maximum likelihood sequence estimation and the implementation of the Viterbi algorithm. Through simulation and mathematical analysis we establish the bit error probability and the optimum observation window for decoding 4-*ary* CPFSK.

2.2 Traditional Description of the Continuous Phase Modulated (CPM) signal

The CPM signal is described by [2, 3]

$$s(t, \alpha) = \sqrt{\frac{2E_s}{T_s}} \cos\left[2\pi f_0 t + \varphi(t, \alpha) + \varphi_0\right], \qquad t \geq 0 \tag{2.1}$$

where the information-carrying phase is

$$\varphi(t, \alpha) = 2\pi h \sum_{i=0}^{\infty} \alpha_i f(t - iT_s), \qquad t \geq 0 \tag{2.2}$$

The parameter T_s is the channel-symbol time, E_s the energy per symbol, and h is the modulation index. The modulation index h determines the rate of change of the phase and is assumed to be a rational number

$$h = \frac{K}{P} \tag{2.3}$$

where K and P are relative prime positive integers. This constraint gives the CPM system a periodic (modulo-2π) phase structure, that is, a finite number of phase states, otherwise the optimum receiver has infinite complexity.

$$f(t) = \int_0^t g(\tau) d(\tau) \tag{2.4}$$

is the phase response function with $g(t)$ a frequency shape function defined over a finite time interval $0 \leq t \leq LT_s$ and zero outside.

L is the length of the pulse, and is equal to the number of input symbols which affect the

change in phase over the current symbol interval and is also sometimes called the memory of the scheme. $g(t)$ is defined to limit $f(t)$ as follows:

$$f(t) = \begin{cases} 0, & t < 0 \\ \frac{1}{2}, & t \geq LT_s \end{cases} \tag{2.5}$$

Systems with $L = 1$ are called full-response schemes; whereas partial-response schemes have $L > 1$.

The M-*ary* information sequence is $\boldsymbol{\alpha} = (\alpha_0, \alpha_1, \alpha_2, \alpha_3, \ldots)$ where

$$\alpha_i \in \begin{cases} \{\pm 1, \pm 3, \ldots, \pm(M-1)\} & M \text{ even}, i \geq 0 \\ \{0, \pm 2, \pm 4, \ldots, \pm(M-1)\} & M \text{ odd}, i \geq 0 \end{cases}$$

2.3 Tilted Phase Description of the CPM signal

The continuous phase modulated signal waveform can also be described [21] by

$$s(t, \mathbf{u}) = \sqrt{\frac{2E_s}{T_s}} \cos\left[2\pi f_1 t + \psi(t, \mathbf{u}) + \varphi_0\right], \qquad t \geq 0 \tag{2.6}$$

where $\psi(t, \mathbf{u})$ is the information-carrying phase defined as

$$\psi(t, \mathbf{u}) = 4\pi h \sum_{i=0}^{\infty} u_i f(t - iT_s), \qquad t \geq 0 \tag{2.7}$$

and f_1 is an asymmetrical carrier frequency. The parameter $\mathbf{u}$ signifies a baseband data symbol sequence $\mathbf{u} = (u_0, u_1, u_2, \ldots)$ with u_i a member of the M-element alphabet $\{0, 1, \ldots, (M-1)\}$. Rimoldi [21] translated the traditional phase $\varphi(t, \alpha)$ and symmetrical carrier frequency f_0 of Equation 2.2 into the *tilted phase* and *asymmetrical carrier frequency* as it appear in Equation *2.6*. This allowed him to decompose the CPM modulator into a continuous-phase encoder (CPE) and a memoryless modulator (MM). We will use the tilted phase description of the CPM signal throughout this research.

2.4 Description of the 4-*ary* CPFSK signal

Continuous-Phase Frequency-Shift-Keying (CPFSK) is a full-response $L = 1$ CPM scheme characterized by the phase response function

$$f(t) = \begin{cases} 0, & t \leq 0 \\ \frac{t}{2T_s}, & 0 \leq t < T_s \\ \frac{1}{2} & T_s < t \end{cases} \tag{2.8}$$

For $(M = 4)$-*ary* CPFSK, $h = 1/4$. We can now plot the information carrying phase $\psi(t, \mathbf{u})$ versus time for all possible symbol sequences. This phase tree ($\varphi_0 = 0$) is illustrated in Figure 2.1 and the corresponding state trellis in Figure 2.2. On the state trellis, each state σ is associated with a phase state on the phase tree, taken modulo-2π. From the state trellis it can be seen that there is a one-to-one correspondence between the information sequence $\mathbf{u}$ and the state sequence σ, as described by the following equation:

$$u_n = (\sigma_{n+1} - \sigma_n) \bmod(M) \tag{2.9}$$

where "mod" is the modulo operator. A signal $s(t, u_n)$ transmitted during the n-th signalling interval is represented on the state trellis by the vector $X_n = (u_n, \sigma_n)$.

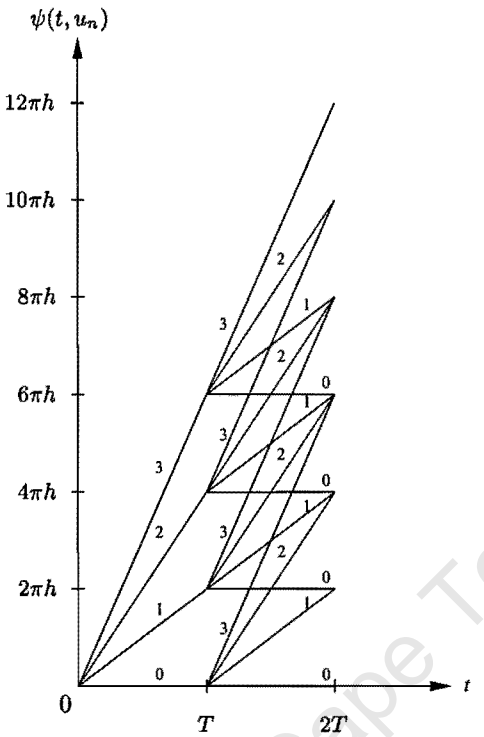


Figure 2.1: Phase tree for 4-*ary* CPFSK over the first two intervals

2.5 Maximum Likelihood Receivers

2.5.1 Maximum Likelihood Sequence Estimation

The following paragraphs are along the lines of [1]. The function of a maximum likelihood detector is to choose that valid signal vector that was most likely to have been transmitted given the received channel output vector. This corresponds to finding that allowable signal vector $\mathbf{s_i}$ that maximizes the log-likelihood function

$$\ln [Pr(\mathbf{r}/\mathbf{s_i})] \tag{2.10}$$

where $\mathbf{r}$ is the received channel output vector that includes noise. It does not require *a priori* information about the parameter of interest. If the transmit signal vector alphabet contains elements that are a priori equally probable, this decision rule can mathematically be reduced to minimization of the function $||\mathbf{r} - \mathbf{s_i}||$ with respect to $\mathbf{i}$.

The term $||\mathbf{r} - \mathbf{s_i}||$ is the distance in vector space between the two vectors (norm), also called the *Euclidean distance*. This shows that the maximum-likelihood decision rule estimates the received signal vector to be that one which is closest in Euclidean distance to one of the element vectors in the transmit signal vector alphabet. Similarly, the function of a maximum likelihood sequence estimator is to find that allowable signal sequence that was most likely to have been transmitted given the received channel output signal. This is equivalent to finding that "allowable" transmitted

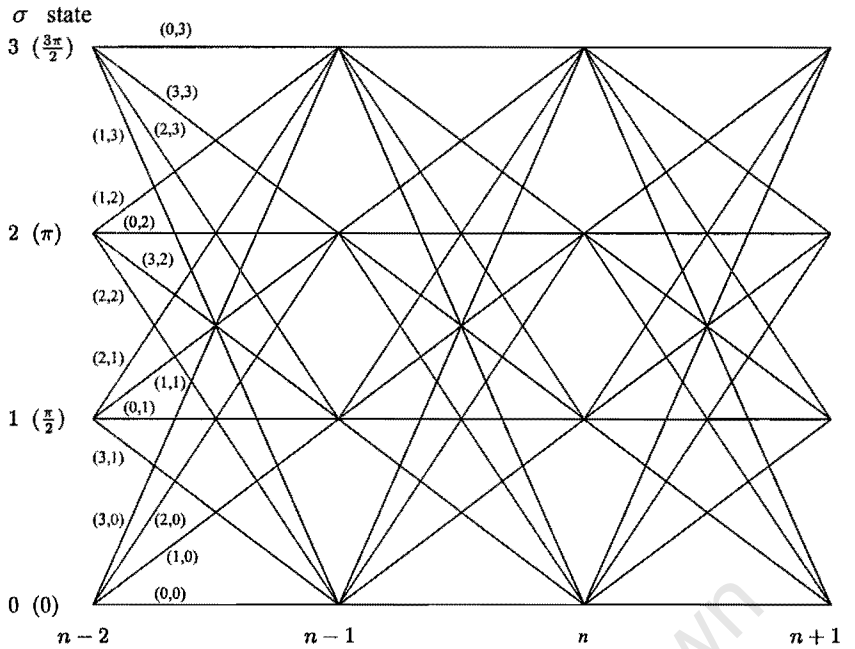


Figure 2.2: State trellis for 4-*ary* CPFSK

signal sequence $s(t, \mathbf{u})$ that maximizes the log-likelihood function

$$\ln [Pr(r(t)/s(t, \mathbf{u}))] \tag{2.11}$$

where $r(t)$ is the received channel output, $r(t) = s(t, \mathbf{u}) + n(t)$, and the noise $n(t)$ is Gaussian and white. The parameter $\mathbf{u}$ signifies a baseband input symbol sequence, where each symbol has length T_s and is selected from a fixed alphabet. In the case of continuous phase modulation (CPM), this baseband signal is used to modulate the phase of some carrier frequency, resulting in the sequence $s(t, \mathbf{u})$, as shown previously. Again, this decision rule can be reduced to minimization of the squared Euclidean distance function

$$||r(t) - s(t, \mathbf{u})||^2 \tag{2.12}$$

In the normed vector space, Parseval's identity can be used to show that:

$$||r(t) - s(t, \mathbf{u})||^2 = \int [r(t) - s(t, \mathbf{u})]^2 \, dt \tag{2.13}$$

Expanding the right side of this equation shows that minimization of this Euclidean distance function is equivalent to maximizing the correlation

$$J(\mathbf{u}) = \int_0^{NT_s} r(t)s(t, \mathbf{u}) \, dt \tag{2.14}$$

where the source sequence $\mathbf{u}$ consists of N symbols. The value of this integral is the metric used by the maximum likelihood receiver to decide the likelihood that a particular signal was sent.

In principle, our maximum likelihood sequence estimator is a correlation receiver, in which all

possible transmitted signals $s(t, \mathbf{u})$ (reference signals) are correlated with the received signal. The reference signal $s(t, \mathbf{u})$ that result in the largest correlation with the received signal $r(t)$ is chosen as the transmitted signal. The symbol sequence $\mathbf{u}$ corresponding to this most likely transmitted signal is obviously our decoded symbol sequence. The number of reference signals $s(t, \mathbf{u})$ required at the receiver grow very fast as the observation period NT_s increases. There are M^Npossibilities, where $M = 4$ for 4-*ary* CPFSK. For example, with $M = 4$ and $N = 100$, one would have to perform 4^{100} correlations! This is not a feasible structure in practice.

Our problem is analogous to finding the shortest (or longest) route between two nodes, with many randomly distributed nodes in between. The Viterbi algorithm is a recursive procedure that was developed to solve this type of problem. Applied to our problem, it finds that sequence which maximizes the log-likelihood function (the metric) up to the nth symbol interval.

2.5.2 The Viterbi Algorithm (VA)

The Viterbi algorithm (VA) [10] is a maximum-likelihood decoder. We can define the metric in Equation 2.14 over the first n intervals of a given source sequence $\mathbf{u}$ as:

$$\lambda_n(u_0, \ldots, u_{n-1}) = \int_0^{NT_s} r(t)s(t, \mathbf{u})\, dt \tag{2.15}$$

Since there is a one-to-one correspondence between the information sequence $\mathbf{u}$ and the state sequence σ as mentioned earlier in that $u_{n-1} = \sigma_n - \sigma_{n-1}$, we can rewrite the last equation as

$$\lambda_n(\sigma_0, \ldots, \sigma_n) = \int_0^{nT_s} s(t, \mathbf{u})r(t)\, dt \tag{2.16}$$

$\lambda_n(\sigma_0, \ldots, \sigma_n)$ can be computed recursively: By replacing n with $n + 1$ in Equation 2.16 we obtain

$$\begin{aligned}
\lambda_{n+1}(\sigma_0, \ldots, \sigma_{n+1}) &= \int_0^{(n+1)T_s} s(t, \mathbf{u})r(t)\, dt && (2.17)\\
&= \int_0^{nT_s} s(t, \mathbf{u})r(t)\, dt + \int_{nT_s}^{(n+1)T_s} s(t, \mathbf{u})r(t)\, dt && (2.18)\\
&= \lambda_n(\sigma_0, \ldots, \sigma_n) + \Delta\lambda(X_n) && (2.19)
\end{aligned}$$

where

$$\Delta\lambda_n(X_n) = \int_{nT_s}^{(n+1)T_s} s(t, \mathbf{u})r(t)\, dt \tag{2.20}$$

Using this recursive approach, the receiver must now, in the n-th interval, computes $\Delta\lambda_n(X_n)$ for all $M^L P$ possible values of X_n and put the result in the state trellis, next to the applicable branch. $\Delta\lambda_n(X_n) = \Delta\lambda_n(u_n, \sigma_n)$ will be called a branch metric.

One way to find the sequence with the largest metric is to add the branch metrics along each possible path in the trellis. However, this would still require us to compute M^N branch metrics.

The number of calculations can be dramatically decreased by using the following observation which is basic to the VA:

For any state at any specified depth in the trellis, the path with the highest overall metric (from root[1] to toor[2]) among all paths visiting that node accumulates also the highest metric between the

[1]The leftmost state in the state trellis is called the root

[2]The rightmost state in the state trellis is called the toor (notation used by [22])

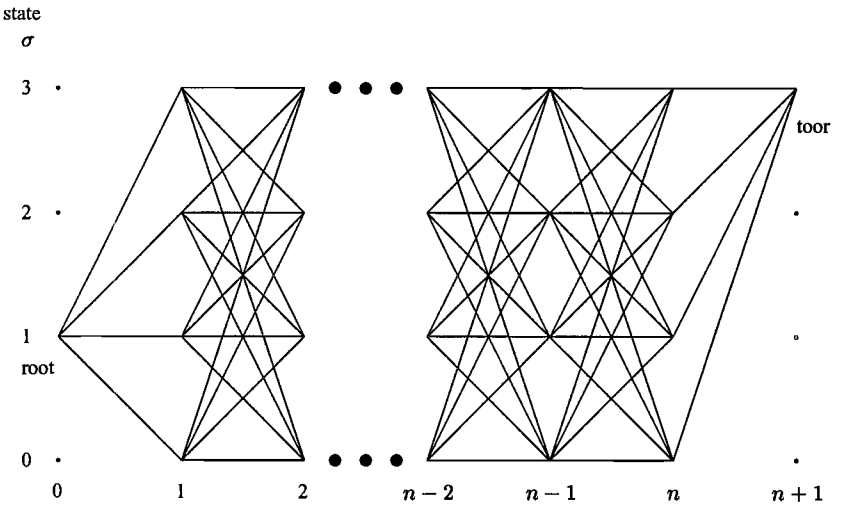


Figure 2.3: Root to Toor State trellis for 4-*ary* CPFSK

root of the trellis and the specified node.

Thus, the path with the highest accumulated metric up to a given node is that node's candidate to have the highest metric at the toor. A root to toor state trellis is shown in Figure 2.3.

The VA then, starts at the root of the trellis and processes all nodes at depth 1, depth 2, and so on until it reaches the toor. At each node at depth $n+1$, it detects the path with the highest metric entering that node and prunes the rest. In other words, at each depth $n+1$ it samples all $M^L P$ branch metrics $\Delta\lambda_n(X_n)$, add all the metrics of the branches that end in state $\sigma_{n+1} = j$ to the metrics of the retained paths at depth n. Of all the paths ending in this state, only the one with the highest metric is retained. This procedure is repeated for all $j \in \{0, 1, \ldots, M^L - 1\}$ at depth $n+1$.

This establishes exactly one connection, i.e. one path, from the root of the trellis to that node. The VA will store the value of the accumulated metric of that path, e.g. labeling it $pathM1[\sigma]$ (path Metric of path Ending-In-State σ at depth $n+1$. At the end of the trellis there is only one node, i.e., the toor, and thus, only one path will connect the root to the toor. This path represents the decoded signal. It may be said that some of the pruned paths crossing a specific node may have a higher overall metric than one of the non-pruned paths crossing the other nodes at the same depth. This may well be the case but remember that the path with the highest overall metric will always be retained – we are not interested in whether the path whose overall metric has been calculated in this way to be the one with the second largest overall metric is really the one with second largest overall metric.

2.5.3 Obtaining the branch metrics

The Viterbi decoder (as well as the other algorithms) will need branch metrics. Following from Equations 2.6, 2.7 and 2.8, for any specific signalling interval n, the alphabet of possible transmitted signals for full-response CPFSK is

$$s(t, X_n) = \sqrt{\frac{2E_s}{T_s}} \cos\left[2\pi f_1 t + \psi(\tau, X_n) + \varphi_0\right] \tag{2.21}$$

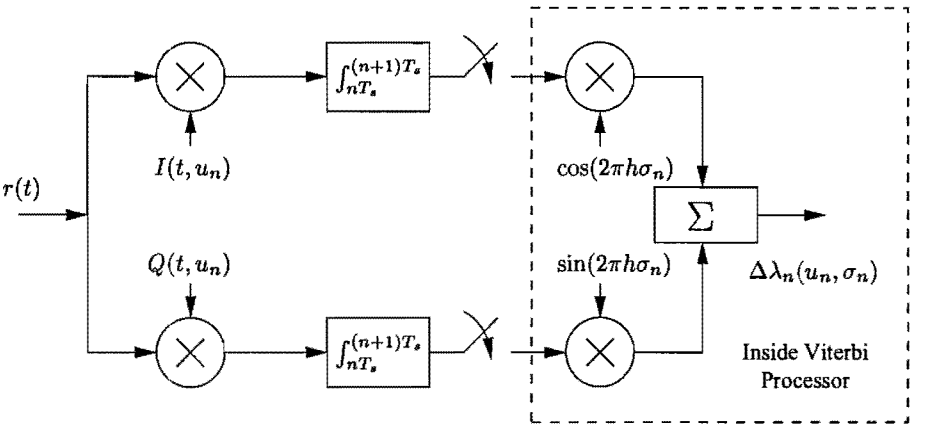


Figure 2.4: Circuit for generating branch metrics

where $X_n = (u_n, \sigma)$ and $t = \tau + nT_s$. We can then write

$$\psi(\tau, X_n) = 2\pi h \left(\frac{u_n}{T_s}\right)\tau + 2\pi h\sigma_n \tag{2.22}$$

where for 4-*ary* CPFSK, $u_n \in \{0, 1, 2, 3\}$, and $\sigma \in \{0, 1, 2, 3\}$. Using a well-known trigonometric conversion and letting $\varphi_0 = 0$, we can write Equation 2.21 as

$$s(t, X_n) = \sqrt{\frac{2E_s}{T_s}}\left\{\cos(2\pi f_1 t)\cos\left[2\pi h\left(\frac{u_n}{T_s}\tau + \sigma_n\right)\right] - \sin(2\pi f_1 t)\sin\left[2\pi h\left(\frac{u_n}{T_s}\tau + \sigma_n\right)\right]\right\} \tag{2.23}$$

or as

$$s(t, X_n) = \sqrt{\frac{2E_s}{T_s}}\left\{\cos(2\pi h\sigma_n)\, I(t, u_n) - \sin(2\pi h\sigma_n)\, Q(t, u_n)\right\} \tag{2.24}$$

where

$$I(t, u_n) = \cos\left[2\pi f_1 t + 2\pi h\frac{u_n}{T_s}\tau\right] \tag{2.25}$$

$$Q(t, u_n) = -\sin\left[2\pi f_1 t + 2\pi h\frac{u_n}{T_s}\tau\right] \tag{2.26}$$

Substituting Equation 2.24 in 2.20 result in:

$$\Delta\lambda_n(X_n) = \sqrt{\frac{2E_s}{T_s}}\left\{\cos(2\pi h\sigma_n)\int_{nT_s}^{(n+1)T_s} I(t, u_n)r(t)\,dt + \sin(2\pi h\sigma_n)\int_{nT_s}^{(n+1)T_s} Q(t, u_n)r(t)\,dt\right\} \tag{2.27}$$

Our branch metric for 4-*ary* CPFSK can be solved by using, for each u_n, a set of correlators as shown in Figure 2.4. A more general and mathematically complete outline of the above procedure is presented in [1] and [22].

For 4-*ary* CPFSK, we can eliminate the need for the Viterbi processor to perform the two multiplications and one addition required for each u_n in the alphabet. We start by making a list of all possible transmitted signals in a signalling interval. Setting $\varphi_0 = 0$ and using the notation $S_{u_n,\sigma_n}(t)$ instead of $s(t, X_n)$, we list the signals as:

$$
\begin{array}{ll}
\text{Signal} & \text{Signal}/\sqrt{\frac{2E_s}{T_s}} \\
S_{0,0}(t): & \mathrm{Re}\left\{e^{\mathrm{j}0}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{1,0}(t): & \mathrm{Re}\left\{e^{\mathrm{j}2\pi h\frac{1}{T_s}\tau}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{2,0}(t): & \mathrm{Re}\left\{e^{\mathrm{j}2\pi h\frac{2}{T_s}\tau}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{3,0}(t): & \mathrm{Re}\left\{e^{\mathrm{j}2\pi h\frac{3}{T_s}\tau}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{0,1}(t): & \mathrm{Re}\left\{e^{\mathrm{j}\frac{\pi}{2}}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{1,1}(t): & \mathrm{Re}\left\{e^{\mathrm{j}2\pi h\left(\frac{1}{T_s}\tau+1\right)}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{2,1}(t): & \mathrm{Re}\left\{e^{\mathrm{j}2\pi h\left(\frac{2}{T_s}\tau+1\right)}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{3,1}(t): & \mathrm{Re}\left\{e^{\mathrm{j}2\pi h\left(\frac{3}{T_s}\tau+1\right)}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{0,2}(t): & \mathrm{Re}\left\{e^{\mathrm{j}\pi}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{1,2}(t): & \mathrm{Re}\left\{e^{\mathrm{j}2\pi h\left(\frac{1}{T_s}\tau+2\right)}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{2,2}(t): & \mathrm{Re}\left\{e^{\mathrm{j}2\pi h\left(\frac{2}{T_s}\tau+2\right)}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{3,2}(t): & \mathrm{Re}\left\{e^{\mathrm{j}2\pi h\left(\frac{3}{T_s}\tau+2\right)}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{0,3}(t): & \mathrm{Re}\left\{e^{\mathrm{j}\frac{3\pi}{2}}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{1,3}(t): & \mathrm{Re}\left\{e^{\mathrm{j}2\pi h\left(\frac{1}{T_s}\tau+3\right)}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{2,3}(t): & \mathrm{Re}\left\{e^{\mathrm{j}2\pi h\left(\frac{2}{T_s}\tau+3\right)}e^{\mathrm{j}2\pi f_1 t}\right\} \\
S_{3,3}(t): & \mathrm{Re}\left\{e^{\mathrm{j}2\pi h\left(\frac{2}{T_s}\tau+3\right)}e^{\mathrm{j}2\pi f_1 t}\right\}
\end{array}
$$

We notice that

$$
\begin{aligned}
S_{0,0}(t) &= -S_{0,2}(t) & S_{0,1}(t) &= -S_{0,3}(t) \\
S_{1,0}(t) &= -S_{1,2}(t) & S_{1,1}(t) &= -S_{1,3}(t) \\
S_{2,0}(t) &= -S_{2,2}(t) & S_{2,1}(t) &= -S_{2,3}(t) \\
S_{3,0}(t) &= -S_{3,2}(t) & S_{3,1}(t) &= -S_{3,3}(t)
\end{aligned}
$$

and consequently that

$$
\begin{aligned}
\Delta\lambda_n(0,0) &= -\Delta\lambda_n(0,2) & \Delta\lambda_n(0,1) &= -\Delta\lambda_n(0,3) \\
\Delta\lambda_n(1,0) &= -\Delta\lambda_n(1,2) & \Delta\lambda_n(1,1) &= -\Delta\lambda_n(1,3) \\
\Delta\lambda_n(2,0) &= -\Delta\lambda_n(2,2) & \Delta\lambda_n(2,1) &= -\Delta\lambda_n(2,3) \\
\Delta\lambda_n(3,0) &= -\Delta\lambda_n(3,2) & \Delta\lambda_n(3,1) &= -\Delta\lambda_n(3,3)
\end{aligned}
$$

Therefore, we only need to generate the following set of metrics at the receiver:

$$
\begin{aligned}
\Delta\lambda_n(0,0) &= \int S_{0,0}(t)r(t)\,dt & \Delta\lambda_n(0,1) &= \int S_{0,1}(t)r(t)\,dt \\
\Delta\lambda_n(1,0) &= \int S_{1,0}(t)r(t)\,dt & \Delta\lambda_n(1,1) &= \int S_{1,1}(t)r(t)\,dt \\
\Delta\lambda_n(2,0) &= \int S_{2,0}(t)r(t)\,dt & \Delta\lambda_n(2,1) &= \int S_{2,1}(t)r(t)\,dt \\
\Delta\lambda_n(3,0) &= \int S_{3,0}(t)r(t)\,dt & \Delta\lambda_n(3,1) &= \int S_{3,1}(t)r(t)\,dt
\end{aligned}
$$

We can likewise show that for MSK:

$$
\begin{aligned}
S_{0,0}(t) &= -S_{0,1}(t) \\
S_{1,0}(t) &= -S_{1,1}(t)
\end{aligned}
$$

The set of metrics for 4-*ary* CPFSK can be generated as shown in Figure 2.5. Figure 2.6 shows the 4-*ary* CPFSK baseband signals.

2.5.4 Implementation of the VA

The VA as explained before pertains to the case where the trellis starts at a specific node (the root), and ends in a specific state (the toor). It was also assumed that we need to receive the whole sequence before we can make an optimum decision about it. This is not exactly what we want:

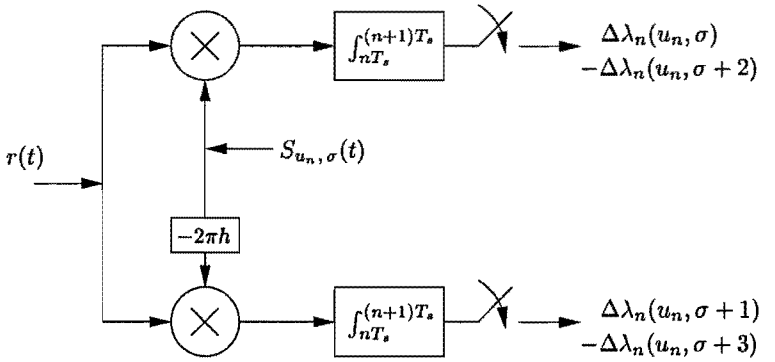


Figure 2.5: Simplified circuit for generating branch metric

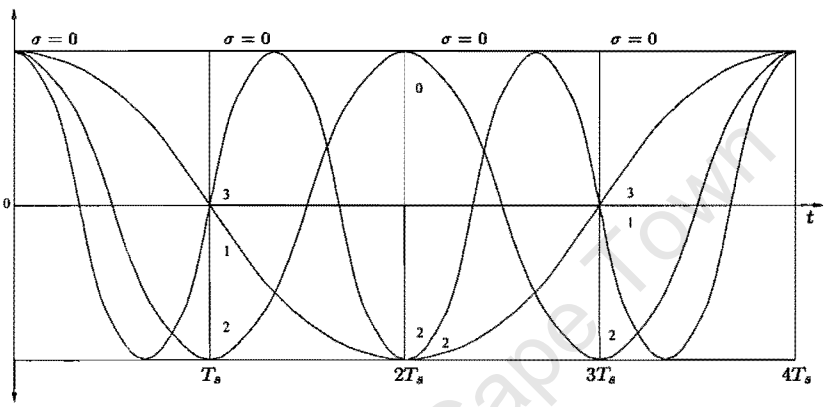


Figure 2.6: 4-*ary* CPFSK baseband signals

1. We want our VA processor to find that single path, with the largest metric, between any node existing at the starting depth and those existing at the termination depth. It is a characteristic of the VA that if we program our VA to process all possible branches at the starting depth and the termination depth, it will find this path.

2. We want to make our decisions on-the-fly, not waiting until the whole sequence is received. Intuitively, we know we have to allow the VA a certain number of intervals (D, the observation period) before making an optimum decision about the symbol transmitted during the first interval. We expect this delay to be a function of the inherent memory of the CPM signal.

   It has been proven by Rimoldi [21] that for MSK, this observation period is equal to two symbol intervals. Stated in another way, he showed that the VA will, after processing the metrics at depth n, point out a root at depth $n-1$. This means we can optimally decode the state sequence up to this root and from $u_{n-1} = \sigma_n - \sigma_{n-1}$, we can obtain the symbol sequence. This means that the symbol sequence decision is delayed by one interval.
   For 4-*ary* CPFSK, proving the value of the decision delay ($D-1$) involves lengthy mathematical manipulations. Intuitively one would expect this observation period to be larger than or equal to two intervals. We will establish this decision delay through simulation later in this chapter.

2.6 Bit error rate performance

The communication channel used in this thesis is assumed to be contaminated with AWGN. The received signal can then be written as $r(t) = s(t) + w(t)$, where $w(t)$ is a stationary Gaussian random process with constant two-sided power spectrum density $N_o/2$ (W/Hz).

The following paragraphs follow the descriptions as in [1]. To determine the probability of making an error in the detection of $r(t)$ when $s_i(t)$ was sent, we have to integrate the likelihood function $f_R[r(t)/s_i(t)]$ over a *signal space* where this *space* excludes all points that are closer to $s_i(t)$ than to any other signal. This procedure must be repeated for each transmittable signal, the values multiplied with the respective *a priori* probabilities of the signals and then summed to determine the average error probability.

The conditional probability density functions, $f_R[r(t)/s_i(t)]$, for each transmitted signal $s_i(t)$ are called *likelihood functions*. The likelihood function [1] of $s_i(t)$ is

$$f_R[r(t)/s_i(t)] = \prod_{j=1}^{J_s} (\pi N_o)^{\frac{-1}{2}} \exp\left[\frac{-(r_j - s_{ij})^2}{N_o}\right] \tag{2.28}$$

where J_s is the dimension of the signal space, and r_j and s_{ij} are the components of $r(t)$ and $s_i(t)$ respectively, in the direction of the jth basis function. $r(t)$ is the received signal.

It is clear from this that the signal space distance between the elements of the transmit signal vector alphabet determines the probability of error. For 4-*ary* CPFSK, integrating the likelihood functions are impractical, and we therefore resort to "bounds".

Let $P_e(k, i)$ be the probability that the data bearing signal $s_k(t)$ is chosen by the maximum likelihood receiver, given that $s_i(t)$ was sent. From the likelihood function, this will happen only if the received $r(t)$ is closer to $s_k(t)$ than to $s_i(t)$. This situation in signal space is sketched in Figure 2.7 [1]. Since white noise is identically distributed along any set of orthogonal axes, let these axis temporarily be chosen so that the first axis runs directly through the signal space points corresponding to $s_i(t)$ and $s_k(t)$. Here the first axis corresponds to $j = 1$ in the likelihood function.

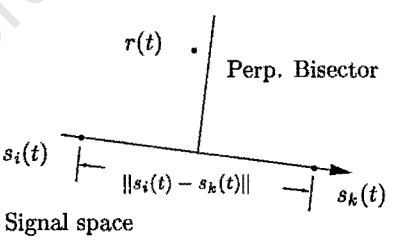


Figure 2.7: $s_i(t)$ and $s_k(t)$ in signal space.

The perpendicular bisector of the line between the signals is the locus of points equidistant from both signals. The probability that $r(t)$ lies on the "$s_k(t)$" side of the bisector is the probability that the first component $r_1 - s_{i1}$ exceeds half of the signal space distance $\Delta = ||s_i(t) - s_k(t)||$.

The error probability $P_e(k, i)$ is given by

$$P_e(k, i) = \int_{\frac{\Delta}{2}}^{\infty} \frac{\exp\left(\frac{-u^2}{N_o}\right)}{\sqrt{\pi N_o}} du \tag{2.29}$$

since $r_1 - s_{i1}$ is a Gaussian random variable with mean 0 and variance $N_o/2$.

Next we need to find the probability that any other signal is detected besides $s_i(t)$. To do this we use the *union bound* of probability, which states that if A_1, A_2, ... are events, the probability $\Pr\{\text{union } A_i\}$ that one or more happens is over bounded by $\sum_i \Pr\{A_i\}$. Consequently, the probability of error if $s_i(t)$ is sent is bounded by

$$P_e(i) \leq \sum_{k \neq i} \int_{\frac{||s_i(t)-s_k(t)||}{2}}^{\infty} \frac{\exp\left(\frac{-u^2}{N_o}\right)}{\sqrt{\pi N_o}}\, du \tag{2.30}$$

The total average probability of error is

$$P_e = \sum_i P_e(i) \Pr\{s_i(t) \text{ sent}\} \tag{2.31}$$

We assume that all transmitted signals are equally likely. Then we can write the last equation as:

$$P_e = \frac{1}{W} \sum_i P_e(i) \tag{2.32}$$

where W is the size of the transmit signal alphabet. In terms of the Q function, we can rewrite Equation 2.30 as

$$P_e(i) \leq \sum_{k \neq i} Q\left(\frac{||s_i(t) - s_k(t)||}{\sqrt{2N_o}}\right) \tag{2.33}$$

The Euclidean distance $||s_i(t) - s_k(t)||$ simplifies considerably for constant envelope phase–varying signals such as 4-*ary* CPFSK, since the distance between two signals depends only on the phase difference between them.

In the normed vector space, Parseval's identity can be used to show that the signal space distance between the signals $s_i(t)$ and $s_k(t)$, D^2, can be written as:

$$D^2 = ||s_i(t) - s_k(t)||^2 \tag{2.34}$$

$$= \int^{NT_s} [s_i(t) - s_k(t)]^2\, dt \tag{2.35}$$

where the integration is over NT_s seconds. We can write the normalized squared Euclidean distance as

$$d^2 = \frac{D^2}{2E_b} \tag{2.36}$$

$$= \frac{1}{2E_b} \int^{NT_s} [s_i(t) - s_k(t)]^2\, dt \tag{2.37}$$

$$= \sum_{n=0}^{N-1} \frac{1}{2E_b} \int_{nT_s}^{(n+1)T_s} [s_i(t) - s_k(t)]^2\, dt \tag{2.38}$$

$$= \sum_{n=0}^{N-1} d_n^2 \tag{2.39}$$

where d_n^2 is the n-th incremental normalized squared Euclidean distance (see [23]). Defining the phase difference at time nT_s as

$$\Delta\psi_n = \psi(nT_s, \mathbf{u}_i) - \psi(nT_s, \mathbf{u}_k) \tag{2.40}$$

where $\mathbf{u}_i$ and $\mathbf{u}_k$ are the information sequences corresponding to $s_i(t)$ and $s_k(t)$, respectively. We find that d_n^2 simplifies [4] to

$$d_n^2 = \begin{cases} \frac{1-\sin\Delta\psi_{n+1}-\sin\Delta\psi_n}{\Delta\psi_{n+1}-\Delta\psi_n}, & \Delta\psi_{n+1} \neq \Delta\psi_n \\ 1-\cos\Delta\psi_n, & \Delta\psi_{n+1} = \Delta\psi_n \end{cases} \tag{2.41}$$

We see that d_n^2 depends only on the phase difference at the beginning and at the end of the n-th interval and is symmetrical with respect to $\Delta\psi_n$ and $\Delta\psi_{n+1}$.

When the ratio of signal energy to noise energy is reasonably high, it turns out that the term(s) in Equation 2.33 with the smallest distance $||s_i(t) - s_k(t)|| = D_{min}$ among all possible signals will strongly dominates the $P_e(i)$ expression. At these reasonably high SNRs we can write the approximate bound as

$$P_e(i) \leq K_i Q\left(\frac{D_{min}}{\sqrt{2N_o}}\right) \tag{2.42}$$

$$\leq K_i Q\left(\min_{k\neq i} d(s_i, s_k)\sqrt{\frac{E_b}{N_o}}\right) \tag{2.43}$$

where K_i is the number of signals that attain the minimum distance with respect to signal $s_i(t)$. We can bound the total average error probability as

$$P_e \leq Q\left(\sqrt{\frac{d_{min}^2 E_b}{N_o}}\right) \sum_i \frac{K_i}{W} \tag{2.44}$$

where $d_{min}^2 = \min_{k\neq i} d^2(s_i, s_k)$.

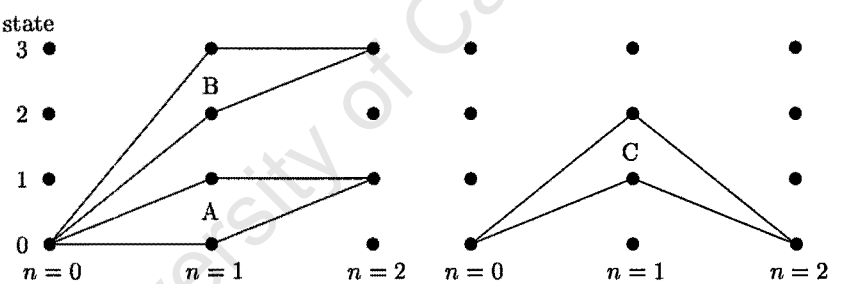


Figure 2.8: A few minimum distance error events for 4-*ary* CPFSK

Figure 2.8 shows minimum distance events on the state trellis of 4-*ary* CPFSK. With the help of this figure and Equations 2.39 and 2.41, we can show that for 4-*ary* CPFSK, d_{min}^2 is equal to 1.454, $K_i = 18$ and $W = 16$. It follows that

$$P_e \leq \frac{18}{16} Q\left(\sqrt{\frac{1.454E_b}{N_o}}\right) \tag{2.45}$$

In [24] a formula is derived for the exact value of D_{min}^2 for CPFSK in general. The Q-function above defines the probability that an observed value of a Gaussian random variable X of zero mean and unit variance will be greater than v:

$$Q(v) = \frac{1}{\sqrt{2\pi}} \int_v^\infty \exp\left(-\frac{x^2}{2}\right) dx \tag{2.46}$$

2.6.1 Bit error rate (BER) simulation

For bit error rate simulation, Gaussian noise, having a standard deviation σ and mean equal to 0, was added to the channel.

The power from the Gaussian noise source is:

$$P_{ENref} = \frac{\sigma^2}{R_{ref}} \quad \text{(W)} \tag{2.47}$$

where σ is also the RMS value of the output of the noise source in volts, and R_{ref} is the reference resistance, which is equal to $1\,\Omega$ in our simulation.

Sampling the baseband noise at $1/T_{step}$ Hz, the one-sided bandwidth of the noise process will be $1/(2T_{step})$ Hz (T_{step} is the simulation time step).

The power spectrum density of the noise N_o will then be

$$\begin{aligned} S_n(f) &= \frac{2\sigma^2 T_{step}}{R_{ref}} \quad \text{(W/Hz)} && (2.48)\\ &= 2\sigma^2 T_{step} && (2.49)\\ &= N_o \quad \text{(W/Hz)} && (2.50) \end{aligned}$$

for $0 \leq f \leq 1/(2T_{step})$.

The symbol energy is

$$\begin{aligned} E_s &= P_{ESref} T_s \\ &= E_b \log_2 M \end{aligned}$$

and then follows that

$$\frac{E_b}{N_o} = \frac{P_{ESref} T_s}{2\sigma^2 T_{step} \log_2 M} \tag{2.51}$$

where P_{ESref} is the carrier power.

T_{step} was chosen small enough

1. for the noise to appear white to the system
2. to ensure sufficient accuracy of the integration algorithm.

The bit error simulations were done using "commsim"[3], a simulation program written in the C language. See appendix 2.10 for a summary of how the code was implemented.

For comparison purposes the error probabilities of various modulation schemes are listed in Table 2.1. The entry for 4-*ary* CPFSK in the table was obtained from simulations and does not agree exactly with the theory (Equation 2.45), although it must be said that Equation 2.45 specify a lower bound on the error probability. See Figures 2.9 and 2.10 for bit error simulation results. The optimum observation period (D) for 4-*ary* CPFSK is, according to these results equal to $D = 3$. These results will be used as reference for the experiments in the next chapters. About 30 to 40 percent of the errors made by the 4-*ary* CPFSK Viterbi decoder occurred as single symbol errors, the remaining occurred in bursts.

[3]This program is not related to any other program that may have an identical name. I am not aware of the existence of another program with this name.

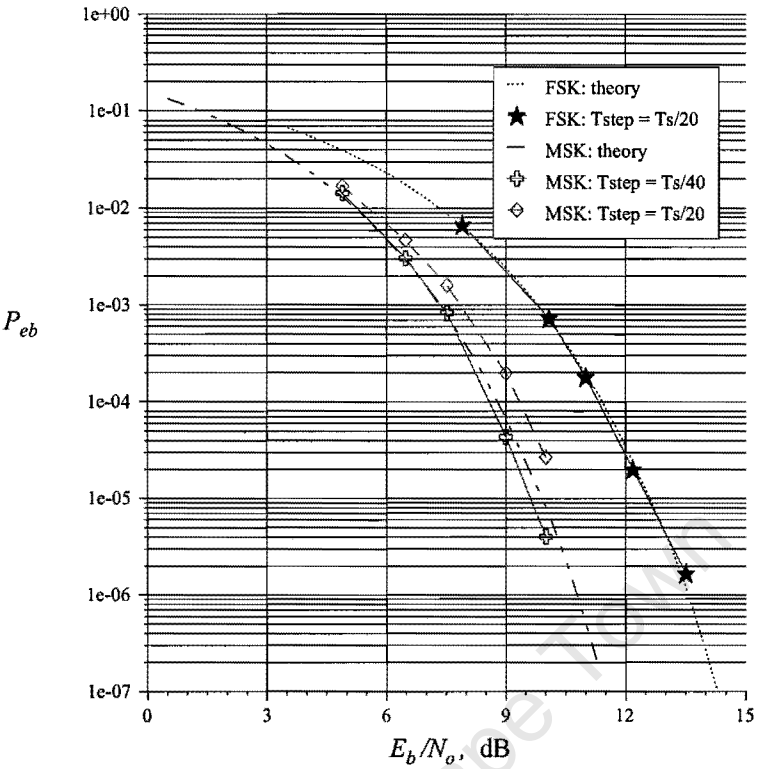


Figure 2.9: BER results for MSK and FSK

Modulation scheme	P_e
BPSK	$Q\left(\sqrt{\frac{2E_b}{N_o}}\right)$
DPSK	$\frac{1}{2}\exp\left(\frac{-E_b}{N_o}\right)$
QPSK and OQPSK	$\simeq 2Q\left(\sqrt{\frac{2E_b}{N_o}}\right)$
MPSK ($M \geq 4$)	$\simeq Q\left(\sqrt{\frac{4E_b \log_2 M}{N_o}} \sin\left(\frac{\pi}{2M}\right)\right)$
Coherent Binary FSK	$Q\left(\sqrt{\frac{E_b}{N_o}}\right)$
Noncoherent Binary FSK	$\frac{1}{2}\exp\left(\frac{-E_b}{2N_o}\right)$
MSK	$\simeq 2Q\left(\sqrt{\frac{2E_b}{N_o}}\right)$
GMSK	$\simeq Q\left(\sqrt{\frac{1.36E_b}{N_o}}\right)$ for $BT = 0.25$
M-*ary* QAM	$\simeq 4\left(1-\frac{1}{\sqrt{M}}\right)Q\left(\sqrt{\frac{3E_{ave}}{(M-1)N_o}}\right)$
MFSK	$\leq (M-1)Q\left(\sqrt{\frac{E_b \log_2 M}{N_o}}\right)$
4-*ary* CPFSK	$\leq 4Q\left(\sqrt{\frac{1.454E_b}{N_o}}\right)$

Table 2.1: P_e equations for various modulation schemes

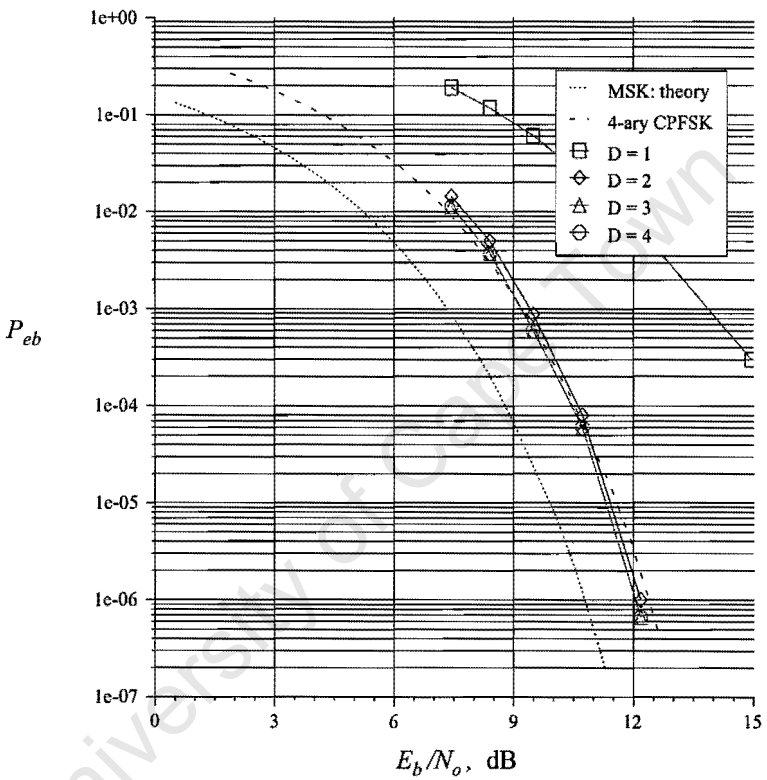


Figure 2.10: BER results for 4-*ary* CPFSK, $T_{step} = T_s/40$

Chapter 3

Sequential Decoding Algorithms

3.1 Sequential Decoding

In this section we will describe sequential decoding and the Fano algorithm through a discussion of its application to 4-*ary* CPFSK. Wozencraft [34] gives a very good description of sequential decoding, including the Fano algorithm, but his discussion is based on an application to the discrete binary symmetric channel.

Given a 4-*ary* CPFSK demodulator that provides us with a full set of metrics, let us consider the outputs when the input is not corrupted by noise. The decoder starts at the root of the trellis, read in all metrics associated with the *branches diverging from this node* and follows the branch with the largest correlation (metric) to the next level node. Having thus been directed to a particular next depth node of the trellis, the decoder again read in all metrics associated with the branches diverging from this next level node and continue in this fashion. This works fine when no noise is present. However, when noise corrupts our demodulator in such a way that one of the branches diverging from the node under discussion receives a metric that is larger than the branch which is suppose to give the largest metric, our decoder will initially proceeds along the incorrect path. I say initially since it is possible that our path will cross the correct node farther down the trellis, and it is possible that we may continue on the correct path from this point. This will obviously result in an error in the decoding process. Once on the wrong path, the decoder is unlikely to find any path stemming from the initial incorrect node which agrees well with the received sequence.

The idea of sequential decoding is to program the decoder to act much as a person who makes a "best" choice at a cross-road on the basis of information laid before him about the short term smoothness of the possible paths. If his best choice was an incorrect one, his path will soon become increasingly bumpy. He will intelligently choose a "bumpyness" threshold or discard criterion. Violation of this threshold signals an incorrect decision somewhere back. He then goes back and tries the other best choice. Even if he crosses the correct path on one of his future crossroads and makes a correct decision, the effect of his previous incorrect choice is suppose to cleave to him, and the effect of any further incorrect choices will add to this memory. The principle to be exploited is that only at those rare times where we do make an incorrect decision do we need to go back and investigate other best options.

Let us assume that the decoder has penetrated l intervals into the trellis, $l \geq 0$. Let $d(l)$ denotes the accumulated path metric observed by the decoder, between the tentative path it is following

and the corresponding l-interval segment of the received sequence. As the decoder penetrates deeper into the trellis, it maintains a running count of $d(l)$. After each successive penetration the decoder compares $d(l)$ against a discard criterion function, $\Delta(l)$. If $d(l)$ ever exceeds $\Delta(l)$, the tentative path is discarded as too improbable. The decoder then backs up to the next best unexplored branch for which $d(l) \leq \Delta(l)$ and starts moving forward as far as the discard criterion function $\Delta(l)$ permits. If the tentative path is the correct path, then $d(l) = 0$ and $\Delta(l)$ will be a constant. Due to noise $d(l)$ will oscillate around this zero value. When the tentative path departs from the *starting node* along the incorrect branch, we anticipate that $d(l)$ will oscillates about a line with an average slope $(p - v_{ave})$, where p is the branch metric value $\Delta\lambda_n(x, y)$ of its corresponding received signal $S_n(x, y)$ when no noise is present, while v_{ave} will be the average of the single largest metric available at each of the possible "incorrect" states. Thus, we cannot talk about $d(l)$ being a straight line as discussed by Wozencraft for the binary discrete case where the Hamming distance between any two branches are a constant number of bits.

In our case, the distance is not equal, but is a function of the specific received symbol and the present "incorrect" state from which we look forward. The instantaneous value of v will be an element of a set of best metrics as seen from the possible "incorrect" states. The best possible metrics[1] (normalized) for 4-*ary* CPFSK as seen from an incorrect state, are listed in Table 3.1 as obtained from Table 3.2. The maximum and minimum values of v are 0.32 and 0, respectively, and the average value of the best possible metrics is $v_{ave} = 0.125$. A typical plot of $d(l)$ can be seen in Figure 3.1 showing an incorrect decision at interval 1.

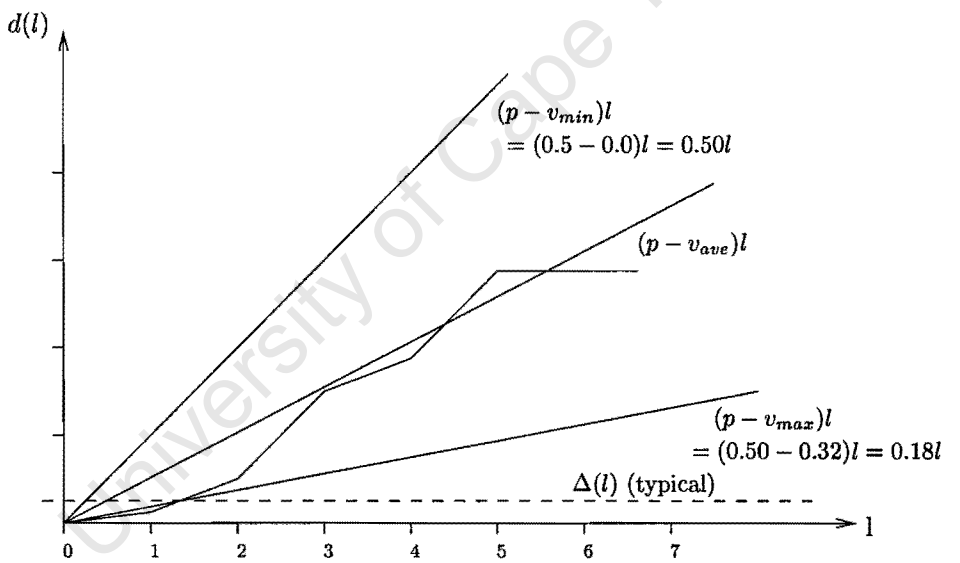


Figure 3.1: Typical plot of $d(l)$ showing incorrect decision at interval 1

$r(t)$	$\Delta\lambda_{max}(x,0)$	$\Delta\lambda_{max}(x,1)$	$\Delta\lambda_{max}(x,2)$	$\Delta\lambda_{max}(x,3)$
$S_{y,0}(t)$	—	0.0, 0.32	0.0, 0.11	0.0, 0.32
$S_{y,1}(t)$	0.0, 0.32	—	0.0, 0.32	0.0, 0.11
$S_{y,2}(t)$	0.0, 0.11	0.0, 0.32	—	0.0, 0.32
$S_{y,3}(t)$	0.0, 0.32	0.0, 0.11	0.0, 0.32	—

Table 3.1: Set of Best Metrics as seen from "incorrect" state

[1]The values in the table are not highly accurate but rounded off in order to reduce the size of the metric alphabet and to simplify the examples that follow.

$\Delta\lambda_n \rightarrow$ $r(t) \downarrow$	(0,0)	(1,0)	(2,0)	(3,0)	(0,1)	(1,1)	(2,1)	(3,1)
$S_{0,0}(t)$	0.50	0.32	0.00	-0.11	0.00	-0.32	-0.32	-0.11
$S_{1,0}(t)$	0.32	0.50	0.32	0.00	0.32	0.00	-0.32	-0.32
$S_{2,0}(t)$	0.00	0.32	0.50	0.32	0.32	0.32	0.00	-0.32
$S_{3,0}(t)$	-0.11	0.00	0.32	0.50	0.11	0.32	0.32	0.00
$S_{0,1}(t)$	0.00	0.32	0.32	0.11	0.50	0.32	0.00	-0.11
$S_{1,1}(t)$	-0.32	0.00	0.32	0.32	0.32	0.50	0.32	0.00
$S_{2,1}(t)$	-0.32	-0.32	0.00	0.32	0.00	0.32	0.50	0.32
$S_{3,1}(t)$	-0.11	-0.32	-0.32	0.00	-0.11	0.00	0.32	0.50

Table 3.2: Normalized metrics for 4-*ary* CPFSK, $E_s = 0.5$ J and $T_s = 1$ second.

Let us assume that the decoder need to penetrate at least K intervals, starting from node n, to recognize an incorrect choice at node $n-1$. Then our decoder may decode σ_n once it has analyzed up to interval $n+K$ without exceeding the discard criterion (threshold). The *first basic concept* of sequential decoding is that *the saving in the number of computations decrease exponentially as K becomes smaller.* With the threshold defined as Δ, the worst case K for 4-*ary* CPFSK will be:

$$K_{max} = \text{ceil}\left(\frac{\Delta}{p - v_{max}}\right) \tag{3.1}$$

where $p = 0.50$, $v_{max} = 0.32$ and ceil(x) is the least integer greater than or equal to x.

It is clear that the average number of computations is reduced by making Δ small, but if Δ is too small, channel noise may cause many branches in the trellis to exceed Δ, triggering a look-back subroutine which slows down progress.

This lead to the definition of the *second basic concept* of sequential decoding: *On these less frequent occasions (given a reasonable S/N) when all branches are discarded, a less stringent criterion, such as a function Δ_2 is invoked, or Δ_3 in more severe cases.*

An algorithm that permits efficient and flexible implementation of these basic concepts was devised by Fano [9].

3.2 The Fano algorithm

Fano adopted a "tilted" distance function, which I adapted for our application to the following form:

$$t(l) = d(l) - (p - v)l \tag{3.2}$$

The corresponding discard criteria, called thresholds, are horizontal lines with spacing Δ. When following the correct path, we have that $t(l) = -(p - v)l$.

As we update $t(l)$ along the way, a received "distance" tree is formed. A node of the received distance tree is said to satisfy all thresholds that lie on or above it and to violate all thresholds that lie beneath it. The tightest threshold satisfied by a node is the one that lies just on or above it. Of the nodes diverging from any given node, the one with the largest metric is called the *best* and the ones with smaller metrics are called *worst.*

The Fano algorithm maintains a running threshold, denoted by δ and equal to $k\Delta$, where k is a (variable) integer. We say the running threshold is *tightened* when k is assigned so that δ is the tightest threshold satisfied by the search node, that is, by the node then being considered. An essential feature of the algorithm is that we are not allowed to move either forward or backward

unless this can be accomplished without violating the running threshold; the running threshold is raised only when necessary to accommodate such a move. The procedure is best explained by the flow diagram in Figure 3.2 and the following examples. In the examples that follow, metric values typical for $E_b/N_0 = 9.0$ dB are used.

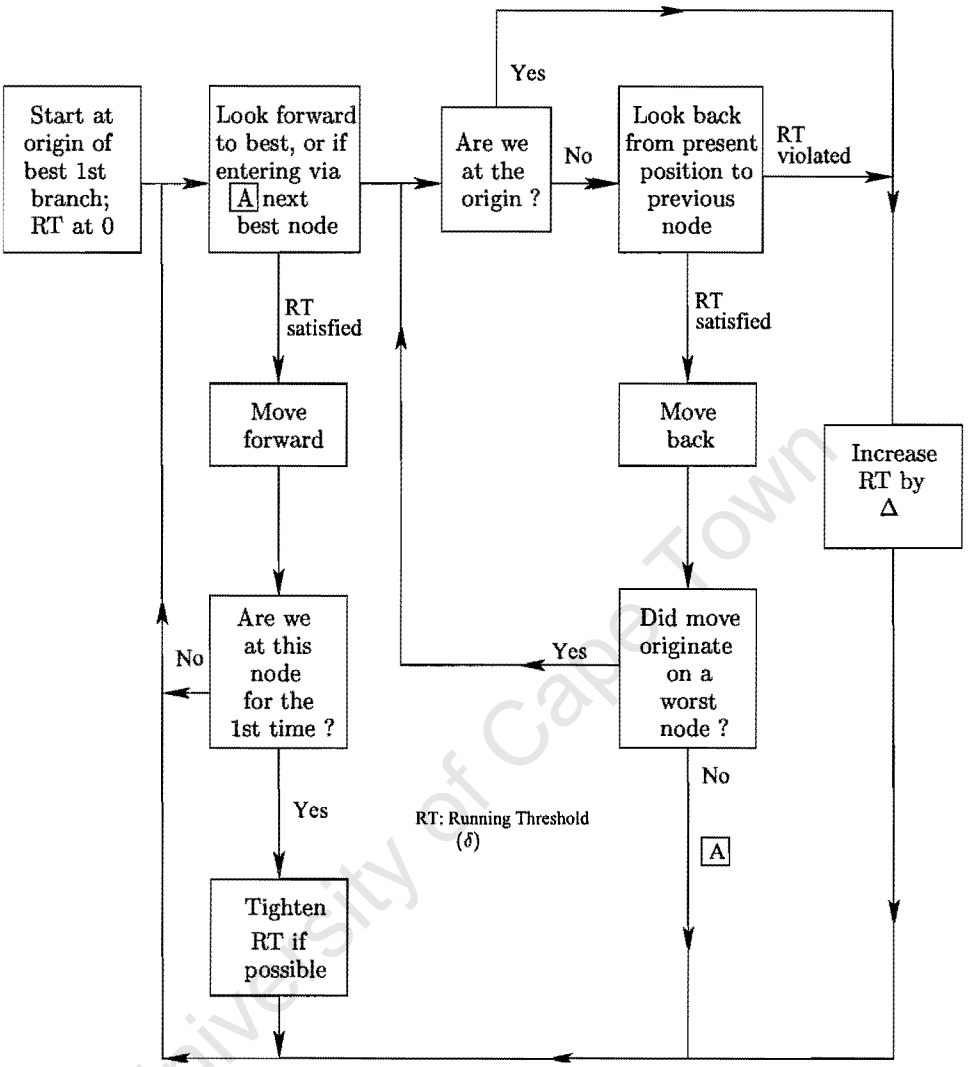


Figure 3.2: The basic Fano Algorithm

Example 1: $(p - v) = 0.30 \approx (p - v_{ave})$; $\Delta = 0.30$

Given the received sequence as in Figure 3.3, let us assume that we've been following the correct path but make an erroneous decision at the $3rd$ interval (indicated by a *). Table 3.3 and Figure 3.4 show the tree searching progression.

The algorithm failed to detect the incorrect path early enough, and re-merging with the correct path occur, resulting in an error event as shown by the dotted line on Figure 3.3. The following should be considered when trying to explain the failure of the algorithm for this example:

1. The smaller the value of $(p - v)$, the higher the likelihood that the look-back subroutine will be triggered unnecessary, especially when $t(l)$ are just below the running threshold. If

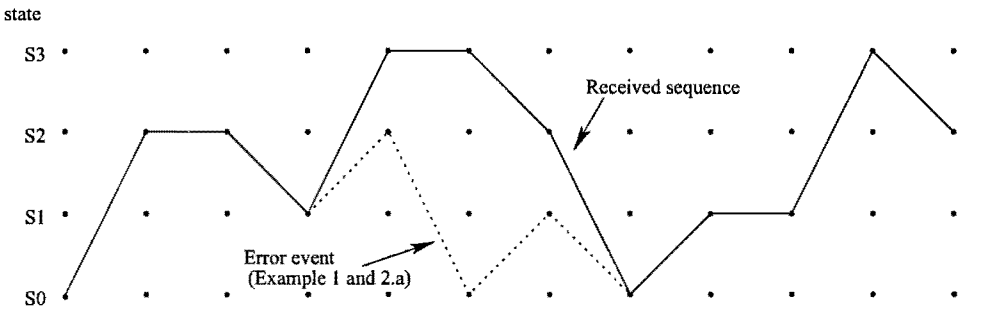


Figure 3.3: Received sequence for Example 1 and 2.a

At	δ	action	$\Delta\lambda_n(x,y)$	$d(l)$	$t(l)$	action
0	0.00	look at 1	0.40	0.10	-0.20	move to 1
1	0.00	look at 2	0.43	0.17	-0.43	move to 2, set $\delta = -0.30$
2	-0.30	look at 3	0.49	0.18	-0.72	move to 3, set $\delta = -0.60$
3*	-0.60	look at 4	0.40	0.28	-0.92	move to 4, set $\delta = -0.90$
4*	-0.90	look at 5	0.38	0.40	-1.10	move to 5
5*	-0.90	look at 6	0.28	0.62	-1.18	move to 6
6*	-0.90	look at 7	0.21	0.91	-1.19	move to 7
7	-0.90	look at 8	0.51	0.90	-1.50	move to 8, set $\delta = -1.5$
8	-1.50	look at 9	0.49	0.91	-1.79	move to 9
9	-1.50	look at 10	0.48	0.93	-2.07	move to 10, set $\delta = -1.80$

Table 3.3: Tree search for Example 1 ($(p-v) = 0.3$, and $\Delta = 0.3$)

required, the Fano algorithm will increase the running threshold in order for the algorithm to continue, however, unnecessary computations will result,

2. If the value of $(p-v)$ is too large, the look-back subroutine may not be triggered soon enough, or more likely, not at all when on an incorrect path,

3. The smaller Δ, not only will $t(l)$ reach the next lower threshold sooner and thereby increasing the probability of the event in 1, but the look-back subroutine will in general be triggered (perhaps unnecessary) with higher probability depending on the noise level, resulting in unnecessary computations,

4. When Δ is too large, the look-back subroutine may be triggered too late, resulting in too many unnecessary move-back computations. This will especially be the case when $t(l)$ is just above the next lower threshold. In fact, if $(p-v)$ is chosen too large, the look-back subroutine may never be triggered when necessary.

Choosing the above parameters incorrectly may result in catastrophic failure in that the algorithm progresses very slowly or not at all, or it may result in too many error events caused by re-merging of the correct and tentative path we are following. In the following example we reduce the values of $(p-v)$ and Δ on the basis of argument 1 and 2 above.

Example 2.a: $(p-v) = 0.20$; $\Delta = 0.20$

See Table 3.4 and Figure 3.5.

Although the look-back routine was triggered, the algorithm still failed to detect the incorrect decision before re-merge.

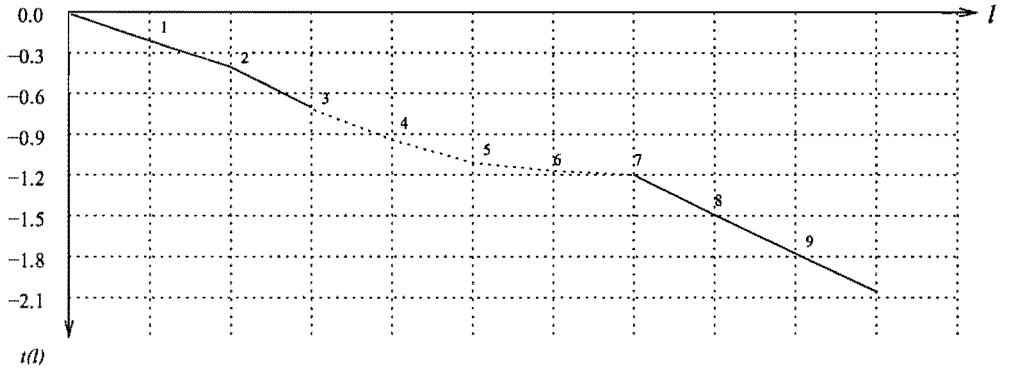


Figure 3.4: Received distance tree for Example 1 ($(p-v)=0.3$, and $\Delta=0.3$)

At	δ	action	$\Delta\lambda_n(x,y)$	$d(l)$	$t(l)$	action
0	0.00	look at 1	0.40	0.10	-0.10	move to 1
1	0.00	look at 2	0.43	0.17	-0.23	move to 2, set $\delta=-0.20$
2	-0.20	look at 3	0.49	0.18	-0.42	move to 3, set $\delta=-0.40$
3*	-0.40	look at 4	0.40	0.28	-0.52	move to 4
4*	-0.40	look at 5	0.38	0.40	-0.60	move to 5, set $\delta=-0.60$
5*	-0.60	look at 6	0.28	0.62	-0.58	move to 4, set $\delta=-0.40$
6*	-0.40	look at 7	0.28	0.62	-0.58	move to 6
7	-0.40	look at 8	0.21	0.91	-0.49	move to 7
8	-0.40	look at 9	0.51	0.90	-0.70	move to 8, set $\delta=-0.60$
9	-0.60	look at 10	0.49	0.91	-0.89	move to 9, set $\delta=-0.80$

Table 3.4: Tree search for Example 2.a ($(p-v)=0.2$, and $\Delta=0.2$)

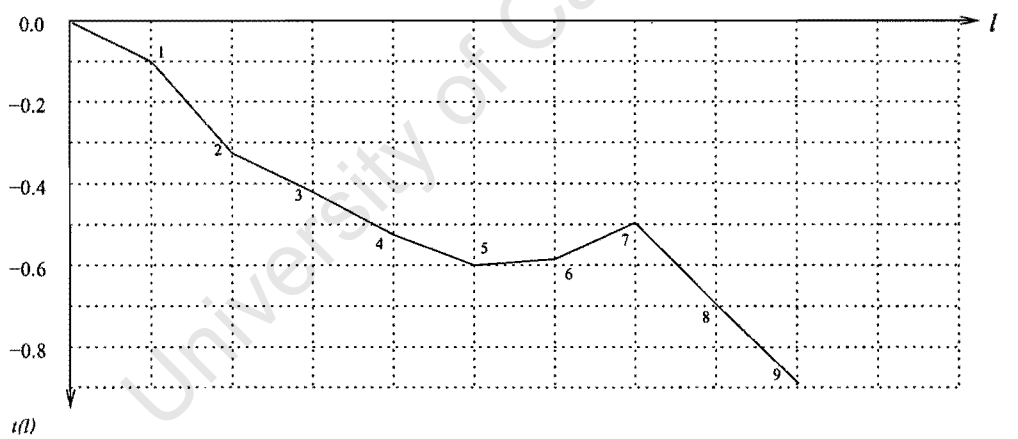


Figure 3.5: Received distance tree for Example 2.a ($(p-v)=0.20$, and $\Delta=0.20$)

Example 2.b: $(p-v)=0.20$, $\Delta=0.20$

Let us repeat the previous example but with the received path slightly changed so as to avoid the early re-merge of the paths. See Figure 3.6, Table 3.5 and Figure 3.7.

Disappointingly, a re-merge occurred sooner! We conclude that the probability of re-merge is not negligent and is a definite drawback of the Fano Algorithm when applied to 4-*ary* CPFSK and very likely, CPM in general.

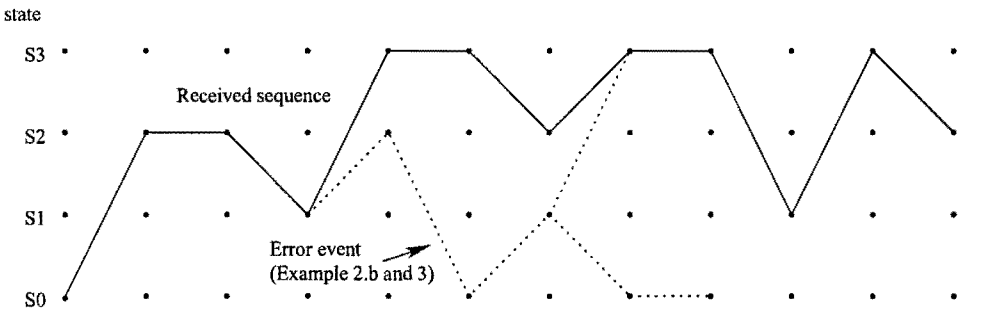


Figure 3.6: Received sequence for Example 2.b and 3

At	δ	action	$\Delta\lambda_n(x,y)$	$d(l)$	$t(l)$	action
0	0.00	look at 1	0.40	0.10	-0.10	move to 1
1	0.00	look at 2	0.43	0.17	-0.23	move to 2, set $\delta = -0.20$
2	-0.20	look at 3	0.49	0.18	-0.42	move to 3, set $\delta = -0.40$
3*	-0.40	look at 4	0.40	0.28	-0.52	move to 4
4*	-0.40	look at 5	0.38	0.40	-0.60	move to 5, set $\delta = -0.60$
5*	-0.60	look at 6	0.28	0.62	-0.58	look at 4, set $\delta = -0.40$
5*	-0.40	look at 6	0.28	0.62	-0.58	move to 6
6*	-0.40	look at 7a	0.41	0.71	-0.69	move to 7a, set $\delta = -0.60$
7a*	-0.60	look at 8a	-0.01	1.22	-0.38	look at 6, set $\delta = -0.40$
7a*	-0.40	look at 8a	-0.01	1.22	-0.38	look at 6, move to 6
6*	-0.40	look at 7b	0.35	0.77	-0.63	move to 7b, set $\delta = -0.60$
7b*	-0.60	look at 8b	0.53	0.74	-0.96	move to 8b, set $\delta = -0.80$

Table 3.5: Tree search for Example 2.b ($(p-v) = 0.2$, and $\Delta = 0.2$)

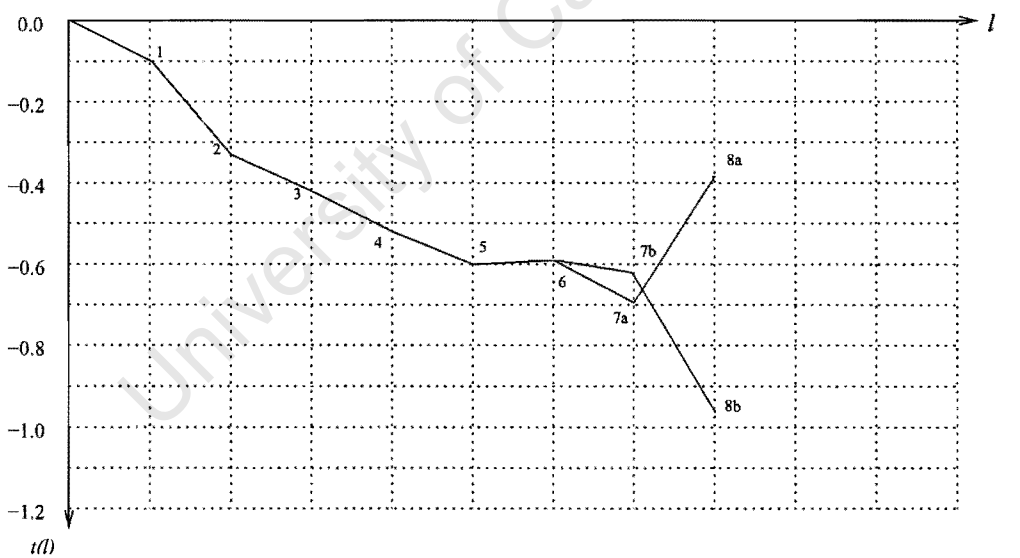


Figure 3.7: Received distance tree for Example 2.b ($(p-v) = 0.20$, and $\Delta = 0.20$)

How then should we select $(p-v)$ and Δ?

From our results thus far and from previous discussion, we know that $(p-v)$ should be as small as possible, but not so small that the look-back routine is triggered falsely due to noise on the first possible "incorrect" decision when $t(l)$ is on or just below the running threshold δ. What is the

Report on corrections to Masters dissertation

Werner F Kleyn

31 May 2002

The following corrections have been done:

1. Lower case 'c' replaced by upper case 'C' where reference is made to specific chapters.
   Applicable page: 13.

2. Lower case 'e' replaced by upper case 'E' where reference is made to specific equations.
   Applicable pages: 14, 17, 18, 19, 23, 24, 25, 26, 35, 52, 54.

3. Lower case 'f' replaced by upper case 'F' where reference is made to specific figures.
   Applicable pages: 14, 18, 19, 21, 22, 26, 29, 31, 32, 33, 35, 36, 38, 47, 48, 49, 52,
   58, 61.

4. Lower case 't' replaced by upper case 'T' where reference is made to specific tables.
   Applicable pages: 26, 29, 32, 33, 36, 51, 54, 58, 60, 61.

The following have been added:

1. A short conclusion chapter (Chapter 5) including summary and a future work section. Table of
   contents were updated.

Signed:

_____ (student) _____ (supervisor)

_____ (date) _____ (date)

smallest $(p - v)$ that will give us satisfactory results? The situation is depicted in Figure 3.8. I assumed that a "very likely" first error event will be due to a supposedly best metric (p) that is, due to noise, about halfway between p and v_{max}, that is, $((p - v_{max})/2 = 0.09)$, and a second best metric (v_{max}) whose value is just above those of the supposedly best metric. As from the interval following the first incorrect decision, we follow a worst case path defined by (from Equation 3.2)

$$t(l) = (p - v_{max})l - (p - v)l \tag{3.3}$$
$$= 0.18l - (p - v)l \tag{3.4}$$

Setting $t(l) = \Delta - 0.09$ (see Figure 3.9) and solving for l gives us a relationship between $(p - v)$, Δ and l:

$$l = \frac{\Delta - 0.09}{0.18 - (p - v)} \tag{3.5}$$

where $l + 1$ is now the maximum number of intervals we spend on an incorrect path before the look-back algorithm is triggered (unless we happen to cross the correct path before the look-back algorithm is triggered). The relationship among Δ, l and $(p - v)$ is shown in Figure 3.9. Since Equation 3.5 describes a worst case situation, the "average" value for l is more likely to follow the curve marked "average" on the graph. For $(p - v)$ values greater than $\cong 0.18$, the probability exist that the look-back algorithm will not be triggered when on an incorrect path. This will happen if we remain on a $(p - v_{max})$ path after the first incorrect decision.

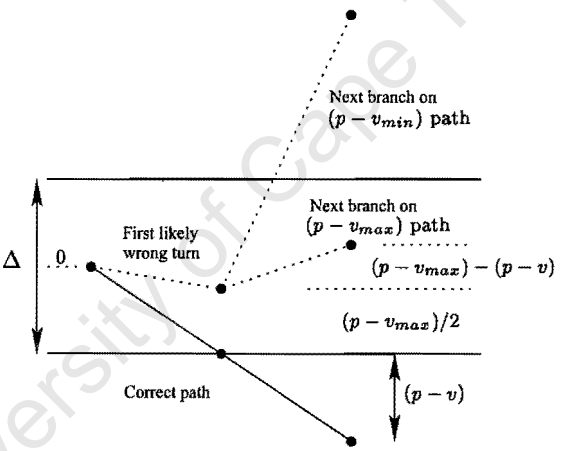


Figure 3.8: Diagrams used for determination of optimum $(p - v)$ and Δ_{max}

Selecting $(p - v) \cong (p - v_{ave}) = 0.12$, and limiting the number of intervals we spend on an incorrect path to two intervals, we have from Figure 3.9 a $\Delta \cong 0.18$. However, noise was not taken into account in this reasoning. Noise will place an lower limit on the value of Δ that may be used. A higher noise level will require an increase in Δ. The Fano algorithm will become increasingly ineffective for higher noise levels, since it will force us to increase Δ to values that will delay triggering of the look-back algorithm to beyond acceptable depths. Increasing noise levels will also require $(p - v)$ to be increased to prevent a false triggering of the look-back subroutine, but then the triggering of the look-back routine may be happen too late, as stated above, and the likelihood of re-merge of the received sequence path with the tentative path will be high.

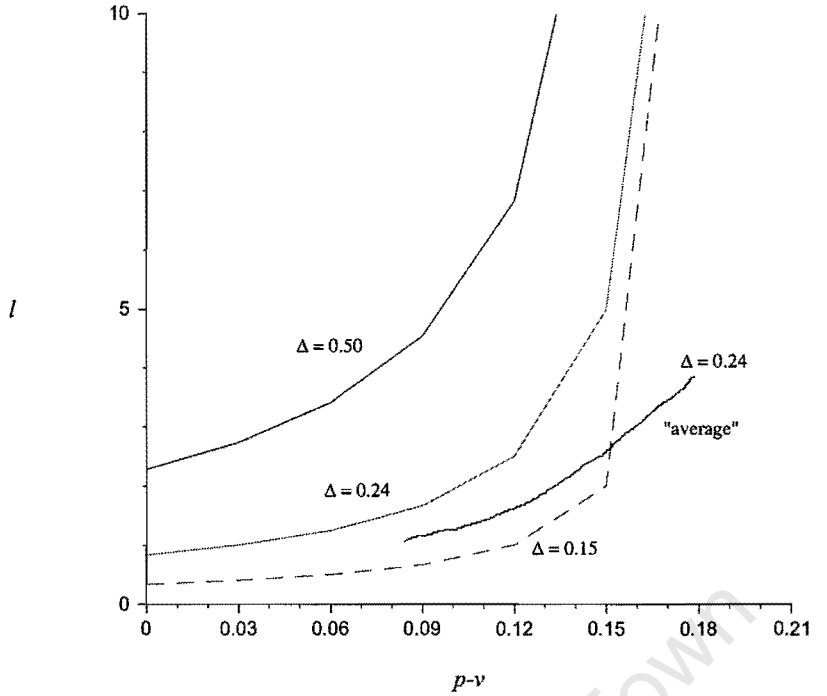


Figure 3.9: Relationship among Δ, $(p-v)$ and l.

Example 3: $(p-v) = 0.12$; $\Delta = 0.18$

In this example, we will let $(p-v) = 0.12$ and $\Delta = 0.18$. See Table 3.6 and Figure 3.10.

At	δ	action	$\Delta\lambda_n(x,y)$	$d(l)$	$t(l)$	action
0	0.00	look at 1	0.40	0.10	-0.02	move to 1
1	0.00	look at 2	0.43	0.17	-0.07	move to 2
2	0.00	look at 3	0.49	0.18	-0.18	move to 3, set $\delta = -0.18$
3*	-0.18	look at 4a	0.40	0.28	-0.20	move to 4a
4a*	-0.18	look at 5a	0.38	0.40	-0.20	move to 5a
5a*	-0.18	look at 6a	0.28	0.62	-0.10	look at 4a, move to 4a
4a*	-0.18	look at 5b	0.30	0.48	-0.12	look at 3, move to 3
3	-0.18	look at 4b	0.38	0.30	-0.18	move to 4b
4b	-0.18	look at 5c	0.54	0.26	-0.34	move to 5c
5c	-0.18	look at 6b	0.48	0.28	-0.44	move to 6b
6b	-0.36	look at 7	0.58	0.20	-0.64	move to 7

Table 3.6: Tree search for Example 3 ($(p-v) = 0.12$, and $\Delta = 0.18$)

It seems that the likelihood of the two paths re-merging is not negligent. However, this time the incorrect decision was detected despite the re-merge. However, noise almost cause the algorithm to trace back further than was necessary on its second visit to node 3.

3.2.1 Simulations

Simulations applying the Fano algorithm to decoding 4-*ary* CPFSK were performed for the following combinations of $(p-v)$, Δ and E_b/N_o :

$E_b/N_o =$ 16.00, 13.10, 11.94, 10.00 and 8.42 dB

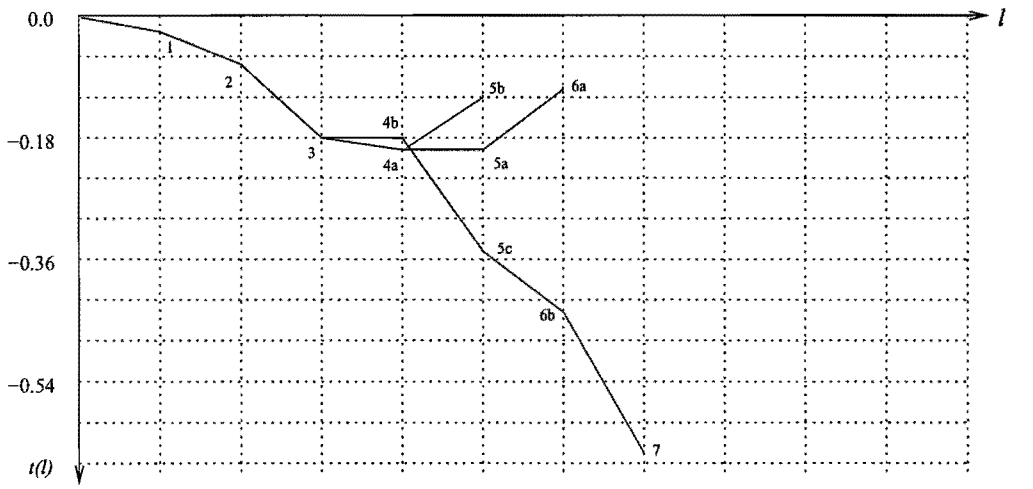


Figure 3.10: Received distance tree for Example 3 ($(p-v) = 0.12$ and $\Delta = 0.18$)

$(p-v) =$ 0.06, 0.09, 0.12, 0.15 and 0.18

$\Delta =$ 0.06, 0.09, ... 0.27 and 0.30.

The following data was recorded:

1. Number of symbol errors
2. Number of burst errors
3. Total number of look-backs
4. Total number of move-back steps
5. Distribution of instantaneous move-backs steps (the number of times that we moved back a specific number of steps)
6. Distribution of maximum depth reached in moving back at each new set of metrics investigated

The Fano decoder was simulated on a pseudo-random sequence of 200000 input bits, except for the tests where $E_b/N_o = 16.00\,dB$ and $E_b/N_o = 13.10\,dB$. At these signal-to-noise ratios 10 million and 1 million input bits were used, respectively. The degree of the pseudo-random generator polynomial used was 18. One might question whether this procedure is appropriate, but the results obtained nevertheless show valuable information as to the suitability of the Fano algorithm for decoding schemes with variable distance between signal sets. A more appropriate method would have been to run the simulations until i.e. 50 symbol errors were generated. In the graphs that follows, all the results were normalized to 100000 input symbols (200000 bits). For these tests a simulation step size of $T_s/20$ seconds were used. The Fano algorithm's memory were limited to 12, that is, all metrics were saved to a depth of 12 and then discarded. This means that whenever the move-back algorithm tried to move back more than 12 intervals, the algorithm reacted as if it has reached the origin (see the flow chart in Figure 3.2). These attempts to exceed a depth of 12 backward moves are recorded in the bar charts that follow, as a '>12' tick. An observation period (number of symbol intervals observed before making decision about the $1st$ symbol transmitted) equal to 12 was used.

Please note that the scales on the vertical axis of the following graphs may differ form graph to graph. The BAR charts show depth of backward steps performed. The instantaneous values refer to backward steps starting at any point in the trellis whereas the maximum values refer to the maximum depth reached starting at the point where a new set of metrics has just been read in.

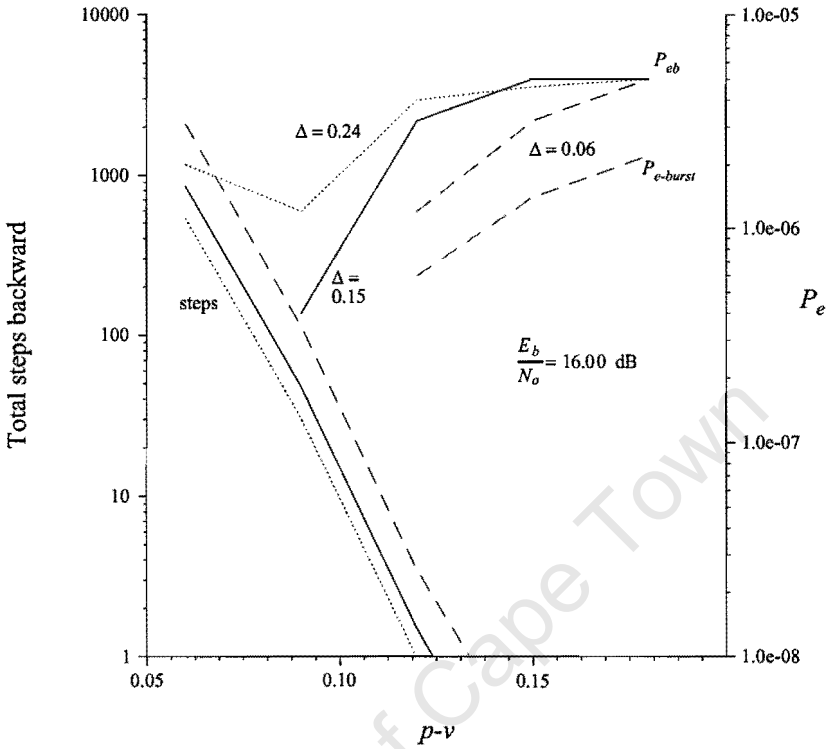


Figure 3.11: Effect of Δ and $(p-v)$ on P_e and number of backward steps, $E_b/N_o = 16.00$ dB

3.3 Performance Analysis and Summary

The Fano decoder approaches the performance of the optimum receiver for high E_b/N_o. For an E_b/N_o of 16.00 dB, no errors occurred for 10 million bits received. The 16.00 dB data points in Figure 3.25 were simply added to show the trend. At an $E_b/N_o = 11.94$ dB, and $p - v = 0.09$, $\Delta = 0.06$, the total number of backward steps amounts to about 2 percent of symbols received. The number of look-backs amounts to 9.3 percent of symbols received. For $p - v = 0.06$, $\Delta = 0.06$, these figures are 11 and 32.7 percent respectively. Figure 3.26 shows the move-backs and look-backs as a percentage of the number of symbols received for the most promising combinations of $(p - v)$ and Δ. Table 3.7 lists the attempts to move back farther than 12 symbol intervals. Nearly all the errors occurred in bursts of length equal to 2. Table 3.8 demonstrates the chain effect on the decoder due to an increase in $(p - v)$. This table also explains the results obtained in terms of Figure 3.9.

The performance degradation with respect to the optimum receiver in terms of bit error rate for lower signal-to-noise ratios may be attributed to paths that re-merge before a look-back trigger happened. The phenomenon of re-merging paths remains the major performance related disadvantage of the Fano algorithm. However, for systems with sufficient signal-to-noise margin available,

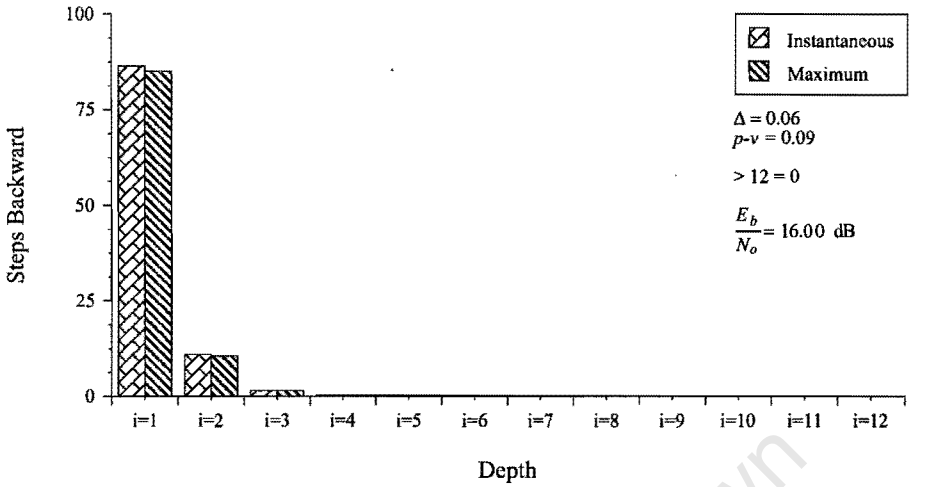


Figure 3.12: Distribution of instantaneous and maximum backward steps for $E_b/N_o = 16.00$ dB

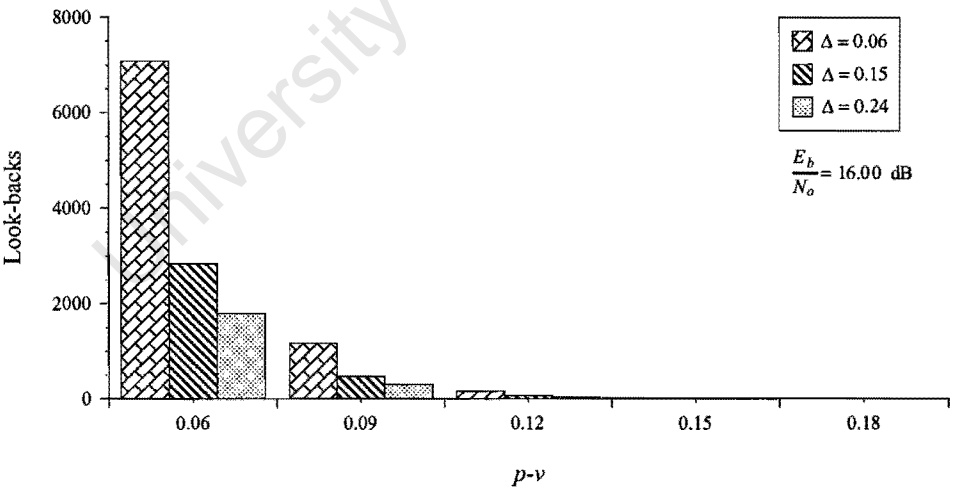
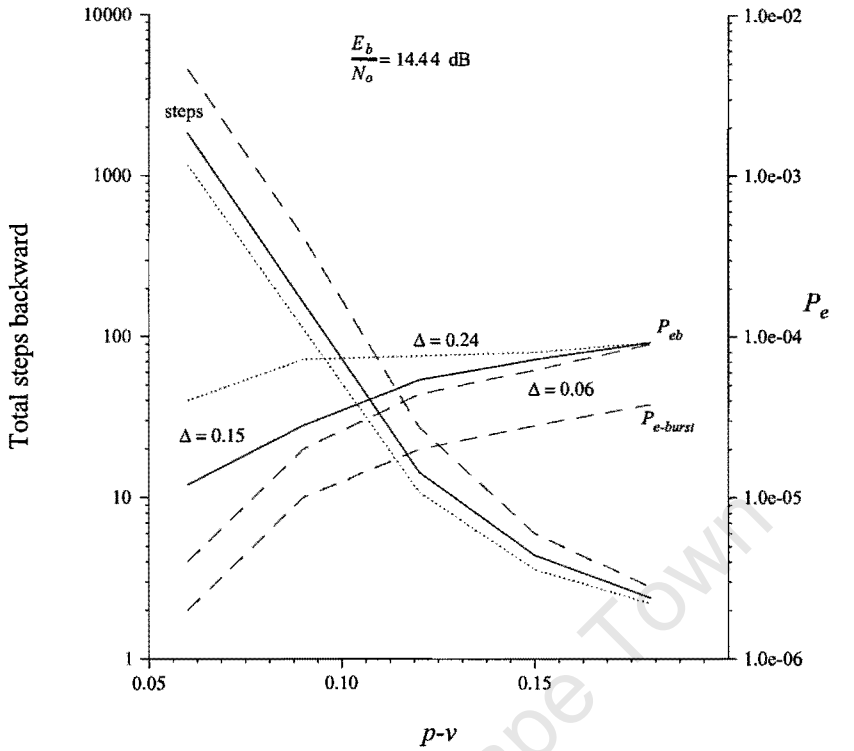


Figure 3.13: Distribution of look-backs for $E_b/N_o = 16.00$ dB



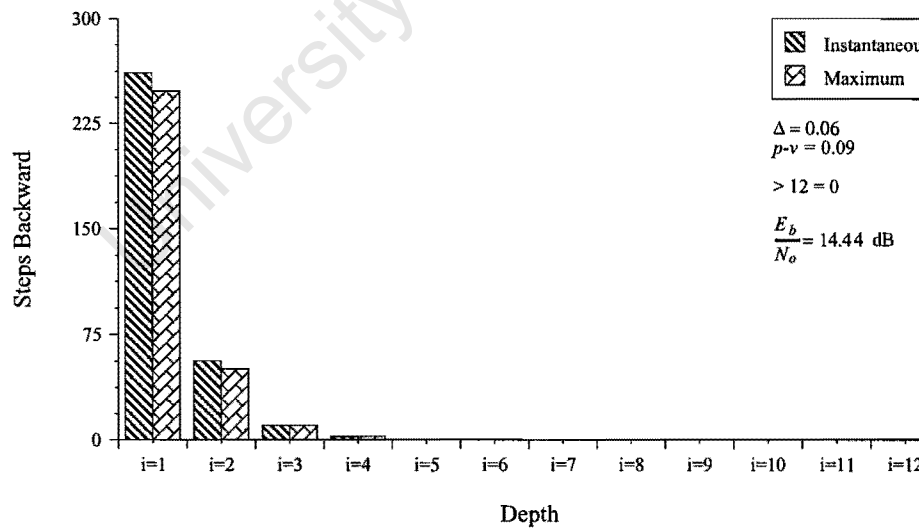


Figure 3.14: Distribution of instantaneous and maximum backward steps for $E_b/N_o = 14.44$ dB

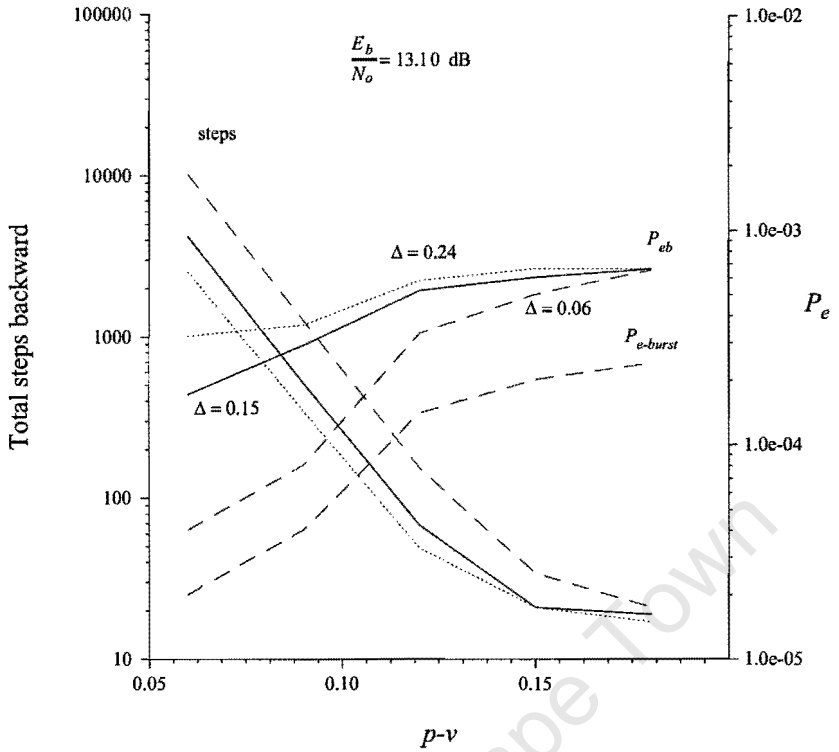


Figure 3.15: Effect of Δ and $(p - v)$ on P_e and number of backward steps, $E_b/N_o = 13.10$ dB

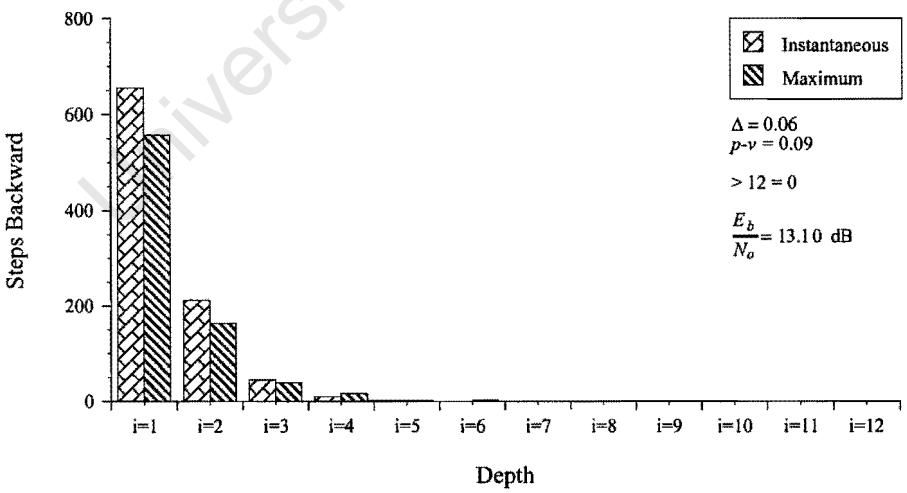


Figure 3.16: Distribution of instantaneous and maximum backward steps for $E_b/N_o = 13.10$ dB

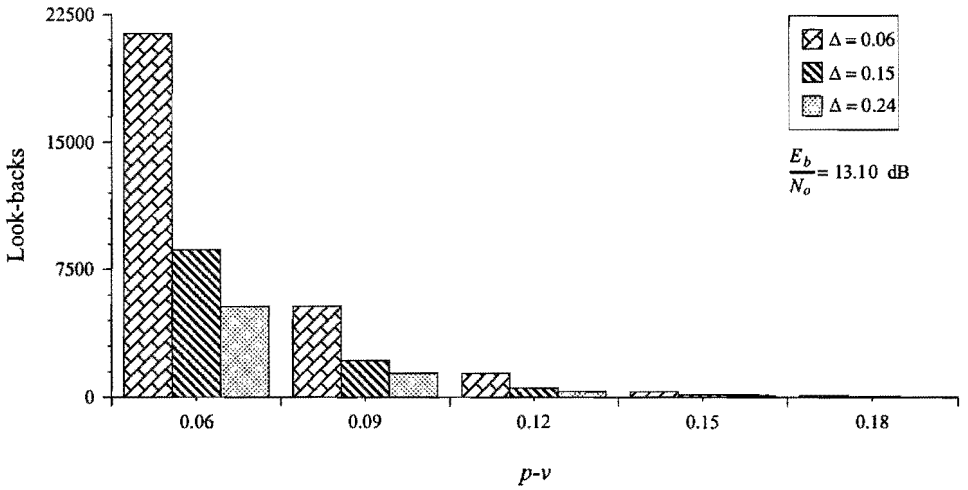


Figure 3.17: Distribution of look-backs for $E_b/N_o = 13.10$ dB

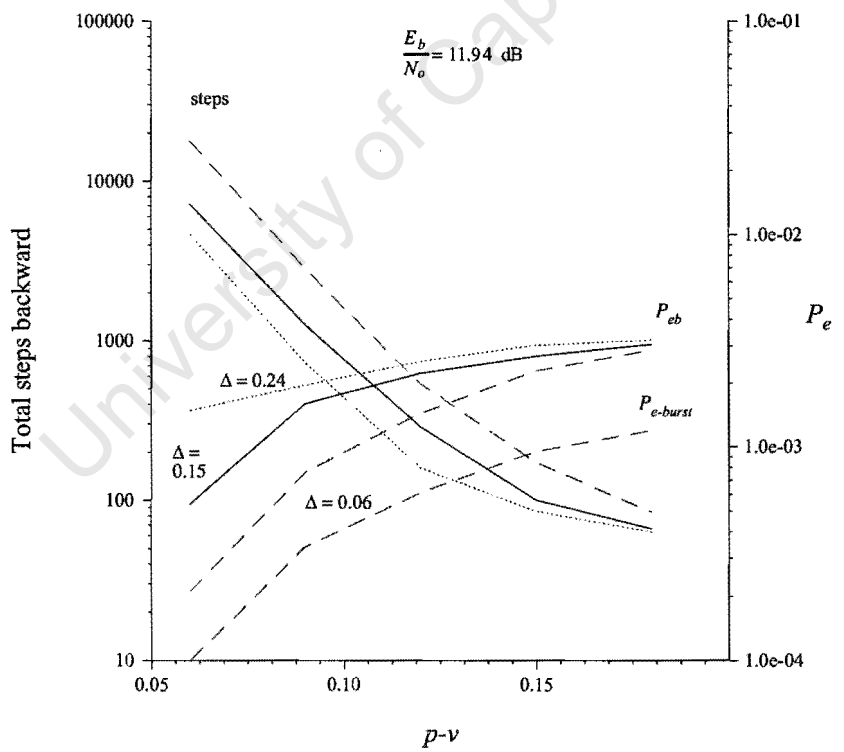


Figure 3.18: Effect of Δ and $(p - v)$ on P_e and number of backward steps, $E_b/N_o = 11.94$ dB

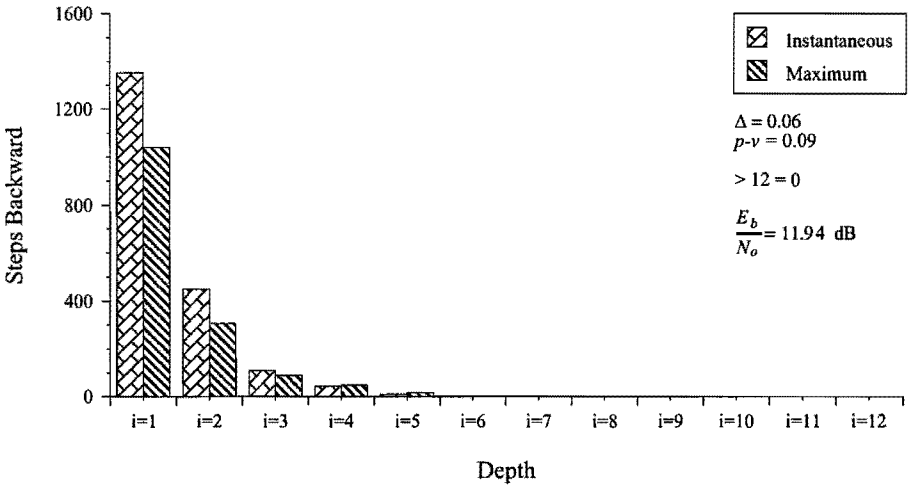


Figure 3.19: Distribution of instantaneous and maximum backward steps for $E_b/N_o = 11.94$ dB

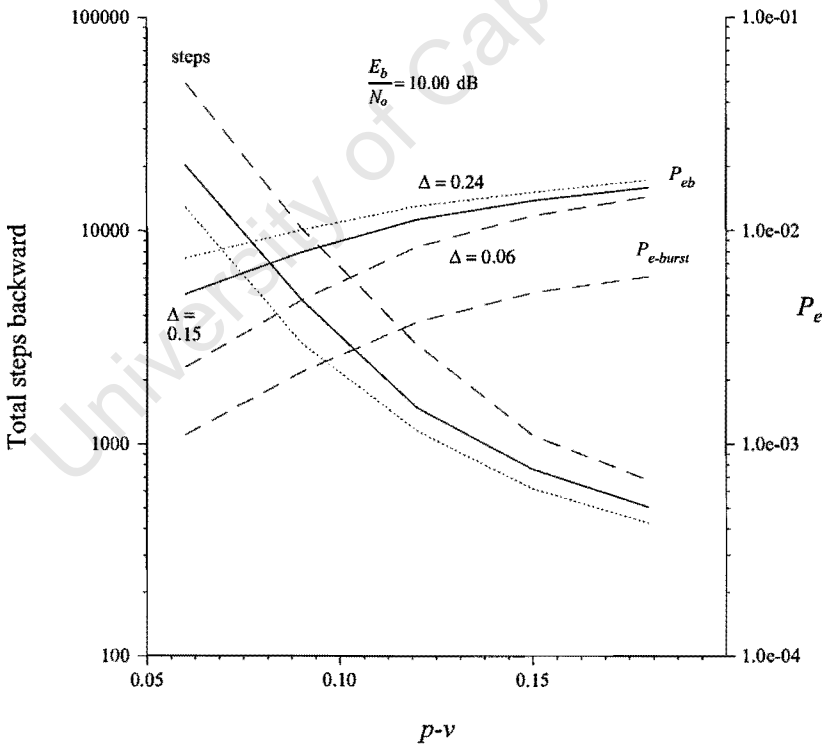


Figure 3.20: Effect of Δ and $(p - v)$ on P_e and number of backward steps, $E_b/N_o = 10.00$ dB

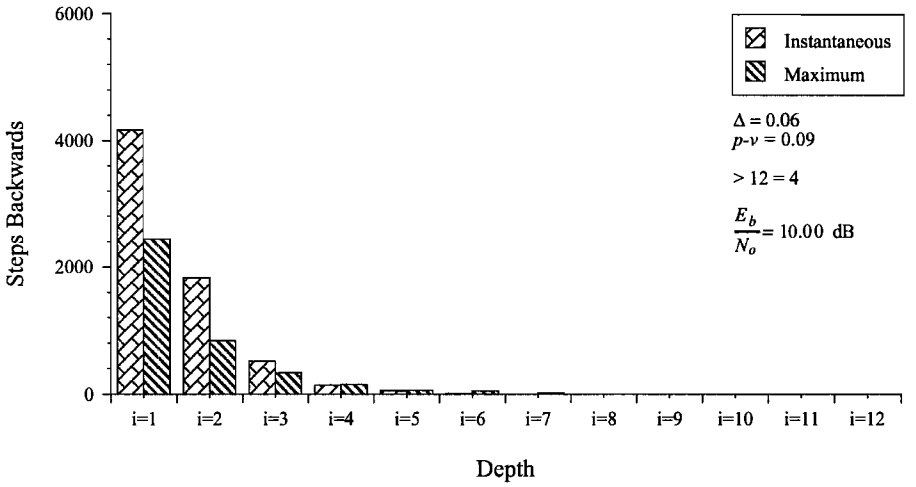


Figure 3.21: Distribution of instantaneous and maximum backward steps for $E_b/N_o = 10.00$ dB

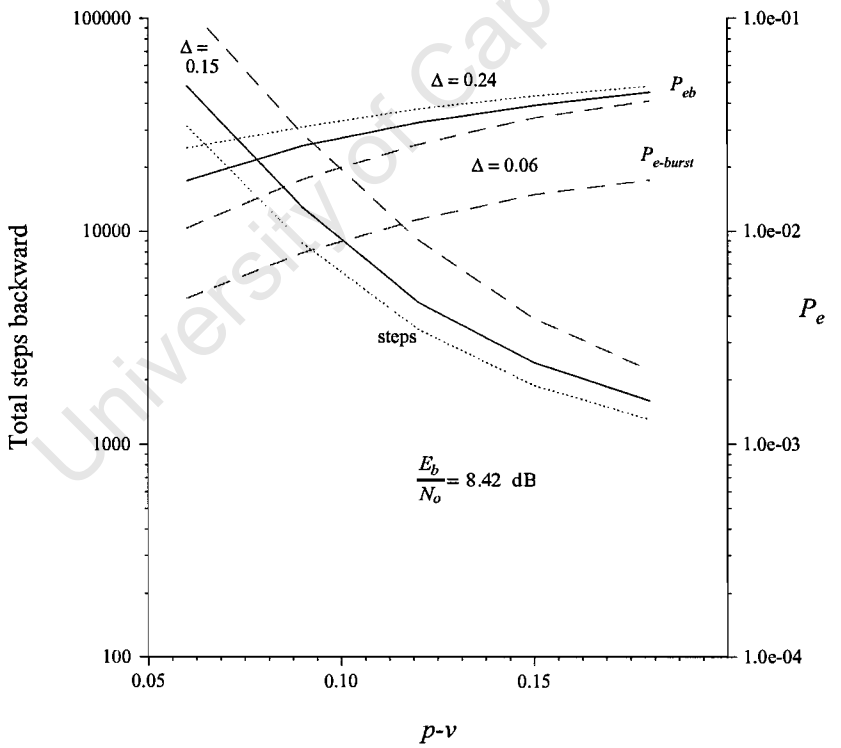


Figure 3.22: Effect of Δ and $(p - v)$ on P_e and number of backward steps, $E_b/N_o = 8.42$ dB

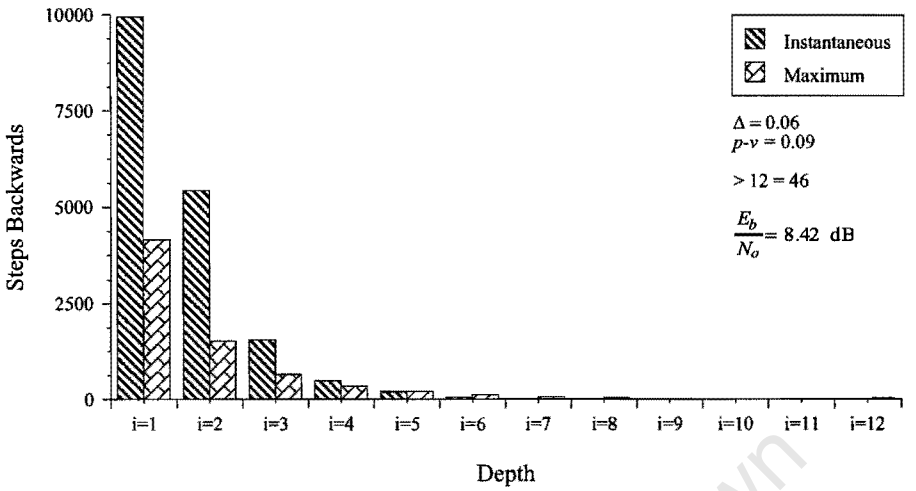


Figure 3.23: Distribution of instantaneous and maximum backward steps for $E_b/N_o = 8.42$ dB

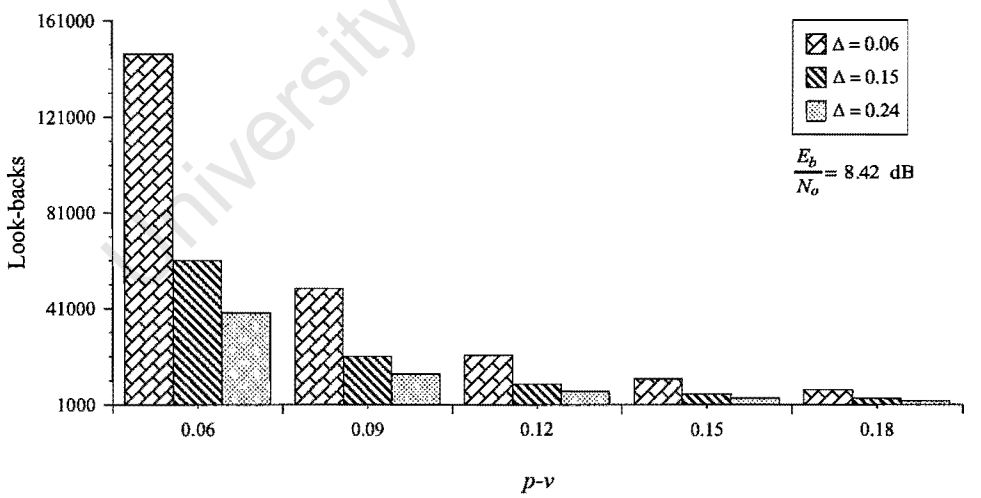


Figure 3.24: Distribution of look-backs for $E_b/N_o = 8.42$ dB

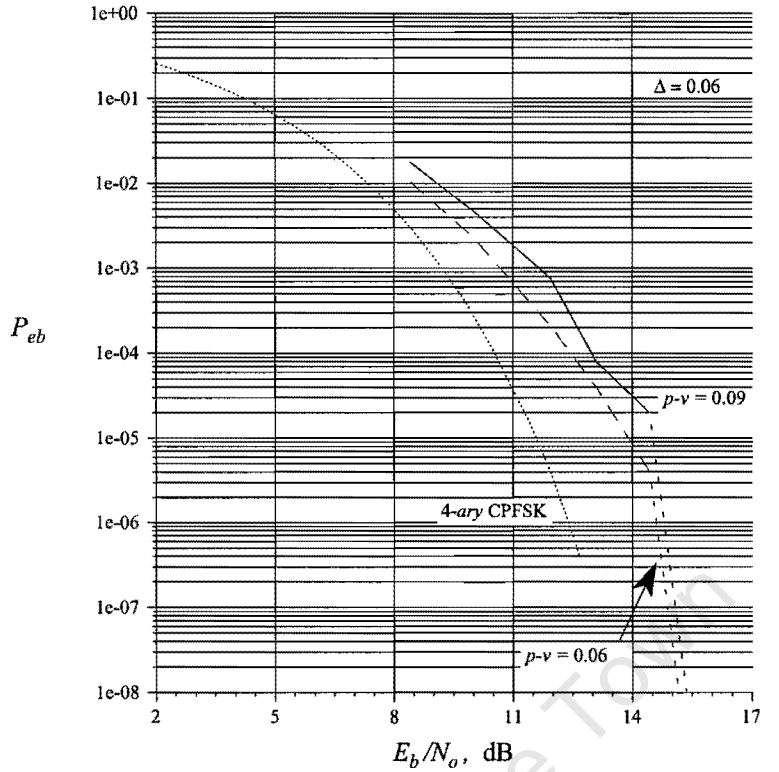


Figure 3.25: Fano decoder BER performance

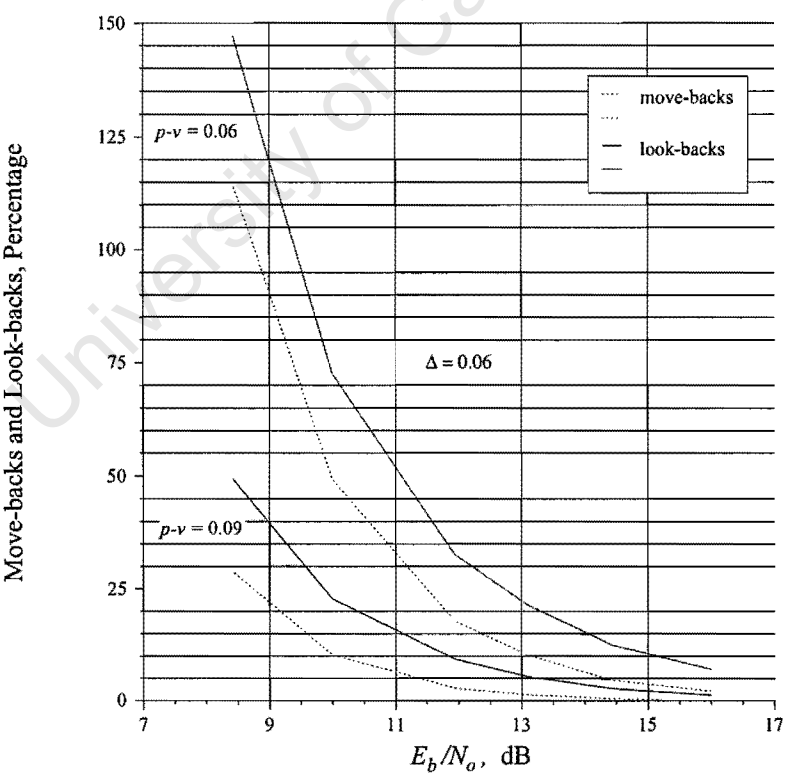


Figure 3.26: Move-backs and look-backs as percentage of symbols received

performance is acceptable. The problem of re-merging paths may be reduced by using schemes with a longer minimum distance error event (schemes with larger constraint length for which applying the Fano decoder would be more appropriate anyway, since exponentially increasing complexity of the Viterbi decoder with increase in constraint length was in the first place the motivation for using the Fano algorithm). Future research on the subject of Fano decoders should therefore focus on a scheme with larger constraint length.

The Fano algorithm is substantially more complex to implement in software than the Viterbi algorithm. This ratio of complexity (about 3 to 4) between the software implementations of the Fano algorithm and the Viterbi algorithm can be expected to project to their respective hardware (VLSI) implementations. On the other hand, whereas the memory required for implementation of the Viterbi algorithm increases exponentially with constraint length of the scheme, it remains a constant for the Fano decoder. Further investigation is required into the VLSI structure for the Fano decoder.

E_b/N_o , dB	$p-v$	
	0.06	0.09
16.00	0	0
14.44	2	0
13.10	26	0
11.94	63	0
10.00	280	4
8.42	747	46

Table 3.7: Number of attempts to move back deeper than 12 symbol intervals for $\Delta = 0.06$

$(p-v)\uparrow$	$\downarrow$ look-backs $\uparrow l$	$\downarrow$ move-backs $\uparrow$ prob (remerge) $\uparrow$ prob (move-back farther than 12)	$\uparrow$ errors

Table 3.8: Effect of the increase of $(p-v)$, Δ constant

Chapter 4

Neural Network-based Receivers

4.1 Introduction

Contrary to conventional serial computers, neural net models explore many competing hypotheses simultaneously by using parallel nets composed of many computational elements connected by links having variable weights [15]. These computational links are also called neurons or perceptrons. The neural net stores experimental knowledge through a learning process as weights on the input nodes to the neurons. Since finding the optimum path through a phase trellis can be related to the problem of exploring many hypotheses simultaneously, we may postulate the artificial neural net it to be a good trellis decoder. However, research [32, 18] done on the neural net's ability as trellis decoder has produced very disappointing results. Most of these research focussed on using very complex neural net structures. I will explore the subject by concentrating on very simple structures.

The first part of this chapter provides an overview of neural nets in general. We will classify specific neural nets according to the type of problem it can solve and then discuss training algorithms. In the second part we apply this knowledge to trellis decoding for MSK and 4-*ary* CPFSK.

4.1.1 The Single Layer Perceptron

The perceptron (see Figure 4.1), or neuron, as it is sometimes called, is the fundamental processing unit of the neural network. The decision regions formed by perceptrons are similar to those formed by the maximum likelihood Gaussian classifier [15]. The perceptron decides whether an input belongs to one of two classes. This single node computes a weighted sum of the inputs, subtract a threshold (θ) and passes the result through a non-linear function (also called the activation function) such that the output is either $+1$ or -1. The activation function's output are limited in range and therefore called a squashing function.

The general formula for the activation of a perceptron (unit) within a network is:

$$a_j(t+1) = f_{act}\left(\sum_i w_{ij}x_i(t) - \theta_j\right) \tag{4.1}$$

where:

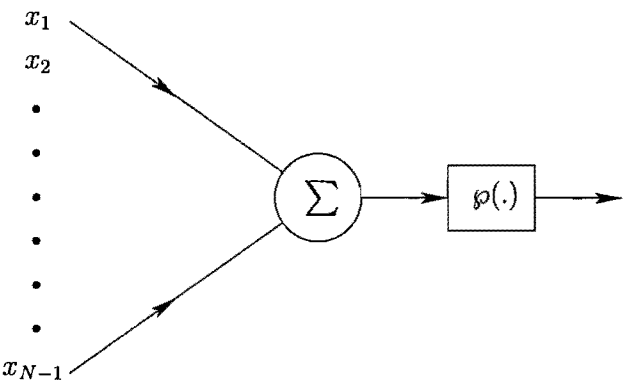


Figure 4.1: Model of the Neuron

w_{ij}	weight of the link from node i to node j
$x_i(t)$	output of node i in step t
$a_j(t)$	activation of unit j in step t
θ_j	threshold (bias) of unit j
j	index of some unit in the net
i	index of a predecessor of the unit j

The term *node* as used here refers to either an input node or a perceptron. The notation I use here is similar to that used by SNNS [28]. The activation function f_{act} computes the unit output by simply summing over all weighted inputs (or activations if predecessor units exist) and then squashing the result. Examples of squashing functions are the hyperbolic tangent, sigmoid and signum functions. The equation describing the sigmoid activation function is:

$$f_{act}(\alpha) = \frac{1}{1 + e^{-\frac{\alpha}{T}}} \tag{4.2}$$

where T denotes the "temperature" of the sigmoid function as shown in Figure 4.2.

4.1.2 Specification of Neural Networks

A Neural Network consists of perceptrons (units) with directed, and weighted links between them. Each perceptron receives an input that is computed from the weighted outputs of prior units with connections leading to this unit.

Neural Networks are specified by the net topology, perceptron (node) characteristics, and training or learning rules. Various neural net models exist and each net can be related to specific pattern classification and clustering problems that are normally implemented on serial machines [15]. The learning rules specify an initial set of weights and indicate how weights should be adapted. Nets can be divided between those trained with supervision and those trained without supervision. Most traditional statistical classifiers, such as Gaussian classifiers, are trained with supervision.

The feed-forward neural network is a layered network consisting of an input layer, hidden layer(s) and an output layer. For example, a feed-forward net having eight nodes as input layer, four as *1st* hidden layer, four as *2nd* hidden layer and two as output layer is called a 8-4-4-2 multilayer feed-forward network. When every node in every layer is connected to every node in the adjacent layer it is called a fully connected feed-forward layer. These multi-layer perceptron structures overcome many of the limitations of single-layer perceptrons. A single-layer perceptron

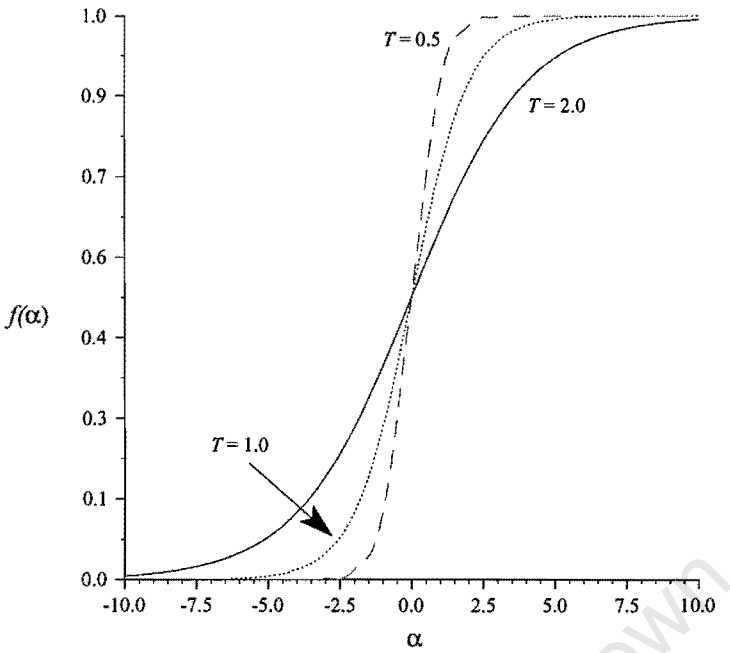


Figure 4.2: The sigmoid squashing function

forms half-plane decision regions. A two-layer network can form any, possibly unbounded, convex region in the space spanned by the inputs [15]. The number of nodes must be enough to form a decision region that is as complex as is required by a given problem [15]. On the other hand, having too many nodes will result in unreliable estimation of weights.

A three-layer network may form arbitrary decision regions, such as those formed using mixture distributions and nearest-neighbor classifiers [7]. For this reason I will concentrate on using simple three-layer network structures for decoding the CPFSK signal.

When sigmoidal nonlinearities are used instead of hard limiting nonlinearities, decision regions are typically bounded by smooth curves instead of straight line segments [7].

4.1.3 Conventional Pattern Recognition versus Trellis Decoding

Neural network models are known to have greatest potential in areas such as speech and image recognition where many hypothesis are explored in parallel and where high computation rates are required [15]. The identification of one of 36 pattern in a 12 x 10 binary image involves the consideration of 36 out of 2^{120} possibilities [33]. In contrast, of the M^{D+1} possible paths in the excess phase trellis of 4-*ary* CPFSK, up to M^{D+1} may be valid, where D is the observation window. That means that the information to be decoded from a phase trellis is far more extensive than that contained in most pattern recognition problems.

So what type of Neural Network model is most appropriate for solving our problem? We need to find a neural net that performs the same function as the CPFSK decoding algorithm. The decision regions required by any classification algorithm can be generated in a straight-forward manner by three-layer feed-forward neural networks [15]. For our application, decision regions are bounded by straight line segments, and according to [7], the use of hard limiting nonlinearities as squashing functions should then be sufficient. However, we will experiment with sigmoidal nonlinearities of different temperatures, some of which approach the hard limiting nonlinearity.

4.1.4 Neural Network Receiver

The input to the neural network consists of sampled in-phase and quadrature data recorded over an observation period of 2 ($D = 2$) symbol intervals. This is the optimum observation window for decoding MSK, and as we have seen in section 2.6.1, result in negligent loss in accuracy for 4-*ary* CPFSK. The sampling rate is equal to $4/T_s$ Hz. Each input vector then has 16 elements.

For MSK, the neural net is required to classify each input vector as either class A or a class B, where class A corresponds to a symbol 0 transmitted during the first interval of the observation window and class B corresponds to a 1 transmitted. For 4-*ary* CPFSK, the neural net classify each input vector as class A, B, C or D, each corresponding to a symbol (U_n) 0, 1, 2 or 3 respectively, transmitted in the $1st$ interval of the observation window.

All the simulation code were written in C. See section D.1 for a description of these functions.

4.1.5 Training of Neural Networks

The method of training depends on the proposed function of the neural net. Training the neural net means presenting it with input sets and the corresponding outputs and then systematically adjusting the synaptic weights of the neurons until the network is able to classify each of the inputs correctly. An input vector and its associated output are called an *exemplar*.

The back-propagation learning algorithm is a recursive gradient descent algorithm designed to minimize the mean squared error between the actual output of a multilayer feed-forward network and the desired output. The synaptic weights are systematically updated and outputs tested until the minimum squared error between the example outputs and neural net outputs are achieved [15]. It cannot be proven that a training algorithm converge as with single-layer perceptrons, but they have been shown to be successful for many problems of interest.

One of the problems with training of neural nets is the occurrence of local minima. Various algorithms have been derived to overcome this problem. Examples of these are guided random searches (genetic algorithms) based on an Least Mean Squares criterion. I have used back-propagation to train the neural networks and tried to minimize the problem of local minima by using training sets based on different initial conditions and algorithm parameters. The back-propagation algorithm as applied here is summarized in appendix A.

The algorithm parameters and initial conditions listed in Table 4.1 were used for training the networks. See Appendix A for a detailed description of these parameters. In the table, T_x refers to the temperature of the sigmoid function. I suspected the algorithm to converge faster for lower sigmoid temperature and therefore used fewer training cycles (K) at lower temperatures. η is a gain term describing the learning rate of the algorithm, and a, b, and c are parameters describing the amplitude of the randomly assigned initial weights and offsets. According to Anderson *et al* [5], these random numbers tend to clump the weight space resulting in the weight space being unevenly covered. They suggest the use of quasi-random numbers to obtain even coverage.

We will use the network as a classifier which means that all desired outputs are set to zero except for that corresponding to the class the input is from. The input vectors are presented cyclically until weights stabilize. Normally one would use a convergence criteria such as a minimum squared error (MSE) between the desired outputs and the neural net outputs to establish when sufficient training is achieved. In the simulations I have seen neural nets corresponding to moderate MSE performing better than those with very small MSE. Each set of weights produced by the training sets in Table 4.1 were verified to produce a moderate to very small MSE.

For MSK, using an observation window equal to two ($D = 2$), the exemplars were associated with two (P) major classes: class A and class B. Class A is associated with a symbol $U_0 = 0$

transmitted in the first interval, class B is associated with $U_0 = 1$ transmitted in the first interval. Likewise for 4-*ary* CPFSK, the exemplars were associated with four major classes: A ($U_0 = 0$), B ($U_0 = 1$), C ($U_0 = 2$) and D ($U_0 = 3$) where U_0 refers to the symbol transmitted in the first interval of the observation period. The number of exemplars, assuming coherent demodulation, are $PN^{(D-1)}$, where P is the number of baseband symbols (U_n), and N is the number of possible signals per symbol interval for the modulation scheme. The input vectors were generated using Equation 2.23. Each major class has $N^{(D-1)}$ associated input vectors with half of them being the negative of the other half.

Training Set	η	a	b	c	$K(x = 0.5)$	$K(x = 1.0)$	$K(x = 1.5)$
$1T_x$	0.5	0.4	0.4	0.2	280	375	500
$2T_x$	0.5	1.0	1.0	0.2	280	375	500
$3T_x$	0.2	1.0	1.0	0.2	280	375	500
$4T_x$	0.2	1.0	1.0	0.2	800	1250	1800
$5T_x$	0.2	2.5	2.5	0.2	280	375	500
$6T_x$	0.5	1.0	1.0	0.2	800	1250	1800

Table 4.1: Training parameters (unless otherwise stated) used for MSK and 4-*ary* CPFSK decoding

4.1.6 A simple case: FFSK

The perceptron forms two decision regions separated by a hyper-plane [15]. These regions are similar to those formed by maximum likelihood Gaussian classifiers. We can therefore use the perceptron to optimally decode FFSK.

Figure 4.3 shows the P_{eb} results for the perceptron decoding of FFSK. The observation window size is $D = 1$, and the sampling rate $4/T_s$ Hz. The four samples form the input vector to the perceptron. The training procedure used is presented in appendix B. Figure 4.3 shows that the perceptron does indeed form decision regions that are similar to those formed by maximum likelihood Gaussian classifiers.

4.2 Neural Network Decoding applied to MSK and 4-*ary* CPFSK

Experiment 1: 8-4-4 Network for MSK

Figure 4.4 shows our initial neural classifier network for decoding MSK.

The circles in Figure 4.4 represent the perceptrons of Figure 4.1. Figure 4.5 shows the prototype receiver structure for MSK. Table 4.2 shows the exemplars used for this experiment. The subscripts for parameters d in the table refer to the node numbers in Figure 4.4. The $1st$ and $2nd$ interval $S_{u,\sigma}$'s were generated as in section 2.5.3 and sampled at a rate of $4/T_s$ Hz to provide the input vector.

Figure 4.6 shows the results obtained for training sets $1T_{1.0}$, $2T_{1.0}$, $3T_{1.0}$, $4T_{1.0}$ and $5T_{1.0}$.

Experiment 2: 16-4-4 network for MSK

Figures 4.7 and 4.8 show the neural net receiver used for this experiment.

The weights for training set $1T_{0.5}$ did not converge when applied to MSK. Tables 4.4 and 4.5 list the bit errors as a function of E_b/N_o, sigmoid temperature and training set used. The best and worst results are also shown in Figure 4.9. The best results were obtained for set $3T_{1.0}$, with

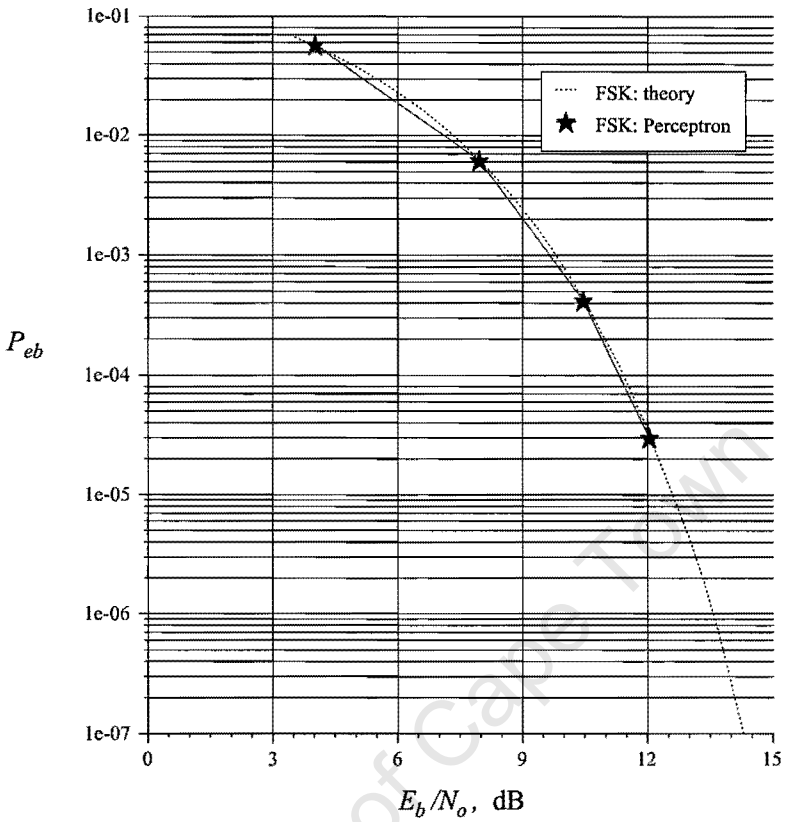


Figure 4.3: Perceptron BER results for FFSK

Interval > class	$1st$	$2nd$	output node setting
A_0	$S_{0,0}$	$S_{0,0}$	$d_i = 0$, $d_{12} = 1$
A_1	$S_{0,0}$	$S_{1,0}$	$d_i = 0$, $d_{12} = 1$
A_2	$S_{0,1}$	$S_{0,1}$	$d_i = 0$, $d_{14} = 1$
A_3	$S_{0,1}$	$S_{1,1}$	$d_i = 0$, $d_{14} = 1$
B_0	$S_{1,0}$	$S_{0,1}$	$d_i = 0$, $d_{13} = 1$
B_1	$S_{1,0}$	$S_{1,1}$	$d_i = 0$, $d_{13} = 1$
B_2	$S_{1,1}$	$S_{0,0}$	$d_i = 0$, $d_{15} = 1$
B_3	$S_{1,1}$	$S_{1,0}$	$d_i = 0$, $d_{15} = 1$

Table 4.2: Exemplars for MSK training, experiment 1

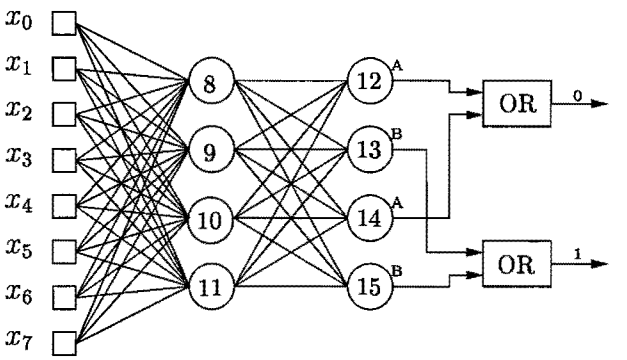


Figure 4.4: 8-4-4 Feedforward Network for MSK

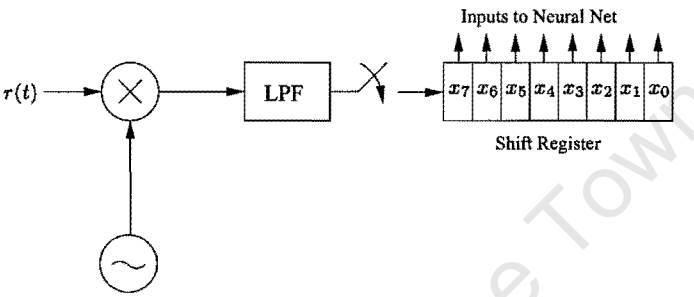


Figure 4.5: Neural Network MSK Receiver

most of the other results very close to the best. It is interesting to note that among the minimum squared error (MSE - see Equation 4.3) results for sigmoid temperature equal to 1.0, set $3T_{1.0}$ has the largest MSE. At a $P_{eb} = 10^{-6}$, this set produced a performance of about 2 dB below the optimum. The weights for set $5T_x$ resulted in a somewhat higher MSE (see Table 4.3) than the rest (except for $3T_{1.0}$) and may explain the poor results for this training set. This may be explained on the basis of the large initial weights used for set 5 which may have led to a higher than normal local minimum in the weight space. The MSE is defined as [32]:

$$MSE = \frac{1}{N}\sum_i (d_i - y_i)^2 \tag{4.3}$$

where d_i and y_i are the desired and actual output for the ith exemplar.

Better results were obtained for sigmoid temperature 1.0 and 1.5 than for temperature 0.5. This is despite the fact that the MSEs for temperature 0.5 are lower than those of temperatures 1.0 and 1.5.

Set	$T_{0.5}$	$T_{1.0}$	$T_{1.5}$
1.	n/a	0.00563	0.00713
2.	0.00348	0.00575	0.00725
3.	0.01126	0.01863	0.02288
4.	0.00300	0.00363	0.00463
5.	0.01350	0.01540	0.02040
6.	0.00108	0.00150	0.00163

Table 4.3: MSE numbers for MSK, 16-4-4 Neural Network

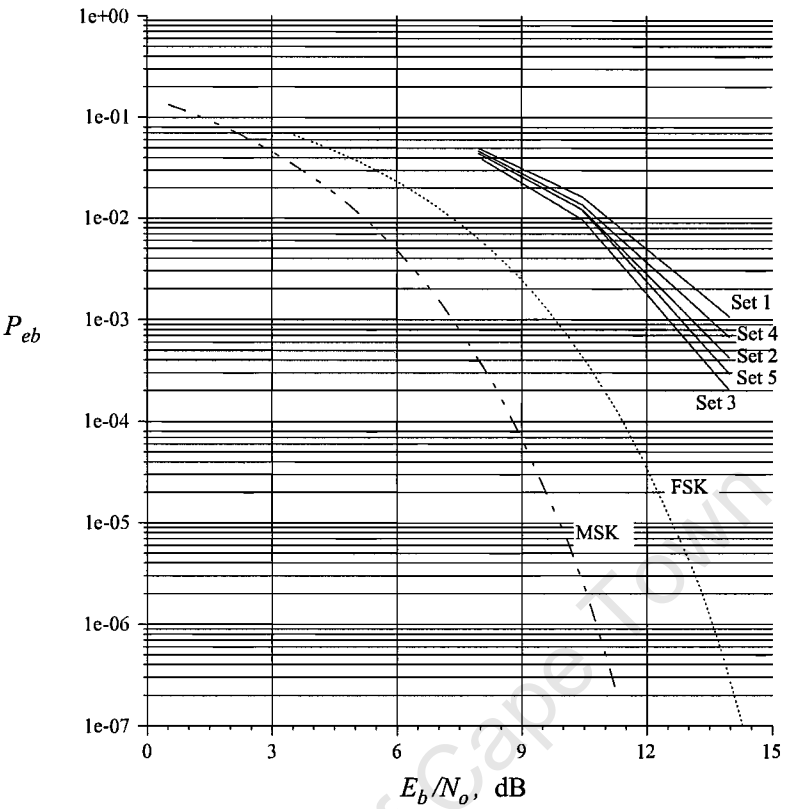


Figure 4.6: BER results for MSK using 8-4-4 Neural Network

Test	E_b/N_o , dB	$1T_{0.5}$	$1T_{1.0}$	$1T_{1.5}$	$2T_{0.5}$	$2T_{1.0}$	$2T_{1.5}$	$3T_{0.5}$	$3T_{1.0}$	$3T_{1.5}$	bits
1.	12.04	n/a	7	7	6	2	3	8	2	2	500k
2.	10.45	n/a	57	47	60	26	25	51	21	23	500k
3.	7.95	n/a	217	197	240	166	166	241	136	147	100k

Table 4.4: 16-4-4 Neural Network MSK Bit Errors

Test	E_b/N_o , dB	$4T_{0.5}$	$4T_{1.0}$	$4T_{1.5}$	$5T_{0.5}$	$5T_{1.0}$	$5T_{1.5}$	$6T_{0.5}$	$6T_{1.0}$	$6T_{1.5}$	bits
4.	12.04	4	2	4	8894	1004	816	5	3	4	500k
5.	10.45	57	27	26	18057	2778	2388	65	34	31	500k
6.	7.95	240	168	173	7832	1759	1557	250	186	172	100k

Table 4.5: 16-4-4 Neural Network MSK Bit Errors

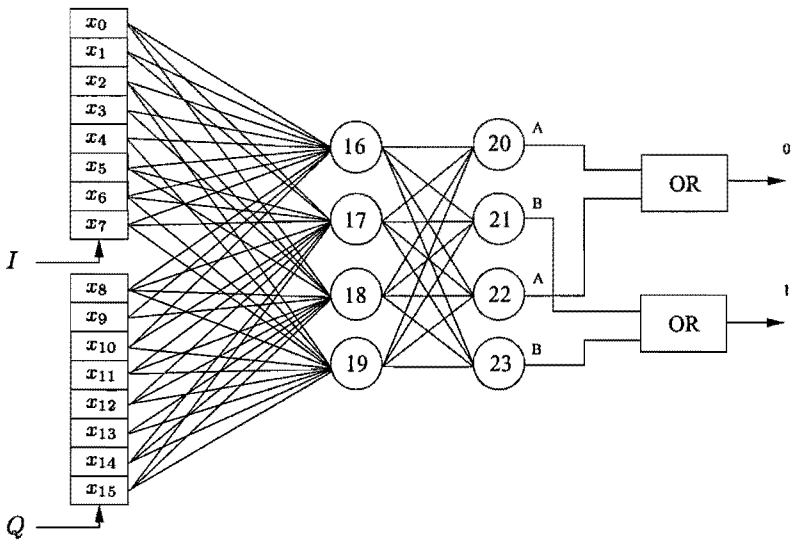


Figure 4.7: Fully Connected 16-4-4 Feedforward Network for MSK (not all connections shown)

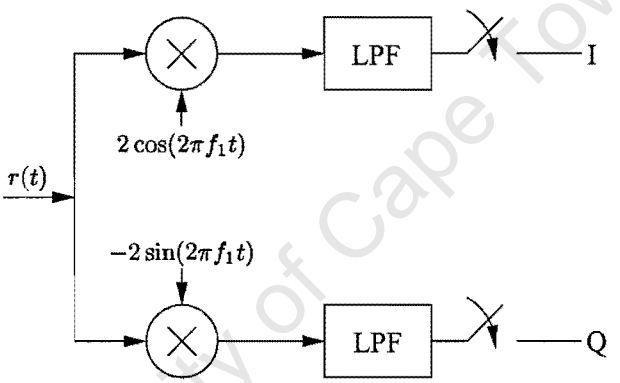


Figure 4.8: Demodulator structure

4.2.1 Sensitivity Analysis

The MSE is used as an indicator to signal convergence of weights down to an acceptable level. However, as shown above, this is not an appropriate measure since it does not reveal anything about sensitivities caused by internal nodes. Also, attempts have been made to analyze the response of the feed-forward Neural Net to noise [32]. In their research, Veciana and Zakhor [32] derived the noise factor for each input vector. This allowed them to establish *a priori* which classifier will perform better. The noise factor is the ratio between output and input noise variance. They discovered a large variation in the noise factors for vectors within a given training set. Some vectors would allow much more noise to propagate to the output than others.

In this section I will try to analyze the sensitivity of the Neural Net's output to perturbations on its input vector. Let $d_j(\mathbf{x})$ be the output of output node j of the NN in response to an input vector $\mathbf{x} = \{x_0, x_1, x_2, \ldots x_{15}\}$. We can approximate the sensitivity of d_j to perturbations on the input vector, using the first two terms of the Taylor-series expansion [31]:

$$d_j(x_0 + \Delta x_0, x_1, x_2, \ldots x_{15}) = d_j(\mathbf{x}) + \frac{\partial d_j(\mathbf{x})}{\partial x_0}\Delta x_0 \tag{4.4}$$

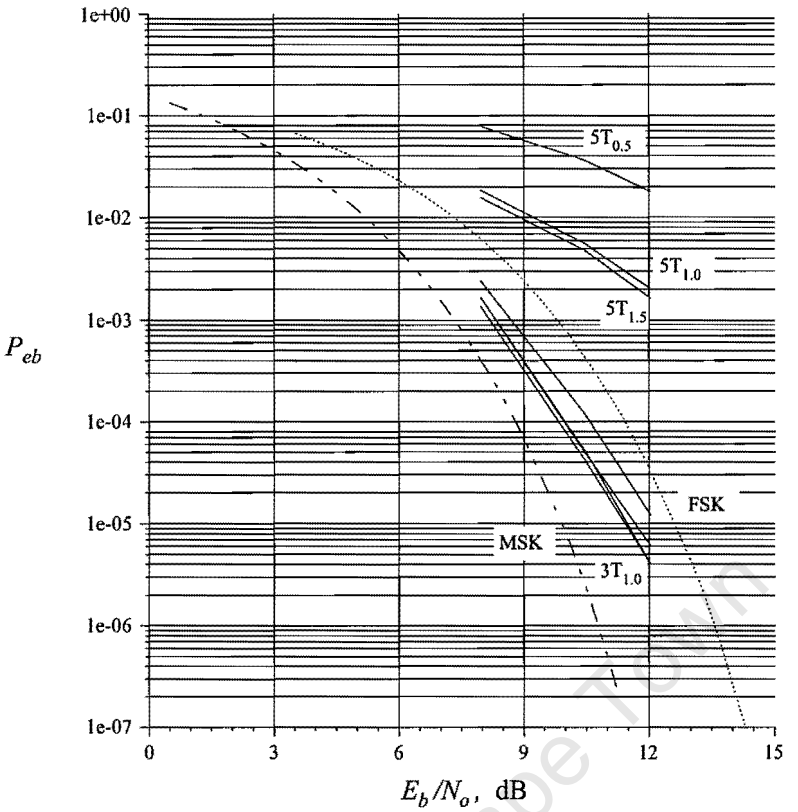


Figure 4.9: Best and worst BER results for MSK using 16-4-4 Neural Net

Solving for $\frac{\partial d_j}{\partial x_0}$ and using the *central-difference quotient* to approximate the derivatives result in:

$$\frac{\partial d_j(\mathbf{x})}{\partial x_0} \approx \frac{d_j(x_0 + \Delta x_0, x_1, x_2, \ldots x_{15}) - d_j(x_0 - \Delta x_0, x_1, x_2, \ldots x_{15})}{2\Delta x_0} \tag{4.5}$$

I used this equation to calculate the sensitivity matrix. It should be noted that if Δx_0 is too large, the difference quotient is normally a poor approximation of the derivative. The sensitivity matrix consist of $\partial d_j(\mathbf{x})/\partial x_0$ values for all output nodes (all j's). The matrix has one sensitivity number for each element of the input vector $\mathbf{x}$, for all valid input vectors. Each element in each input vector is perturbed by $\Delta x_i = x_i/10$. Table 4.6 shows the maximum values among the elements of the sensitivity matrixes for each training set. See appendix C for a listing of two of these matrixes ($5T_{0.5}$ and $3T_{1.0}$).

Training Set	$T_{0.5}$	$T_{1.0}$	$T_{1.5}$
$1T_x$	n/a	0.0926	0.0577
$2T_x$	0.1766	0.1293	0.0459
$3T_x$	0.2091	0.1663	0.0665
$4T_x$	0.1807	0.1216	0.0546
$5T_x$	0.3883	0.2133	0.2680
$6T_x$	0.1420	0.0901	0.0429

Table 4.6: Maximum sensitivity numbers parameters for 16-4-4 Neural Net for MSK

Although using only the maximum element of the matrix is not an appropriate criterion (some measure need to be derived that take into account all the sensitivities) to describe the *a priori* performance of the network, it does show better agreement between actual performance and predicted performance of the network than does the MSE. Figure 4.10 shows a comparison between

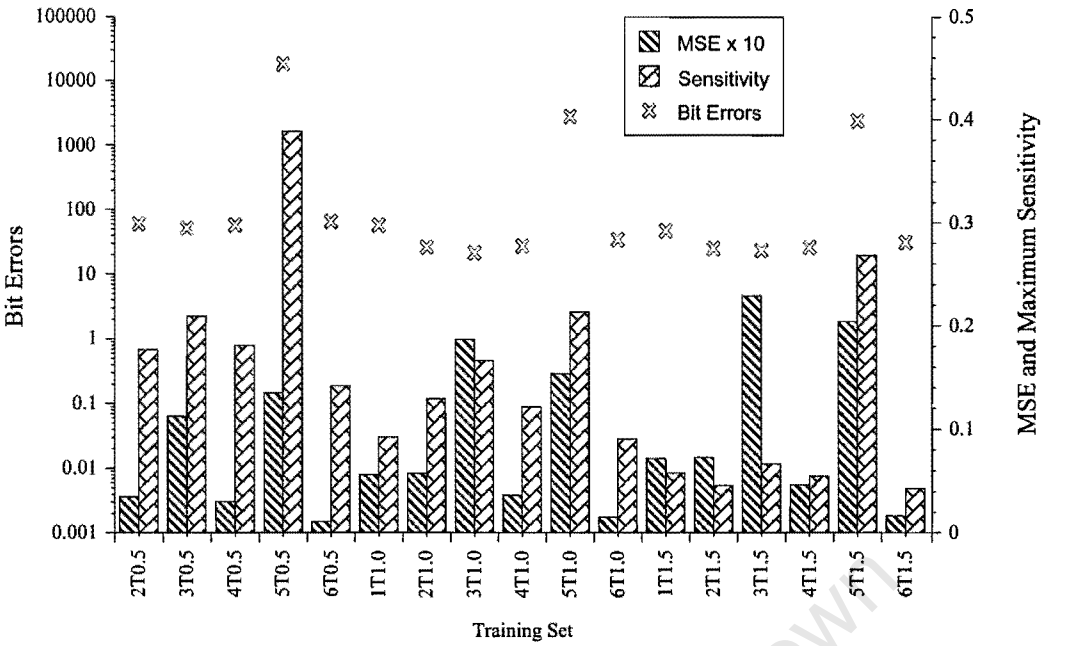


Figure 4.10: Maximum Sensitivity Numbers, MSE and corresponding number of bit errors at E_b/N_o of 10.45 dB, 16-4-4 NN

the MSE and maximum sensitivity number and the corresponding number of bit errors for an $E_b/N_o = 10.45\,\text{dB}$.

Experiment 3: 16-8-4 Network for MSK

In this experiment we increased the number of hidden nodes. The results are presented in Table 4.7, Table 4.8 and Figure 4.11. Comparing Table 4.7 with Table 4.3 shows that the increased number of neurons led to a faster convergence on a solution. Despite the lower MSE, this network performed worse than the 16-4-4 net. Under certain circumstances, neural nets are known to exhibit deteriorating performance with increasing number of hidden neurons. As stated before, one should use as many neurons as is necessary to draw the required decision boundaries. Too many neurons may lead to too high an order function for establishing the decision boundaries. Again, the sensitivity matrix provided a better measure of performance than the MSE (see Figure 4.12).

Set	MSE	Max. Sensitivity
$1T_{1.0}$	0.0031	0.0219
$2T_{1.0}$	0.0031	0.0877
$3T_{1.0}$	0.0106	0.1121
$4T_{1.0}$	0.0025	0.0953
$5T_{1.0}$	0.0094	0.0840
$6T_{1.0}$	0.0009	0.0689

Table 4.7: MSE and maximum sensitivity numbers for MSK, 16-8-4 Neural Network

Experiment 4: 16-8-8 Network for MSK

In the previous experiments, each output node was associated with two exemplars. That is, each output node was trained to indicate two possible input vectors (see Table 4.2), where this pair of input vectors were associated with either a 0 or a 1 transmitted in the first interval of the

Test	E_b/N_o , dB	$1T_{1.0}$	$2T_{1.0}$	$3T_{1.0}$	$4T_{1.0}$	$5T_{1.0}$	$6T_{1.0}$	bits
1.	12.04	5	121	111	254	1359	220	500k
2.	10.45	73	704	655	1079	3312	1006	500k
3.	7.95	240	803	770	1032	1874	991	100k

Table 4.8: 16-8-4 Neural Network MSK Bit Errors

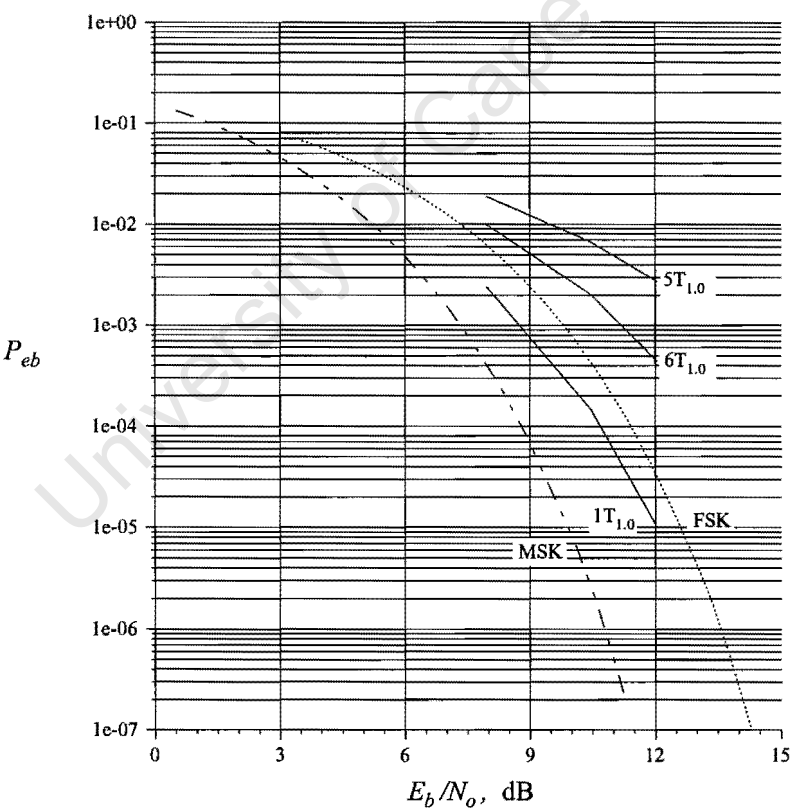


Figure 4.11: BER results for MSK using 16-8-4 Neural Net

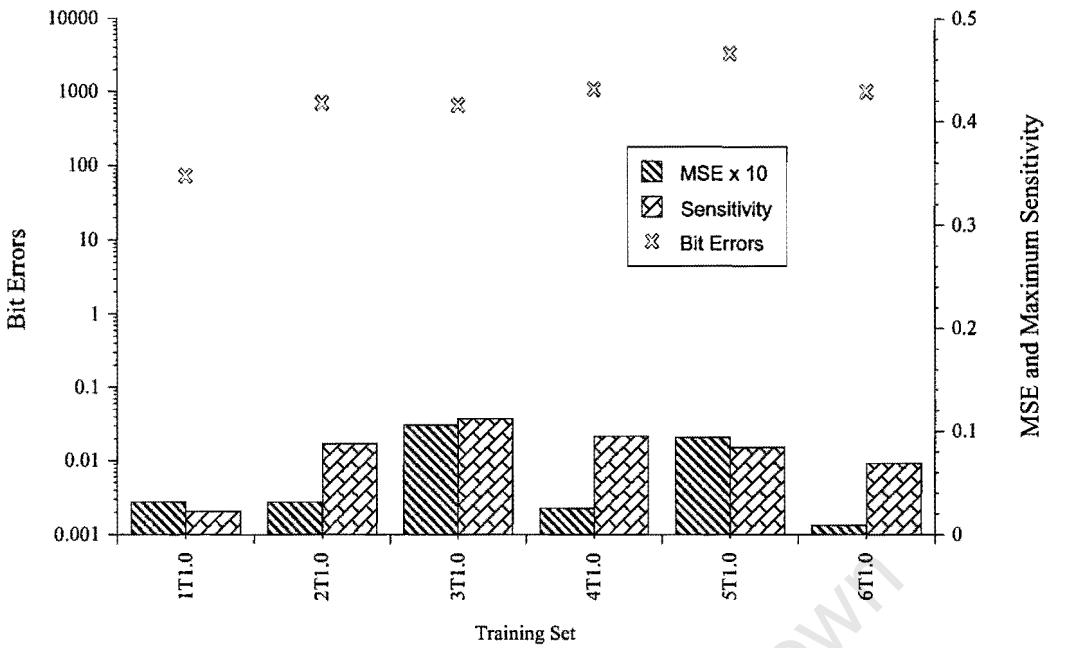


Figure 4.12: Maximum Sensitivity Numbers, MSE and corresponding number of bit errors at E_b/N_o of 10.45 dB, 16-8-4 NN

observation period. In this experiment we assigned each exemplar an unique output node and we expect to see better results. We do this experiment using training sets $3T_{0.5}$, $3T_{1.0}$ and $3T_{1.5}$, but now with varying number of training cycles to ensure that the MSE is between 0.0007 and 0.0004 (except for set $3T_{1.0b}$). The results are shown in Tables 4.10 and 4.9. The maximum sensitivity numbers does not show very good agreement with bit error performance. This may indicate that the complete sensitivity matrix should be used as a figure of merit rather than the maximum sensitivity number.

The error rate for this network is surprisingly higher than the 16-4-4 network.

Parameter	$3T_{0.5}$	$3T_{1.0}$	$3T_{1.0b}$	$3T_{1.5}$
MSE	0.000660	0.000515	0.01380	0.000493
Max. Sensitivity	0.0251	0.0652	0.0561	0.0263
Training Cycles	3000	14000	4000	15000

Table 4.9: 16-8-8 Neural Network for MSK showing MSE, training cycles and maximum sensitivity numbers

Test	E_b/N_o , dB	$3T_{0.5}$	$3T_{1.0}$	$3T_{1.0b}$	$3T_{1.5}$	bits
1.	12.04	57	48	38	97	500k
2.	10.45	373	256	299	554	500k
3.	7.95	621	426	525	695	100k

Table 4.10: 16-8-8 Neural Network MSK Bit Errors

Experiment 5: 16-16-8 Network for MSK

In this experiment we increased the number of hidden neurons. Tables 4.11 and 4.12 show the results for training set $3T_{1.0}$. Increasing the number of hidden neurons improved the performance with respect to the 16-8-8 net, but performance is still disappointing compared to the simple 16-4-4

network.

Parameter	$3T_{1.0}$
MSE	0.000266
Max. Sensitivity	0.0346
Training Cycles	13000

Table 4.11: 16-16-8 Neural Network for MSK showing MSE, training cycles and maximum sensitivity numbers

Test	E_b/N_o , dB	$3T_{1.0}$	bits
1.	12.04	12	500k
2.	10.45	153	500k
3.	7.95	461	100k

Table 4.12: 16-16-8 Neural Network MSK Bit Errors

Experiment 6: 16-16-8 Network for 4-*ary* CPFSK

The results of this experiment were disappointing. Table 4.14 and Tables 4.15, 4.16 and 4.17 show that some decoding is performed, but that the error rates are completely unacceptable. The same decoder concept was used as before and as shown in Figure 4.7. This bad performance can be attributed to either too many (over-fitting of the decision boundaries) or too few hidden neurons, and /or to the occurrence of a local minimum in the weight space. I did not perform a sensitivity analysis for each training set. The maximum sensitivity numbers for set $1T_{0.5}$ and $6T_{1.5}$ were 0.4709 and 0.2812 respectively. This contradicts our previous success, but as stated before, the complete sensitivity matrix should be studied instead of one number only. Another contributing factor to the poor performance of this net may be due to the way the exemplars were set up. Table 4.13 list the class A exemplars. Each output node were required to identify eight different input vectors. This setup was applied to MSK and a feasible explanation for its success is that the 2 input vectors associated with each output node are coincidentally on the same side of a decision boundary for that specific net. The eight input vectors associated with each output node may be on different sides of a decision boundary for this specific net.

4.2.2 Performance Analysis and Summary

With respect to sigmoid temperature, the best results were obtained for values between $T = 1.0$ and $T = 1.5$. One might expect a net to perform better with hidden nodes saturated and having higher temperatures since this would indicate lower hidden sensitivities.

The MSE criteria is only a measure of how close the output nodes are to required values when presented with a specific input vector. It does not reveal any hidden sensitivities. The sensitivity matrix should be used as a convergence measure in training and to evaluate a network's performance under noisy circumstances.

The 16-4-4 net for MSK network performed surprisingly better than the nets for which each output node were required to identify an unique input vector. I cannot explain this, except than to say that the input vector pair associated with each output node for the 16-4-4 net may be on the same side of some decision boundary.

Comparing the results of the 16-8-8 and the 16-16-8 networks for MSK show that increasing the number of hidden nodes improve the performance. This tendency was also noticed by de Veciana

Interval > class	$1st$	$2nd$	output node setting
A_0	$S_{0,0}$	$S_{0,0}$	$d_i = 0$, $d_{33} = 1$
A_1	$S_{0,0}$	$S_{1,0}$	$d_i = 0$, $d_{33} = 1$
A_2	$S_{0,0}$	$S_{2,0}$	$d_i = 0$, $d_{33} = 1$
A_3	$S_{0,0}$	$S_{3,0}$	$d_i = 0$, $d_{33} = 1$
A_4	$S_{0,1}$	$S_{0,1}$	$d_i = 0$, $d_{33} = 1$
A_5	$S_{0,1}$	$S_{1,1}$	$d_i = 0$, $d_{33} = 1$
A_6	$S_{0,1}$	$S_{2,1}$	$d_i = 0$, $d_{33} = 1$
A_7	$S_{0,1}$	$S_{3,1}$	$d_i = 0$, $d_{33} = 1$
A_8	$S_{0,2}$	$S_{0,2}$	$d_i = 0$, $d_{37} = 1$
A_9	$S_{0,2}$	$S_{1,2}$	$d_i = 0$, $d_{37} = 1$
A_{10}	$S_{0,2}$	$S_{2,2}$	$d_i = 0$, $d_{37} = 1$
A_{11}	$S_{0,2}$	$S_{3,2}$	$d_i = 0$, $d_{37} = 1$
A_{12}	$S_{0,3}$	$S_{0,3}$	$d_i = 0$, $d_{37} = 1$
A_{13}	$S_{0,3}$	$S_{1,3}$	$d_i = 0$, $d_{37} = 1$
A_{14}	$S_{0,3}$	$S_{2,3}$	$d_i = 0$, $d_{37} = 1$
A_{15}	$S_{0,3}$	$S_{3,3}$	$d_i = 0$, $d_{37} = 1$

Table 4.13: Exemplars for 4-*ary* CPFSK training (only class A shown here)

Set	$T_{0.5}$	$T_{1.0}$	$T_{1.5}$
1.	0.00136	0.00157	0.00215
2.	0.06343	0.00218	0.00485
3.	0.00435	0.00688	0.00675
4.	0.00127	0.00081	0.00096
5.	0.13124	0.06799	0.00773
6.	0.06283	0.00036	0.000472

Table 4.14: 16-16-8 Neural Network for 4-*ary* CPFSK, MSE

Test	E_b/N_o , dB	$1T_{0.5}$	$1T_{1.0}$	$1T_{1.5}$	$2T_{0.5}$	$2T_{1.0}$	$2T_{1.5}$	bits
1.	17.00	399	1970	4207	7462	5005	3363	100k
2.	13.47	3077	7588	8167	10925	9232	10988	100k
3.	10.97	8013	15235	14969	16520	15655	19326	100k
4.	7.45	20904	30493	30033	29518	29751	22867	100k

Table 4.15: 16-16-8 Neural Network 4-*ary* CPFSK Bit Errors

Test	E_b/N_o , dB	$3T_{0.5}$	$3T_{1.0}$	$3T_{1.5}$	$4T_{0.5}$	$4T_{1.0}$	$4T_{1.5}$	bits
5.	17.00	4410	1730	1915	4848	1215	1633	100k
6.	13.47	11161	7035	7803	11607	6477	8109	100k
7.	10.97	19367	14883	15573	19652	14475	16681	100k
8.	7.45	34494	30036	30942	34578	30151	32165	100k

Table 4.16: 16-16-8 Neural Network 4-*ary* CPFSK Bit Errors

Test	E_b/N_o , dB	$5T_{0.5}$	$5T_{1.0}$	$5T_{1.5}$	$6T_{0.5}$	$6T_{1.0}$	$6T_{1.5}$	bits
9.	17.00	13289	7495	3301	7559	2309	2969	100k
10.	13.47	16538	11785	10582	10559	6669	9733	100k
11.	10.97	21596	18419	18877	15793	13230	18149	100k
12.	7.45	33310	32534	33650	28574	28073	33185	100k

Table 4.17: 16-16-8 Neural Network 4-*ary* CPFSK Bit Errors

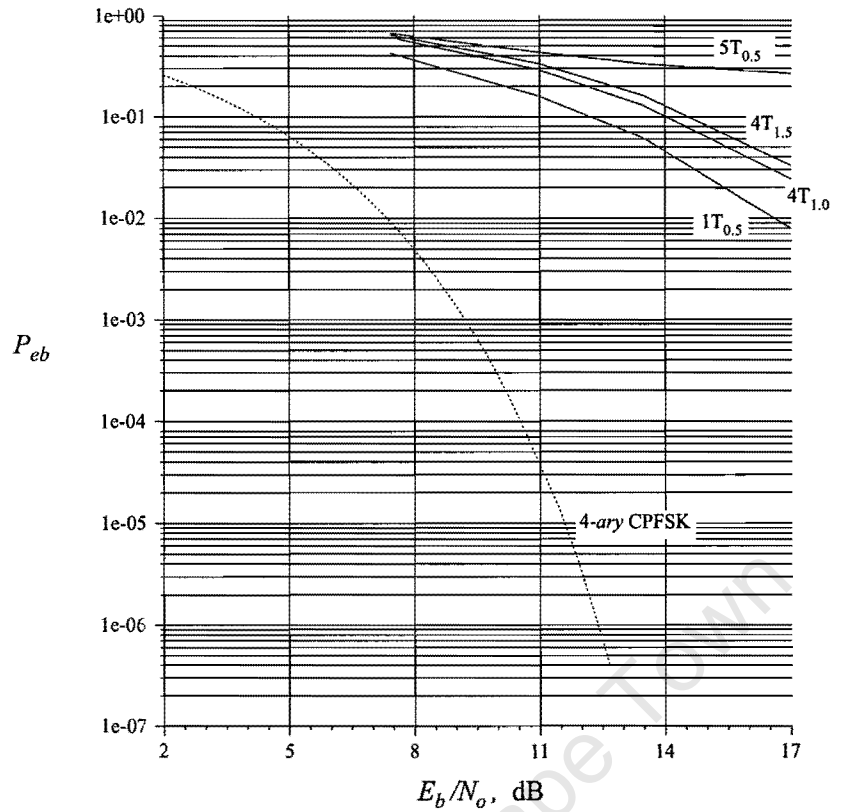


Figure 4.13: Best and worst BER results for 4-*ary* CPFSK

and Zakhor [32], and I agree with their rule of thumb for best performance: have approximately as many hidden nodes as input nodes.

The occurrence of local minima may always be a problem. One possible way of minimizing the phenomenon of local minima may be to apply a guided random search (genetic algorithm) for optimum weights. Otherwise, my suggestion is to keep with the standard back-propagation training algorithm (except than to integrate the sensitivity matrix into the convergence criterion) and do manual guided searches. This may include:

1. Lowering the gain term η in the back-propagation training algorithm
2. Performing training runs starting with many different sets of initial random weights, and
3. Changing the order in which the inputs vectors are presented

Further options for future research may include using a larger observation window and a higher sampling rate. It could also be that the neural nets applied in this research is not able to define the complex decision boundaries required by convolutional codes. More emphasis should be placed on understanding the boundaries formed by these neural nets. A final note is that nearly all the symbol errors generated by the neural nets occurred as single burst length errors. This was not the case with the Fano algorithm where nearly all the symbol errors occurred as burst errors of length 2, whereas about 30 to 40 percent of the errors made by the Viterbi decoder were single symbol errors.

Wicker and Wang [33] designed a neural network that implements the Viterbi algorithm using analog artificial neurons. The performance of this network matches the ideal Viterbi algorithm since it is basically a "neuralized" version of the Viterbi algorithm. They point out that the code

words of conventional pattern recognition problems form a vector space over a finite field, whereas the algebraic properties of practical block codes and convolutional codes may introduce additional structure. This type of artificial neural net decoder is superior in terms of speed and is better suited for VLSI implementation than current decoder designs.

4.2.3 Complexity

Veciana and Zakhor [32] provide an arithmetic complexity comparison between conventional receivers and NN based receivers. Their arithmetic comparison is based on a digital implementation and in terms of number of addition, multiply, table lookup, compare and select operations performed per symbol interval for each receiver. The conclusion is that the NN numerical complexity exceeds that of a conventional Viterbi implementation by a factor of about 3 (Wicker's neural net decoder as discussed above is excluded from these calculations). Whereas the Viterbi decoder requires substantial RAM and some ROM, the NN requires virtually no RAM but substantial ROM.

Chapter 5

Conclusion

5.1 Summary

5.1.1 The Fano Algorithm

This thesis contains new results on the subject of two methods for decoding the continuous-phase modulated signal. Neither of these two techniques have been effectively implemented in practice and there is a proportional paucity in research results, especially for the Fano decoder.

We applied the Fano decoder to 4-*ary* CPFSK with the purpose of investigating *firstly*, its suitability as decoder for CPM in general, and *secondly*, the possibility of it providing a less complex decoder structure. The second technique, namely the neural network decoder, was in addition to 4-*ary* CPFSK also applied to MSK.

We confirmed through simulation that the Fano decoder approaches the bit error rate performance of the optimum decoder for high signal-to-noise ratios.

The major problem at lower E_b/N_o seems to be that of undetected re-merging paths. The re-merging of paths causes error bursts of length 2. One important conclusion is that catastrophic events are, within a fairly large range of threshold and slope parameters, not the handicap we anticipated it to be. Of special importance is the results on the percentage of time the decoder spent looking back and actually moving back as a function of E_b/N_o, Fano threshold, and Fano slope parameters. We showed that at E_b/N_o = 13.1 dB where bit error rate performance is still acceptable, look-backs amount to about 5 percent of symbols received and move-backs amount to about 2 percent.

One disadvantage is the decoding delay with respect to the Viterbi decoder. This delay also determines the amount of memory required since all metric values need to be stored up to this depth. This delay is a function of the maximum depth reached in looking back (values in brackets) for E_b/N_o= 16 dB (3), 13.1 dB (5) and 8.4 dB (12). However, it is not certain how the decoding delay of the Fano algorithm relative to those of the Viterbi will change for schemes with longer constraint length. It may well be that the decoding delay compared to those of the Viterbi decoder decreases with increase in constraint length.

The Fano algorithm was substantially more complex to implement in software than the Viterbi algorithm, but it remains to be investigated how the respective complexities of the hardware implementations compare with each other.

5.1.2 The Neural Net Decoder

The neural network decoder take as input samples of the baseband in-phase and quadrature waveforms instead of the traditional approach of using the output from a set of matched filters as input for the neural network. Our bit error rate results were disappointing and confirms the results of other researchers.

We presented a method for determining the hidden sensitivities of the neural network which gives an *a priori* performance indication. This method should rather be used as a convergence measure in training and to evaluate a network's performance under noisy circumstances.

We concluded that the decision regions required by convolutional codes have additional structure which is not adequately represented by the decision boundaries formed by feed-forward neural nets. The NN's numerical complexity exceeds that of the conventional Viterbi implementation by a factor of about 3.

5.2 Future Work

Of the two techniques applied, the Fano algorithm turned out to be the most promising. Future research on the subject of Fano decoders may be divided in two parts:

1. The research should be repeated for a modulation scheme with larger constraint length in order to establish the effect off constraint length on decoding delay and consequently memory usage. To what extend can the decoding delay be reduced for the algorithm to still function with acceptable performance?

2. Transforming the Fano algorithm into a simplified hardware implementation and doing a resource comparison with the Viterbi implementation.

Suggested future research on the neural network decoder may include the following:

1. A criteria which indicate *a priori* performance based on hidden neuron sensitivities should be investigated.

2. Finally, the additional structure in the decision regions required by convolutional codes need to be investigated for better understanding and possible modification to the neural net structure.

Appendix A

The Back-propagation Neural Net Training Algorithm

The back-propagation learning algorithm is a recursive gradient algorithm designed to minimize the mean squared error between the actual output of a multilayer feed-forward network and the desired output. The algorithm as presented here is from [15], but with slightly different notation.

We assume a sigmoidal activation function:

$$f_{act}(\alpha) = \frac{1}{1+e^{-\frac{\alpha}{T}}} \tag{A.1}$$

Step 1. **Initialize weights and offsets**

Set all synaptic weights and node offsets to small random values.

The weights were set upon initialization using the following C equations:

$1st$ layer: $w_1[i] = a * (1.0 * \text{rand}()/\text{RAND\_MAX} - 0.5)$

$2nd$ layer: $w_2[i] = b * (1.0 * \text{rand}()/\text{RAND\_MAX} - 0.5)$

offsets: $offset[i] = c * (1.0 * \text{rand}()/\text{RAND\_MAX} - 0.5)$;

Step 2. **Present input and desired outputs**

Present a continuous valued input vector $x_0, x_1, \ldots x_{N-1}$ and specify the desired outputs $d_0, d_1, \ldots d_{M-1}$. We will use the network as a classifier which means that all desired outputs are set to zero except for that corresponding to the class the input is from. The input samples used as training sets are presented cyclically until weights stabilize. For MSK, using an observation period equal to two, the training sets were associated with two major classes: class A and class B. Class A is associated with an $U_0 = 0$ transmitted in the first interval, class B is associated with $U_0 = 1$ transmitted in the first interval. Likewise for 4-ary CPFSK, the training sets were associated with four major classes: A ($U_0 = 0$), B ($U_0 = 1$), C ($U_0 = 2$) and D ($U_0 = 3$) where U_0 refer here to the symbol transmitted in the first interval of the observation period.

Step 3. **Calculate actual outputs**

Calculate outputs $y_0, y_1, \ldots y_{M-1}$.

Step 4. **Adapt weights**

Use a recursive procedure starting at the output nodes and working back to the first hidden layer. Adjust the synaptic weights according to

$$w_{ij}(t+1) = w_{ij}(t) + \eta \delta_j x_i'$$

x'_i is either the output of node i or is an input, η is a gain term, and δ_j is an error term for node j. If node j is an output node, then

$$\delta_j = y_j(1 - y_j)(d_j - y_j)\,,$$

where d_j is the desired output of node j, and y_j is the actual output.
If node j is an internal hidden node, then

$$\delta_j = x'_j(1 - x'_j)\sum_k \delta_k w_{jk}\,,$$

where k is over all nodes in the layers above node j. Internal node thresholds are adapted in a similar manner by assuming they are connection weights on links from auxiliary constant-valued inputs.

Step 5. **Repeat (K times) by going to step 2**

Appendix B

Perceptron Convergence Procedure for FSK

The original perceptron convergence (training) procedure was developed by ROSENBLATT [25]. Here we apply it to find the optimum weights for FSK.

Step 1. **Initialize weights and threshold**
Set $w_i(0)$, $(0 \leq i \leq N-1)$ and θ to small random values.

Step 2. **Present new input and desired output**
Present new continuous valued input $x_0, x_1 \ldots x_{N-1}$ along with the desired output $d(t)$.

Step 3. **Calculate actual output**

$$y(t) = f_h\left(\sum_{i=0}^{N-1} w_i(t)x_i(t) - \theta\right) \tag{B.1}$$

A linear activation function was used for this application:

$$y(t) = 0.2\left(\sum_{i=0}^{N-1} w_i(t)x_i(t) - \theta\right) \tag{B.2}$$

Step 4. **Adapt weights**

$$\begin{aligned} w_i(t+1) &= w_i(t) + \eta\left[d(t) - y(t)\right]x_i(t), \\ & \quad 0 \leq i \leq N-1 \end{aligned} \tag{B.3}$$

$$d(t) = \begin{cases} +1 & \textbf{if input from class A} \\ -1 & \textbf{if input from class B} \end{cases}$$

Step 5. **Repeat by going to Step 2**

Appendix C

Sensitivity Matrix for 16-4-4 NN MSK

C.1 Sensitivity matrix for training set $5T_{0.5}$

weights16-4-4

Sensitivity analysis: dx = 0.1000

ddj/dx0, class = 0 : -0.0447 0.0682 -0.0004 -0.0623
ddj/dx1, class = 0 : -0.0910 0.0381 0.0000 -0.0335
ddj/dx2, class = 0 : 0.0553 -0.0298 -0.0000 0.0173
ddj/dx3, class = 0 : 0.0006 0.0325 -0.0001 -0.0130
ddj/dx4, class = 0 : 0.1007 -0.0489 0.0001 0.0353
ddj/dx5, class = 0 : 0.0313 -0.0206 -0.0001 0.0123
ddj/dx6, class = 0 : 0.0487 -0.0548 0.0002 0.0459
ddj/dx7, class = 0 : -0.0497 0.0415 -0.0002 -0.0503
ddj/dx8, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx9, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx10, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx11, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx13, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx14, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx15, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 1 : -0.0121 0.0344 -0.0004 -0.0058
ddj/dx1, class = 1 : -0.0097 -0.0028 0.0001 0.0006
ddj/dx2, class = 1 : 0.0041 0.0030 -0.0001 -0.0008
ddj/dx3, class = 1 : 0.0048 0.0007 0.0000 0.0004
ddj/dx4, class = 1 : 0.0108 -0.0085 0.0001 0.0013
ddj/dx5, class = 1 : 0.0000 0.0000 0.0000 0.0000
ddj/dx6, class = 1 : -0.0086 0.0198 -0.0002 -0.0031
ddj/dx7, class = 1 : 0.0095 -0.0173 0.0002 0.0026
ddj/dx8, class = 1 : 0.0000 0.0000 0.0000 0.0000
ddj/dx9, class = 1 : 0.0000 0.0000 0.0000 0.0000
ddj/dx10, class = 1 : 0.0000 0.0000 0.0000 0.0000
ddj/dx11, class = 1 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 1 : -0.0032 -0.0034 0.0001 0.0011
ddj/dx13, class = 1 : 0.0000 0.0000 0.0000 0.0000

ddj/dx14, class = 1 : -0.0070 0.0146 -0.0002 -0.0030
ddj/dx15, class = 1 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 2 : 0.0007 -0.0468 0.0557 0.3176
ddj/dx1, class = 2 : 0.0020 -0.0185 -0.0025 0.2723
ddj/dx2, class = 2 : -0.0012 0.0141 0.0010 -0.1981
ddj/dx3, class = 2 : -0.0002 -0.0191 0.0168 0.1229
ddj/dx4, class = 2 : -0.0022 0.0220 -0.0094 -0.3691
ddj/dx5, class = 2 : -0.0006 0.0113 0.0135 -0.0847
ddj/dx6, class = 2 : -0.0009 0.0322 -0.0339 -0.2808
ddj/dx7, class = 2 : 0.0009 -0.0276 0.0256 0.2052
ddj/dx8, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx9, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx10, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx11, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx13, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx14, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx15, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 3 : 0.0002 -0.0335 0.1455 0.3883
ddj/dx1, class = 3 : 0.0018 -0.0015 -0.0368 0.0844
ddj/dx2, class = 3 : -0.0008 0.0018 0.0290 -0.0237
ddj/dx3, class = 3 : -0.0009 -0.0102 0.0107 0.0356
ddj/dx4, class = 3 : -0.0014 0.0051 -0.0145 -0.1665
ddj/dx5, class = 3 : 0.0000 0.0000 0.0000 0.0000
ddj/dx6, class = 3 : 0.0004 -0.0149 0.0746 0.2532
ddj/dx7, class = 3 : -0.0008 0.0126 -0.0609 -0.2757
ddj/dx8, class = 3 : 0.0000 0.0000 0.0000 0.0000
ddj/dx9, class = 3 : 0.0000 0.0000 0.0000 0.0000
ddj/dx10, class = 3 : 0.0000 0.0000 0.0000 0.0000
ddj/dx11, class = 3 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 3 : 0.0006 -0.0070 -0.0288 0.0532
ddj/dx13, class = 3 : -0.0005 0.0174 -0.0837 -0.2971
ddj/dx14, class = 3 : 0.0006 -0.0039 0.0620 0.1458
ddj/dx15, class = 3 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 4 : 0.0368 0.0155 -0.0360 -0.0002
ddj/dx1, class = 4 : 0.0000 0.0000 0.0000 0.0000
ddj/dx2, class = 4 : -0.0220 0.0146 0.0167 -0.0001
ddj/dx3, class = 4 : -0.0529 -0.0054 0.0300 -0.0000
ddj/dx4, class = 4 : -0.0658 0.0102 0.0305 -0.0002
ddj/dx5, class = 4 : 0.0227 -0.0062 -0.0054 0.0001
ddj/dx6, class = 4 : 0.0381 0.0289 -0.0360 -0.0002
ddj/dx7, class = 4 : -0.0187 0.0008 0.0256 0.0001
ddj/dx8, class = 4 : 0.0217 0.0239 -0.0061 -0.0000
ddj/dx9, class = 4 : 0.0354 0.0288 -0.0325 -0.0002
ddj/dx10, class = 4 : 0.0046 -0.0006 0.0002 0.0000
ddj/dx11, class = 4 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 4 : 0.0000 0.0000 0.0000 0.0000
ddj/dx13, class = 4 : 0.0000 0.0000 0.0000 0.0000
ddj/dx14, class = 4 : 0.0000 0.0000 0.0000 0.0000
ddj/dx15, class = 4 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 5 : 0.0062 0.0414 -0.0262 -0.0006

ddj/dx1, class = 5 : 0.0000 0.0000 0.0000 0.0000
ddj/dx2, class = 5 : -0.0336 0.0226 0.0064 -0.0003
ddj/dx3, class = 5 : -0.0332 -0.0249 0.0254 0.0002
ddj/dx4, class = 5 : -0.0679 0.0203 0.0122 -0.0005
ddj/dx5, class = 5 : 0.0000 0.0000 0.0000 0.0000
ddj/dx6, class = 5 : -0.0089 -0.0351 0.0181 0.0003
ddj/dx7, class = 5 : -0.0109 0.0307 -0.0188 -0.0005
ddj/dx8, class = 5 : 0.0449 0.0136 -0.0056 0.0002
ddj/dx9, class = 5 : 0.0449 0.0321 -0.0206 -0.0001
ddj/dx10, class = 5 : 0.0217 -0.0132 0.0026 0.0003
ddj/dx11, class = 5 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 5 : 0.0250 -0.0428 0.0086 0.0005
ddj/dx13, class = 5 : -0.0075 0.0588 -0.0193 -0.0007
ddj/dx14, class = 5 : 0.0116 0.0177 0.0037 -0.0000
ddj/dx15, class = 5 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 6 : -0.0435 -0.0001 0.0437 0.0355
ddj/dx1, class = 6 : 0.0000 0.0000 0.0000 0.0000
ddj/dx2, class = 6 : 0.0231 -0.0000 -0.0215 0.0134
ddj/dx3, class = 6 : 0.0606 0.0000 -0.0400 -0.0015
ddj/dx4, class = 6 : 0.0816 -0.0000 -0.0372 0.0377
ddj/dx5, class = 6 : -0.0303 0.0000 0.0079 -0.0164
ddj/dx6, class = 6 : -0.0528 -0.0001 0.0397 0.0392
ddj/dx7, class = 6 : 0.0227 0.0000 -0.0300 -0.0177
ddj/dx8, class = 6 : -0.0309 -0.0001 0.0078 0.0074
ddj/dx9, class = 6 : -0.0470 -0.0001 0.0365 0.0350
ddj/dx10, class = 6 : -0.0057 0.0000 -0.0002 -0.0072
ddj/dx11, class = 6 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 6 : 0.0000 0.0000 0.0000 0.0000
ddj/dx13, class = 6 : 0.0000 0.0000 0.0000 0.0000
ddj/dx14, class = 6 : 0.0000 0.0000 0.0000 0.0000
ddj/dx15, class = 6 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 7 : -0.0085 -0.0002 0.0309 0.1237
ddj/dx1, class = 7 : 0.0000 0.0000 0.0000 0.0000
ddj/dx2, class = 7 : 0.0288 -0.0001 -0.0076 0.0696
ddj/dx3, class = 7 : 0.0332 0.0001 -0.0320 -0.0505
ddj/dx4, class = 7 : 0.0666 -0.0001 -0.0144 0.1077
ddj/dx5, class = 7 : 0.0000 0.0000 0.0000 0.0000
ddj/dx6, class = 7 : 0.0120 0.0001 -0.0207 -0.0773
ddj/dx7, class = 7 : 0.0105 -0.0001 0.0198 0.1069
ddj/dx8, class = 7 : -0.0529 -0.0001 0.0075 -0.0391
ddj/dx9, class = 7 : -0.0505 -0.0001 0.0242 0.0227
ddj/dx10, class = 7 : -0.0209 0.0000 -0.0029 -0.0582
ddj/dx11, class = 7 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 7 : -0.0211 0.0002 -0.0109 -0.1254
ddj/dx13, class = 7 : 0.0027 -0.0002 0.0230 0.1551
ddj/dx14, class = 7 : -0.0124 -0.0001 -0.0032 0.0010
ddj/dx15, class = 7 : 0.0000 0.0000 0.0000 0.0000
maxdj/dx = 0.3883

C.2 Sensitivity matrix for training set $3T_{1.0}$

weights16-4-4

Sensitivity analysis: dx = 0.1000

ddj/dx0, class = 0 : 0.0297 0.0124 -0.0001 -0.0087
ddj/dx1, class = 0 : 0.0048 0.0028 -0.0000 -0.0046
ddj/dx2, class = 0 : 0.0121 0.0021 -0.0000 0.0018
ddj/dx3, class = 0 : 0.0191 0.0026 -0.0000 0.0063
ddj/dx4, class = 0 : 0.0139 0.0017 -0.0000 0.0050
ddj/dx5, class = 0 : 0.0027 0.0011 -0.0000 -0.0009
ddj/dx6, class = 0 : -0.0054 -0.0019 0.0000 0.0001
ddj/dx7, class = 0 : 0.0029 0.0006 -0.0000 0.0006
ddj/dx8, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx9, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx10, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx11, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx13, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx14, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx15, class = 0 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 1 : 0.0282 0.0103 -0.0001 -0.0050
ddj/dx1, class = 1 : 0.0042 0.0018 -0.0000 -0.0031
ddj/dx2, class = 1 : 0.0119 0.0011 -0.0000 0.0029
ddj/dx3, class = 1 : 0.0191 0.0019 -0.0000 0.0068
ddj/dx4, class = 1 : 0.0098 0.0008 -0.0000 0.0038
ddj/dx5, class = 1 : 0.0000 0.0000 0.0000 0.0000
ddj/dx6, class = 1 : 0.0038 0.0013 -0.0000 0.0002
ddj/dx7, class = 1 : -0.0028 -0.0004 0.0000 -0.0008
ddj/dx8, class = 1 : 0.0000 0.0000 0.0000 0.0000
ddj/dx9, class = 1 : 0.0000 0.0000 0.0000 0.0000
ddj/dx10, class = 1 : 0.0000 0.0000 0.0000 0.0000
ddj/dx11, class = 1 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 1 : 0.0052 0.0007 -0.0000 0.0020
ddj/dx13, class = 1 : 0.0000 0.0000 0.0000 0.0000
ddj/dx14, class = 1 : -0.0019 -0.0009 -0.0000 -0.0009
ddj/dx15, class = 1 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 2 : -0.0009 -0.0183 0.0202 0.0079
ddj/dx1, class = 2 : -0.0001 -0.0039 0.0059 0.0046
ddj/dx2, class = 2 : -0.0004 -0.0034 0.0067 -0.0021
ddj/dx3, class = 2 : -0.0006 -0.0045 0.0078 -0.0069
ddj/dx4, class = 2 : -0.0004 -0.0030 0.0055 -0.0054
ddj/dx5, class = 2 : -0.0001 -0.0016 0.0020 0.0008
ddj/dx6, class = 2 : 0.0002 0.0027 -0.0027 0.0001
ddj/dx7, class = 2 : -0.0001 -0.0009 0.0014 -0.0007
ddj/dx8, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx9, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx10, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx11, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx13, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx14, class = 2 : 0.0000 0.0000 0.0000 0.0000

ddj/dx15, class = 2 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 3 : -0.0010 -0.0204 0.0223 0.0104
ddj/dx1, class = 3 : -0.0002 -0.0047 0.0067 0.0056
ddj/dx2, class = 3 : -0.0004 -0.0038 0.0070 -0.0017
ddj/dx3, class = 3 : -0.0006 -0.0046 0.0078 -0.0068
ddj/dx4, class = 3 : -0.0003 -0.0022 0.0039 -0.0038
ddj/dx5, class = 3 : 0.0000 0.0000 0.0000 0.0000
ddj/dx6, class = 3 : -0.0001 -0.0022 0.0021 0.0001
ddj/dx7, class = 3 : 0.0001 0.0009 -0.0014 0.0006
ddj/dx8, class = 3 : 0.0000 0.0000 0.0000 0.0000
ddj/dx9, class = 3 : 0.0000 0.0000 0.0000 0.0000
ddj/dx10, class = 3 : 0.0000 0.0000 0.0000 0.0000
ddj/dx11, class = 3 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 3 : -0.0002 -0.0012 0.0017 -0.0023
ddj/dx13, class = 3 : 0.0000 0.0019 -0.0012 -0.0022
ddj/dx14, class = 3 : 0.0001 0.0005 0.0006 0.0017
ddj/dx15, class = 3 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 4 : 0.0931 0.0448 -0.0460 0.0006
ddj/dx1, class = 4 : 0.0000 0.0000 0.0000 0.0000
ddj/dx2, class = 4 : -0.0541 -0.0171 0.0241 -0.0005
ddj/dx3, class = 4 : -0.1317 -0.0454 0.0558 -0.0013
ddj/dx4, class = 4 : -0.0934 -0.0378 0.0433 -0.0010
ddj/dx5, class = 4 : -0.0137 -0.0069 0.0098 -0.0001
ddj/dx6, class = 4 : 0.0317 0.0184 -0.0168 0.0003
ddj/dx7, class = 4 : -0.0181 -0.0080 0.0101 -0.0002
ddj/dx8, class = 4 : 0.0335 0.0228 -0.0163 0.0002
ddj/dx9, class = 4 : 0.0551 0.0361 -0.0284 0.0003
ddj/dx10, class = 4 : 0.0039 0.0053 -0.0018 -0.0000
ddj/dx11, class = 4 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 4 : 0.0000 0.0000 0.0000 0.0000
ddj/dx13, class = 4 : 0.0000 0.0000 0.0000 0.0000
ddj/dx14, class = 4 : 0.0000 0.0000 0.0000 0.0000
ddj/dx15, class = 4 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 5 : 0.0925 0.0433 -0.0445 0.0006
ddj/dx1, class = 5 : 0.0000 0.0000 0.0000 0.0000
ddj/dx2, class = 5 : -0.0540 -0.0168 0.0237 -0.0005
ddj/dx3, class = 5 : -0.1317 -0.0453 0.0554 -0.0013
ddj/dx4, class = 5 : -0.0660 -0.0267 0.0305 -0.0007
ddj/dx5, class = 5 : 0.0000 0.0000 0.0000 0.0000
ddj/dx6, class = 5 : -0.0234 -0.0127 0.0116 -0.0002
ddj/dx7, class = 5 : 0.0177 0.0079 -0.0101 0.0002
ddj/dx8, class = 5 : 0.0341 0.0225 -0.0155 0.0002
ddj/dx9, class = 5 : 0.0560 0.0352 -0.0270 0.0003
ddj/dx10, class = 5 : 0.0039 0.0051 -0.0014 -0.0000
ddj/dx11, class = 5 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 5 : -0.0391 -0.0157 0.0178 -0.0004
ddj/dx13, class = 5 : 0.0002 0.0025 -0.0000 -0.0001
ddj/dx14, class = 5 : 0.0169 0.0099 -0.0046 0.0002
ddj/dx15, class = 5 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 6 : -0.1258 -0.0009 0.0107 -0.0229
ddj/dx1, class = 6 : 0.0000 0.0000 0.0000 0.0000

ddj/dx2, class = 6 : 0.0638 0.0003 -0.0061 0.0198
ddj/dx3, class = 6 : 0.1663 0.0009 -0.0133 0.0535
ddj/dx4, class = 6 : 0.1411 0.0008 -0.0094 0.0412
ddj/dx5, class = 6 : 0.0240 0.0001 -0.0020 0.0036
ddj/dx6, class = 6 : -0.0565 -0.0004 0.0033 -0.0111
ddj/dx7, class = 6 : 0.0306 0.0002 -0.0021 0.0075
ddj/dx8, class = 6 : -0.0584 -0.0005 0.0033 -0.0087
ddj/dx9, class = 6 : -0.0910 -0.0007 0.0059 -0.0114
ddj/dx10, class = 6 : -0.0060 -0.0001 0.0004 0.0023
ddj/dx11, class = 6 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 6 : 0.0000 0.0000 0.0000 0.0000
ddj/dx13, class = 6 : 0.0000 0.0000 0.0000 0.0000
ddj/dx14, class = 6 : 0.0000 0.0000 0.0000 0.0000
ddj/dx15, class = 6 : 0.0000 0.0000 0.0000 0.0000
ddj/dx0, class = 7 : -0.1248 -0.0009 0.0104 -0.0246
ddj/dx1, class = 7 : 0.0000 0.0000 0.0000 0.0000
ddj/dx2, class = 7 : 0.0636 0.0003 -0.0060 0.0200
ddj/dx3, class = 7 : 0.1661 0.0009 -0.0134 0.0531
ddj/dx4, class = 7 : 0.0997 0.0005 -0.0067 0.0288
ddj/dx5, class = 7 : 0.0000 0.0000 0.0000 0.0000
ddj/dx6, class = 7 : 0.0392 0.0003 -0.0024 0.0083
ddj/dx7, class = 7 : -0.0307 -0.0002 0.0020 -0.0075
ddj/dx8, class = 7 : -0.0576 -0.0005 0.0032 -0.0093
ddj/dx9, class = 7 : -0.0895 -0.0007 0.0058 -0.0128
ddj/dx10, class = 7 : -0.0058 -0.0001 0.0003 0.0019
ddj/dx11, class = 7 : 0.0000 0.0000 0.0000 0.0000
ddj/dx12, class = 7 : 0.0568 0.0003 -0.0039 0.0161
ddj/dx13, class = 7 : -0.0003 -0.0001 0.0000 0.0025
ddj/dx14, class = 7 : -0.0259 -0.0002 0.0010 -0.0070
ddj/dx15, class = 7 : 0.0000 0.0000 0.0000 0.0000
maxdj/dx = 0.1663

Appendix D

Simulations Software: Commsim

D.1 Functions

The following tables list all the functions used for performing the simulations.

Table D.1: Modulator functions

f_gen_mcpfsk();	Generate m-*ary* CPFSK signal
osc();	Sinusoidal signal source
mapper_tilted4cpfsk();	Mapper for tilted 4-*ary* CPFSK

Table D.2: Decoder functions

f_v4cpfsk();	Viterbi decoder for 4-*ary* CPFSK
f_vmsk();	Viterbi decoder for MSK
f_f4cpfsk();	Fano decoder for 4-*ary* CPFSK

Table D.3: Demodulator functions

fsk_demod();	FSK demodulator
fsk_demodneural();	FSK neural net demodulator
cpfsk4_demod();	4-*ary* CPFSK demodulator
msk_demod();	MSK demodulator
msk_demodneural();	MSK neural net demodulator
cpfsk4_demodneural();	4-*ary* CPFSK neural net demodulator

D.2 Function Instance Initialization and Data Storage

This section describes the method used to allocate private memory to each function instance. An array of pointers to type double is declared in the main.c module (see section D.3 for information on the main.c module):

```
double *recind[MF-1];
        /* Declare an array of pointers to type double */
        /* MF = maximum number of functions */
```

The address of one of these array elements is passed to the function. When a function is called for the first time (at time = 0), it reserves a memory block of specific size for its personal use if

Table D.4: Miscellaneous functions

gaussian();	Generate Gaussian random variable
pseudoran();	Generate pseudorandom bit stream
divbyn();	Divide clock signal by 'n'
trapezium();	Integration function (trapezoidal)
symbol_delay();	Delay input by 'n' periods

necessary. It then stores the address of this memory block at the unique address it receives from the main module.

The following declaration is required in the function definition to be able to accept the address of the array element:

```
double **lrecind;
        /* Declare a pointer to a pointer to double */
```

See figure D.2 on page 77 for more detail.

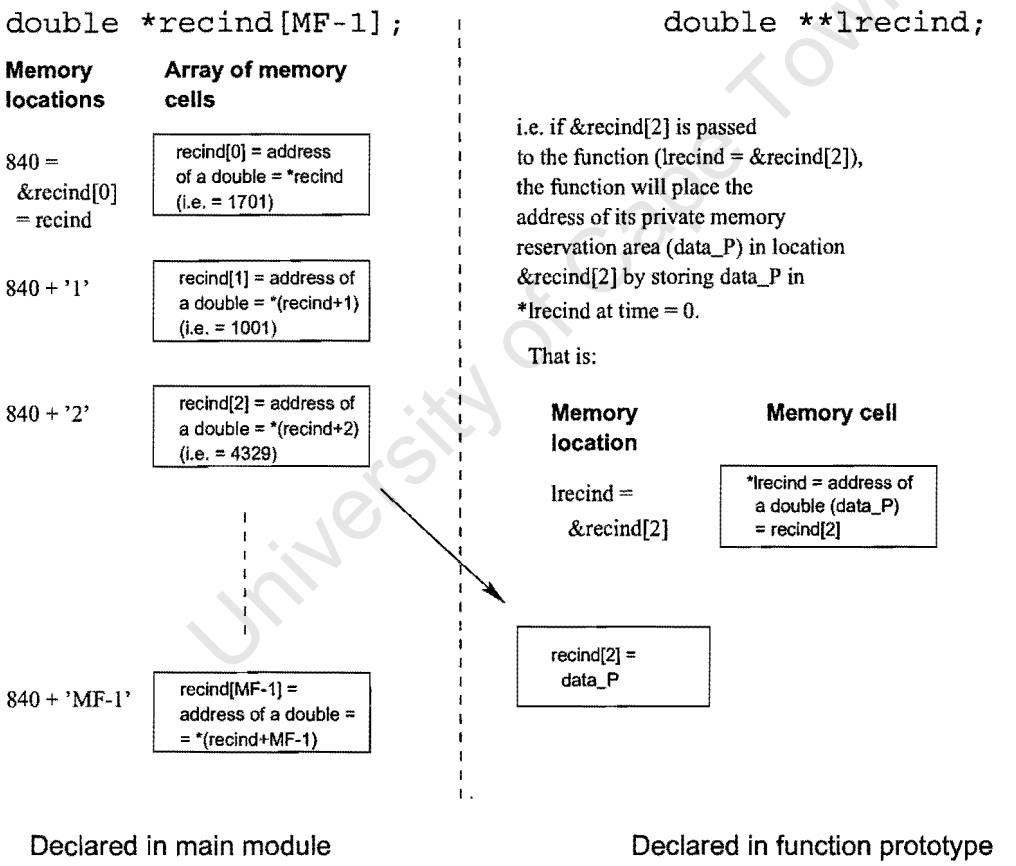


Figure D.1: Function data storage

D.3 Handling of System Time and Clock

The following listing contains extractions of the main.c file and shows the parameter declarations of those variables that maintain system time and clock, as well as the sections in which they are used to perform there work.

```
\* function main.c *\

unsigned long t, tmax;
unsigned char clk, prev_clk;
double time, tstep, clktdown, clktup, clkper;
/* t : counter used to increment system time with simulation timestep value */
/* tmax : maximum number of time steps */
/* clktdown, clktup : parameters that set the on/off ratio of the clock */

void main()
{
extern unsigned long t, tmax;
extern unsigned char clk, prev_clk;
extern double time, tstep, clktdown, clktup, clkper;

tstep = Ts/20; /* Setting simulation time step = Symbol period / x */
time = 0;
tmax = 2000;
clk = 1;
clkper = Ts;
clktdown = 0.5*clkper;
clktup = clkper;
prev_clk = (clk + 1) % 2;

for (t = 0; t <= tmax; t++)
  {
  /*****************************************************/
  /* Do function processing at each multiple of tstep */
  /* User input: */
  1. Encoding functions(vout, vin, time, tstep, &recind[xx]);
  2. Modulation functions ...
  3. Add channel perturbations ...
  4. Demodulation functions ...
  5. Decoding functions ...
  6. Data collection and output ...
  /*****************************************************/
  /* Update "time" and when required, the clock "clk" */
  prev_clk = clk;
  time = time + tstep;
  if (time >= (clktdown - 0.0000001))
    {
    clk = 0;
    clktdown = clktdown + clkper;
    }
  if (time >= (clktup - 0.0000001))
    {
    clk = 1;
```

```
        clktup = clktup + clkper;
        }

      /* Generate new Pseudo-random bit:
      Update shift registers of PRandom MLFSR,
      see fig 8-20, p. 527 Z & P */
      etc.
      }
    } /* End of function "main" */
```

Bibliography

[1] J.B. Anderson, T. Aulin, and C.E. Sundberg. *Digital Phase Modulation.* Plenum Press, 1986.

[2] T. Aulin, and C.E. Sundberg. Continuous phase modulation - Part I: Full response signaling. *IEEE Transactions on Communications*, COM-29(3):196-209, March 1981.

[3] T. Aulin, N. Rydbeck and C.E. Sundberg. Continuous phase modulation - Part II: Partial response signaling. *IEEE Transactions on Communications*, COM-29(3):210-225, March 1981.

[4] J.B. Anderson, D.P. Taylor. A Bandwidth-efficient class of signal-space codes. *IEEE Transactions on Information Theory.* Vol-IT(30):703-712, November 1978.

[5] P. G. Anderson, R.S. Gaborski, M. Ge, S. Raghavendra, M. Lung. Using Quasi-random Numbers in Neural Networks. Research Paper, FTP: ftp.cs.rit.edu, Directory: \pub\pga, File: quasi-NN.ps.

[6] J. Cuthbert. *Multi-h demodulation.* PhD thesis, University of Cape Town, 1998.

[7] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, New York (1973)

[8] M.V. Eyuboglu, S.U. Qureshi. Reduced-State Sequence Estimation with Set Partitioning and Decision Feedback. *IEEE Transactions on Communications*, Vol-36(1):13-20, January 1988.

[9] R.M. Fano. A Heuristic Discussion of Probabilistic Decoding. IEEE T*ransactions on Information Theory*, Vol-9:64-74, April 1963.

[10] G.D. Forney, JR. The Viterbi Algorithm. *Proceedings of the IEEE*, Vol-61(3):268-277, March 1973.

[11] M.S. Hodgart, B. Bohm, and J.A. Schoonees. A robust coherent receiver for MSK. Submitted to *IEEE Transactions on Communications*, June 1996.

[12] J. Huber, W. Liu. An Alternative Approach to Reduced-Complexity CPM-Receivers. *IEEE Journal on Selected Areas in Communications*, Vol-7(8):1437-1449, December 1989.

[13] S. Gish and W. Blanz. Comparing a connectionist trainable classifier with classical statistical decision analysis methods. *Res. Rep., IBM,* Almaden Res. Cen., San Jose, CA, 1989.

[14] S.E. Kritzinger. *An investigation into 4-CPFSK modulation and demodulation.* MSc thesis, University of Cape Town, 1994.

[15] R.P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, Vol-4(2):4-22, April 1987.

[16] J.L. Massey. A generalized formulation of minimum shift keying. *Conf. Rec. Int. Conf. Commun. ICC '80*, volume 2, pages 26.5.1-26.5.4 IEEE, June 1980.

[17] J.L. Massey. The how and why of channel coding. *Proceedings International Zurich Seminar*, pages 67-73, March 1984.

[18] Halvar B. Mathiesen. *Neural Network Based Multi-h CPFSK Receivers.* BSc thesis, University of Cape Town, 1997.

[19] W.P. Osborne and M.B. Luntz. Coherent and noncoherent detection of CPFSK. *IEEE Transactions on Communications*, COM-22(8):1023-1026, August 1974.

[20] Theodore S. Rappaport. *Wireless Communications.* Prentice Hall PTR, 1996.

[21] B.E. Rimoldi. *A* Decomposition Approach to CPM. *IEEE Transactions on Information Theory*, Vol-34(2):260-270, March 1988.

[22] B.E. Rimoldi. *Continuous-Phase Modulation and Coding for Bandwidth and energy Efficiency.* PhD thesis, Swiss Federal Institute of Technology Zurich,1988.

[23] B.E. Rimoldi. Design of Coded CPFSK Modulation Systems for Bandwidth and Energy Efficiency. *IEEE Transactions on Communications*, Vol-37(9):897-905, September 1989.

[24] B.E. Rimoldi. Exact Formula for the Minimum Squared Distance of CPFSK. *IEEE Transactions on Communications*, Vol-39(9):1280-1282, September 1991.

[25] R. Rosenblatt. *Principles of Neurodynamics. New York*, Spartan Books (1959).

[26] Thomas A. Schonhoff. Symbol Error Probabilities for M-*ary* CPFSK: Coherent and Noncoherent Detection. *IEEE Transactions on Communications,* COM-24:644-652, June 1976.

[27] J.A. Schoonees. *A Likelihood Ratio Analysis of Continuous-Phase Frequency Shift Keying.* PhD thesis, University of Cape Town, 1998.

[28] Stuttgart. *Stuttgart Neural Network Simulator (SNNS), User Manual Version 4.1.* Institute for Parallel and Distributed High Performance Systems (IPVR), University of Stuttgart. Report No. 6/95.

[29] C-E. Sundberg. Continuous Phase Modulation. *IEEE Communications Magazine*, Vol-24(4):25-38, April 1986.

[30] A. Svenson. Reduced State Sequence Detection of Full Response Continuous Phase Modulation. *Electronic Letters*, Vol-26(10):652-654, May 1990.

[31] Gabor C. Temes and Jack W. LaPatra. *Introduction to Circuit Synthesis and Design.* McGraw-Hill Inc. 1977.

[32] G. de Veciana, A. Zakhor. Neural Net-Based Continuous Phase Modulation Receivers. *IEEE Transactions on Communications*, Vol-40(8):1396-1408, August 1992.

[33] Xiao-an Wang and Stephen B. Wicker. An Artificial Neural Net Viterbi Decoder. *IEEE Transactions on Communications*, Vol-44(2):165-171, February 1996.

[34] John M. Wozencraft, Irwin M. Jacobs. *Principles of Communication Engineering.* Waveland Press, Inc. 1965.