Artificial Neural Network Decoding of Multi-h CPM

Prepared by:

Johan E. Kannemeyer, *B.Sc.(Eng.), Cape Town.*

Prepared for:

The Department of Electrical Engineering at the University of Cape Town.

21 February 1997

This thesis, together with an equal weighting in course work, is submitted in fulfilment of the requirements for the Degree of M.Sc.(Eng.) in Electrical Engineering.

Declaration

I hereby declare that the work contained in this thesis is my own, both in concept and execution. It is being submitted for the degree of Master of Science in Engineering at the University of Cape Town. It has not been submitted before for any degree or examination in any other university.

.....

Candidate

.....

Supervisor: Dr. R. M. Braun

Acknowledgements

I would like to thank the members of the Digital Modulation Research Group of the Department of Electrical Engineering at the University of Cape Town, and our supervisor Dr R. M. Braun, for their guidance. I would also like to acknowledge the Foundation for Research Development (FRD) for financial assistance.

Glossary and List of Symbols

ANN	artificial neural network
AWGN	additive white Gaussian noise
BPSK	biphase shift-keyed signalling
C	channel capacity
CPFSK	continuous phase frequency shift keying
CPM	continuous phase modulation
E_b	bit energy
E_s	symbol energy
E(N)	expected value of the process N
FEC	forward error correction
g(t)	multi-h frequency pulse
h or h_i	modulation index
ISI	intersymbol interference
K	number of multi-h modulation indices
M	number of symbols
max(a, b)	maximum of a and b
MSK	minimum-shift keying
N_0	single sided PSD of the AWGN
n(t)	AWGN signal
P_b	probability of a bit error
P_E	probability of a symbol error
PSD	power spectral density (Watts/Hertz)
q	modulation index denominator
q(t)	multi-h phase pulse
r(t)	received signal
s(t) or $s(t,\alpha)$	transmitted signal
T	symbol period
T_S	sampling period
VA	Viterbi Algorithm
VLSI	very large scale integration

α	a symbol sequence
β	a symbol sequence which may or may not be the same as α
σ	standard deviation of the AWGN
ϕ_0	initial phase of the multi-h carrier
$\phi(t,\alpha)$ or $\phi(t,\beta)$	phase function of the multi-h carrier
ω_c	multi-h carrier frequency in radians per second

Terms of Reference

This half thesis describes the research done by the author under the supervision of Dr. R. M. Braun. The area of interest is the artificial neural network (ANN) decoding of multi-h continuous phase modulation (CPM) schemes.

The goals were as follows:

- To understand multi-h CPM.
- To find an ANN structure suitable for decoding multi-h CPM schemes.
- To apply the ANN to the decoding of multi-h CPM schemes.
- To verify the ANN multi-h decoder through simulation.
- To hand in a report by 21 February 1997.

Synopsis

The purpose of this report is to set out the results of an investigation into the artificial neural network (ANN) decoding of multi-h continuous phase modulation (CPM) schemes. Multi-h CPM schemes offer forward error correction (FEC) capabilities for continuous transmission, digital communication systems. Multi-h CPM is reported to be a bandwidth efficient alternative to other FEC techniques such as convolutional coding, while neural networks allow for high speed decoding.

A neural network decoder was found in [12], where it had been used for the decoding of a convolutional code. This neural network structure by Xiao-an Wang and Stephen B. Wicker implements the Viterbi Algorithm (VA). All the necessary decoding information is contained in the interconnections of the ANN, and can be found by inspection of the state trellis diagram of the convolutional code. The decoder therefore requires no training. Since all the computation is done by analogue neurons and shift registers, the neural network reduces to a hybrid digital-analogue implementation of the VA. The use of analogue neurons allows the structure to be used for high data rate communications. Furthermore, the decoder is reported to be suitable for VLSI implementation.

Since multi-h CPM can be described by a phase trellis diagram, and since the VA is known to optimally detect both convolutional codes and multi-h CPM schemes, the decoder in [12], with some changes, was thought to be suitable for the decoding of multi-h CPM schemes. The first step in this thesis was to study multi-h CPM. Thereafter a functional implementation of the ANN structure in [12] was done in Borland C, and the error rate results in [12] were verified. This meant that the structure was well understood, after which it was adapted for the decoding of a multi-h CPM scheme. An error rate curve of the multi-h decoder's performance in an AWGN channel was obtained through simulation, and found to be very close to the approximation given in the literature [1].

Although the search for a trainable neural network was abandoned, a multi-h CPM decoder, with all the characteristics of the decoder in [12], was developed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

1.1 Information Theory Background

The transmission of information from one location to another is essential in today's world. A block diagram of a digital communication system is shown in Figure 1. The information is carried by different signals (each being a modulated symbol), from the transmitter over a channel to the receiver. The signal could be electrical, optical or electromagnetic, and the channel could exist in a pair of copper wires, in an optical fibre, in space, etc. All practical channels are imperfect (the channel includes the non-ideal parts of the receiver and the transmitter), causing the transmitted signals to be distorted. Imperfect channels have finite bandwidth and cause phenomena such as ISI. Practical channels are also noisy, in which case random signals are added to the transmitted signals (usually at the input stages of the receivers).

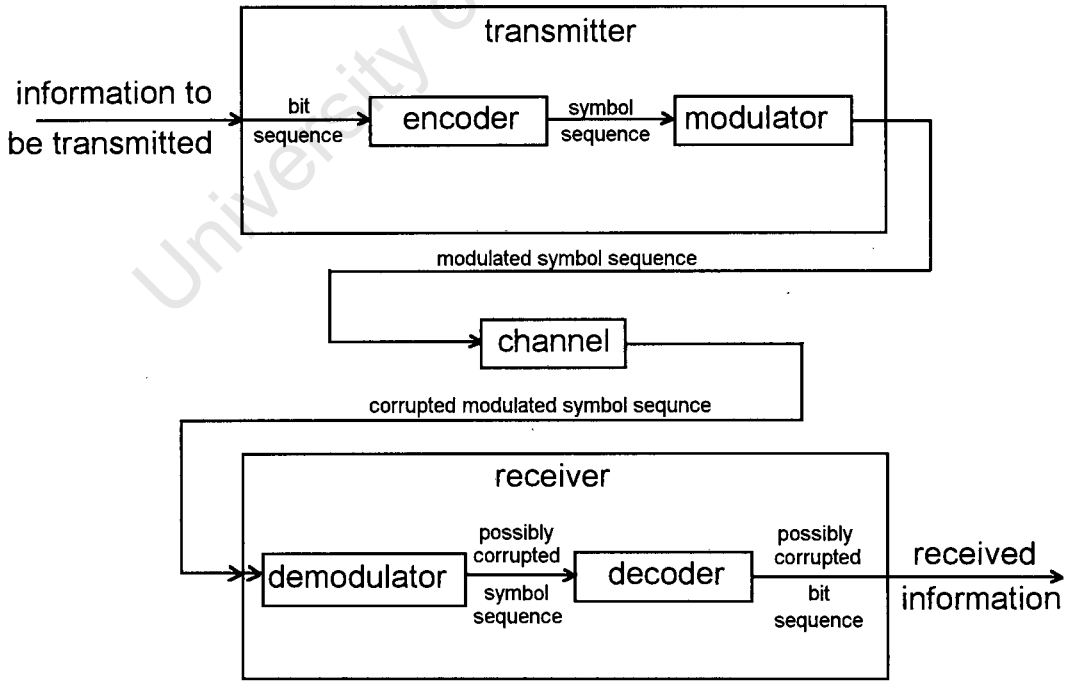


Figure 1. Digital communication system.

The corruption in a channel can be so intense that the receiver is unable to correctly identify the received signal. This results in errors (incorrect information) at the output of the receiver. One method to overcome this problem is to increase the transmitter power, so that the relative effect of the noise is reduced. This however increases operation and equipment costs, and is very seldom an adequate solution. Increased transmitter power also implies increased power in the spectral sidelobes of the signal, causing increased adjacent channel interference.

Finding power and bandwidth efficient methods of transmitting information has been the goal of many researchers over the years. The field of information theory involves a study of the relationships between transmitter power, interference, system complexity, and the probability of making detection errors, say Rodger E. Ziemer and Roger L. Peterson in [2]. An important quantity in information theory is the channel capacity, C. C is the maximum information bit rate (in bits per second) that can be achieved over a channel while maintaining an arbitrary low error rate. Shannon's capacity formula describes this upper bound, and for a binary information sequence and an AWGN channel, the formula is given by Equation 1-1 below.

$$C = B\log_2\left(1+\frac{P}{N_0 B}\right) \tag{1-1}$$

In Equation 1-1, B is the transmission bandwidth, i.e. the bandwidth of the channel, P is the received signal power, and N_0 is the single sided noise power spectral density (PSD). Equation 1-1 implies that if a binary information sequence is transmitted at R_m bits per second, and $R_m < C$, then the error rate can be made as small as desired by using forward error correction (FEC) techniques, without increasing the transmitter power above the value for which C was calculated.

Multi-h CPM and convolutional coding are two FEC techniques. In both these techniques an encoder is used to transform the actual information bit sequence into a sequence of symbols, as shown in Figure 1. Each symbol is one of a finite set. Each symbol interval the encoder can assume one of a finite number of states, and each state

state has an associated output symbol. The encoder operation is such that given the current state and input, only one particular next state is allowed. In this manner only certain encoder state sequences, and hence output symbol sequences, are allowed.

The decoder on the receiver side follows an allowed state sequence, which best matches the state sequence derived by the demodulator from the noisy received signal. From this state sequence an output is produced, which was the most likely input to the encoder. Multi-h CPM and convolutional coding will be discussed in more detail in Chapters 2 and 3 respectively.

In order to visualise the effect of these FEC techniques on C, consider Equation 1-1. The power of the received signal (P) is equal to the energy of the received signal per bit (E_b), multiplied by the transmission bit rate (R_m), i.e.

$$P = E_b \, R_m . \tag{1-2}$$

Then Equation 1-1 can be written as

$$\frac{C}{B} = \log_2\left(1 + \left(\frac{E_b}{N_0}\right)\left(\frac{R_m}{B}\right)\right). \tag{1-3}$$

If we set R_m equal to its maximum, i.e. $R_m = C$, and plot E_b/N_0 versus R_m/B, then Figure 2 is obtained.

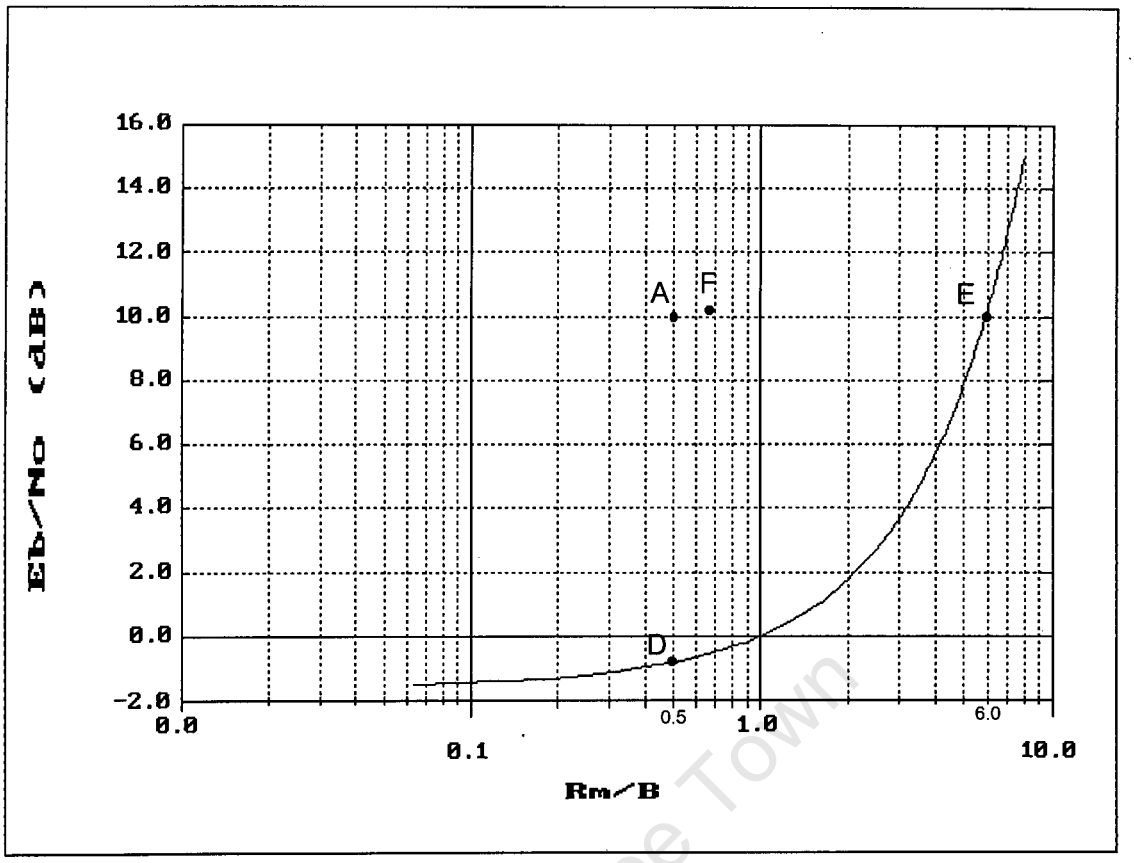


Figure 2. Power bandwidth trade-off for error-free transmission through noisy, band-limited channels.

For a given R_m/B in Figure 2, the corresponding E_b/N_0 is the minimum value for which an arbitrarily low error rate can be achieved. Note that all points to the left of the curve are points for which these arbitrarily low error rates are possible.

Consider the following example: Assume that BPSK is used to transmit information over a band-limited channel and that the required probability of error at the decoder output is less than 3.90×10^{-6}. The probability of error for BPSK is given by Equation 1-4. Note that for BPSK, the bit and symbol energies are equal since only one bit is transmitted per symbol.

$$P_E = Q\left(\sqrt{\frac{2E_s}{N_0}}\right) = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \tag{1-4}$$

The Q-function used in Equation 1-4 is given by Equation 1-5.

$$Q(x) = \int_{x}^{\infty} \frac{e^{-t^2/2}}{\sqrt{2\pi}} dt . \tag{1-5}$$

From the Q-function tables in [22] by F. G. Stremler, it was found that

$$Q\left(\sqrt{2 \times 10}\right) \approx 3.90 \times 10^{-6}. \tag{1-6}$$

It then follows that the required E_b/N_0 for uncoded BPSK is 10, which is equal to 10 dB. According to [2], $R_m/B = 0.5$ for BPSK. The operating point in question is then at A in Figure 2, which is to the left of the limiting curve. Shannon's formula says that this operating point can be moved closer to the limit by using convolutional coding. If E_b/N_0 is kept constant at 10 dB, the operating point can be moved towards the point labelled E in Figure 2, and R_m/B can be increased to approximately 6. This results in more efficient use of bandwidth. If on the other hand R_m/B is kept equal to 0.5, the operating point can be moved towards the point labelled D, and E_b/N_0 can be reduced to less than 0 dB. The amount by which E_b/N_0 is reduced, in this case for BPSK by using convolutional coding, is known as the coding gain.

In multi-h the coding gain is quoted relative to MSK. MSK is a special case of CPFSK, and CPFSK is a subclass of multi-h CPM in which only one h-value is used. The probability of error formula for MSK is given by Equation 1-7.

$$P_E = 2Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \tag{1-7}$$

For a probability of error of 3.90×10^{-6}, the required E_b/N_0 for MSK is 10.28 dB. According to [2], the null-to-null bandwidth of MSK is $1.5R_m$, and therefore R_m/B is 0.667. This point occurs at F in Figure 2.

Both MSK and multi-h CPM are coded modulation schemes, i.e. the coding and modulation is done simultaneously. In convolutional coding, the coding and modulation are independent of each other, and therefore the coding gain is expressed relative to the uncoded modulation scheme being used. Note that in both multi-h and convolutional coding techniques, the coding gain is dependent on the actual code used, and that Shannon's formula indicates nothing about how to find good codes. Shannon's formula only describes the channel capacity limits. Note also that the coding gain is achieved at the cost of increased transmitter and receiver complexity.

Multi-h CPM is reported to be a "bandwidth-efficient alternative to other coding techniques", by Stephen G. Wilson and Richard C. Gaus in [17], and hence our interest. This document focuses on the decoding process, and in particular on the neural network decoding of multi-h CPM schemes. Finding good multi-h sets and describing the power spectra of the signals are outside the scope of this thesis.

1.2 Artificial Neural Networks (ANN's)

An artificial neural network is a network of interconnected nodes, as illustrated in Figure 3. The nodes are called neurons and are connected by branches. There may be several network inputs, several network outputs, and several layers of neurons (also called hidden layers), in the neural network.

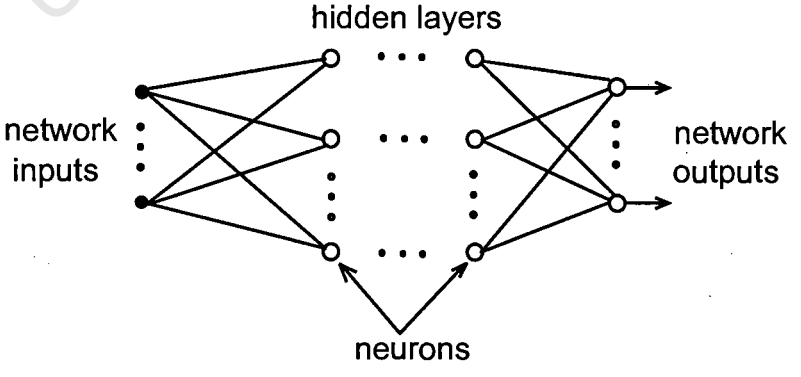


Figure 3. A neural network.

Note: The direction of flow of information is from left to right.

A single neuron is shown in Figure 4. x_i is the i-th input to the neuron and w_i is its weight.

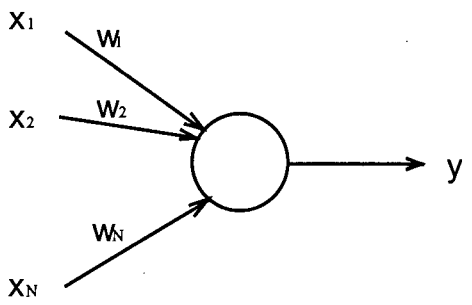


Figure 4. A neuron.

The output of each neuron is a function, linear or non-linear, of the total weighted input to that neuron, and is given by

$$y = f\left(\sum_{i=1}^{N} w_i x_i\right). \tag{1-8}$$

All the neurons in a neural network may or may not implement the same function, and each weight can have any finite value. Any combination of neural network inputs and neuron outputs may constitute the input to a neuron. The most common neural network is however one in which the network inputs are connected only to the first layer of neurons, and only connections in a forward direction are allowed, i.e. from left to right. The output of the neural network is then some function of its inputs.

The desired function may be to classify an input pattern as a specific pattern in a set of patterns. It is known that once a neural network has been constructed to classify patterns, it often also correctly classifies slight variations in the patterns. For example: Let a pattern set consist of only two patterns: 1111 and 0000. Assume that 1111 is transmitted over an imperfect channel, and that due to the noise, the output of the demodulator is 1011. Then there may exist a neural network that will classify 1011 as 1111, because 1011 is not an allowed pattern, and 1011 seems more like 1111 than 0000.

Recall that, given the state of the encoder and a certain input sequence, only one symbol sequence is possible at the output of the encoder. Different input sequences will result in different output sequences. If corruption takes place during the transmission of a symbol sequence, the symbol sequence (or pattern) at the output of the demodulator could be incorrect, i.e. not be an allowed symbol sequence. By storing the last d symbols of the sequence at the output of the demodulator, an ANN input pattern of length d is obtained every symbol period. It is thought to be possible to construct an ANN decoder that could correctly classify symbol sequences that are in error, and hence correct the errors introduced by the channel. Since certain neural networks can be implemented with analogue neurons, and since analogue neurons can perform their operations at very high speeds, the result could be a high speed decoder.

How can such an ANN be constructed? How many neurons, hidden layers, inputs and branches are needed, and how can the weights of these branches be found? It is unnecessary, for the purpose of understanding this thesis, to know the answers to these questions. The reasons will become clear in Chapter 3. It is however important to know that the method of finding the weights is known as training, and training a neural network can be a lengthy process. For a comprehensive study of neural networks, see [19] by S. Haykin or [20] by J. Hertz *et al.*

A literature search has produced an artificial neural network implementation of the Viterbi Algorithm (VA). The VA is described by G. D. Forney in [18]. The ANN (by Xiao-an Wang and Stephen B. Wicker, authors of [12]) is used for the decoding of a convolutional code. It is known that both multi-h CPM and convolutional codes are optimally decoded by the VA. It is therefore expected that the structure used in [12] can be adapted for the decoding of multi-h CPM schemes.

Chapter 2 describes the multi-h signalling format. The ANN structure in [12] will be analysed in Chapter 3, and it will be adapted for the decoding of a multi-h CPM scheme in Chapter 4. Simulation procedures for verifying the decoding structures will be described in Chapter 5, and error rate results will be presented. In Chapter 6 the results will be analysed and conclusions will be drawn.

2. Multi-h Continuous Phase Modulation (CPM)

Multi-h is a constant envelope continuous phase modulation scheme. As mentioned in Chapter 1, multi-h is considered to be a bandwidth efficient alternative to convolutional coding. This chapter will show how the information carrying phase of the multi-h signal can be represented by a phase trellis diagram, and the reader will be referred to the literature for studies of the power spectra of multi-h schemes. The three types of synchronisation required for the decoding of multi-h CPM will be discussed briefly, and it will be shown how to optimally detect multi-h CPM in an AWGN channel. An estimate of the error rate performance will then be given.

2.1 Multi-h Signal Description

The transmitted signal is a constant amplitude sinusoidal signal. The information is carried in the phase of the signal, as described by Equation 2-1 [2].

$$s(t,\alpha)=\sqrt{\frac{2E_s}{T}}\cos(\omega_c t+\phi\ (t,\alpha)+\phi_0) \tag{2-1}$$

In Equation 2-1, E_S is the symbol energy, T is the symbol period, ω_c is the carrier frequency in radians per second, and ϕ_0 is the arbitrary initial phase of the carrier. $\phi(t,\alpha)$ is the information carrying phase and is given by Equation 2-2.

$$\phi\ (t,\alpha)=2\pi\sum_{i=-\infty}^{\infty}h_i\alpha_i\,q(t-iT) \tag{2-2}$$

To see how the information is carried in the phase of the sinusoidal signal, consider the following:

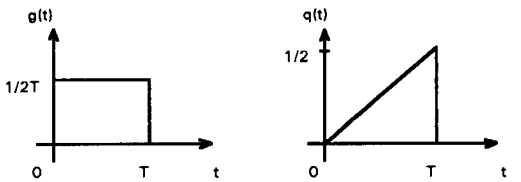
- In Equation 2-2, α_i is the transmitted symbol. According to James Cuthbert, author of [11], $\alpha_i \in \{\pm1, \pm3,..., \pm(M-1)\}$ for even M, and $\alpha_i \in \{0, \pm2, \pm4,..., \pm(M-1)\}$ for odd M, where M is the number of different symbols. In this thesis only binary multi-h schemes will be considered, i.e. M = 2 and $\alpha_i \in \{\pm1\}$.

- h_i is the modulation index of the i-th symbol interval, and is selected in a periodical fashion from a set $H_K = \{p_1/q, p_2/q, ..., p_K/q\}$. q is a constant. For example: If $H_2 = \{1/4, 2/4\}$, then q = 4, K = 2, and $\{h_i\} = \{1/4, 1/2, 1/4, 1/2, ...\}$. It is convenient at this point to introduce the difference between symmetrical and asymmetrical multi-h schemes. In symmetrical multi-h schemes, h_i is independent of the symbol to be transmitted (as in the above example). In asymmetric multi-h schemes, h_i is dependent on the symbol to be transmitted [11]. For each of the M symbols in the asymmetrical scheme, there is a multi-h set of length K. Consider the binary case (M = 2) where either a +1 or a -1 is to be transmitted, and let K = 3. Let the multi-h sets be {a/q, b/q, c/q} for a +1, and {A/q, B/q, C/q} for a -1. Then h_i is selected from the set associated with the symbol to be transmitted, and according to the position in the sequence (modulo-K). For example: If the data sequence is {+1,+1,+1,-1,-1,-1,+1,-1,+1,-1, ...}, then $\{h_i\}$ = {a/q, b/q, c/q, A/q, B/q, C/q, a/q, B/q, c/q, A/q, ...}.

- In Equation 2-2, q(t) is called the phase pulse. It is a continuous function of time and describes the manner in which the phase changes during a symbol interval. Related to q(t) is the frequency pulse g(t), and this relationship is described by Equation 2-3. g(t) describes the manner in which the instantaneous frequency of the transmitted signal changes, and is time limited to $0 \leq t \leq LT$. Many different frequency and phase pulse pairs are described in the literature, for example in [4] by Carl-Eric Sundberg. Figure 5 illustrates the rectangular frequency pulse g(t), and its corresponding q(t). Note that q(t) is normalised so that q(LT) = 1/2, in which case, if $\phi_0 = 0$ and L = 1, the phase at the end of each symbol interval is a multiple of π/q [2]. The value of L divides CPM into full response CPM and partial response CPM. In full response CPM L = 1, as described in [9] by T. Aulin and Carl-Eric W. Sundberg. In partial response CPM L > 1, as described in [10] by

T. Aulin, N Rydbeck and Carl-Eric W. Sundberg. Only full response symmetrical multi-h schemes will be considered in the remainder of this document.

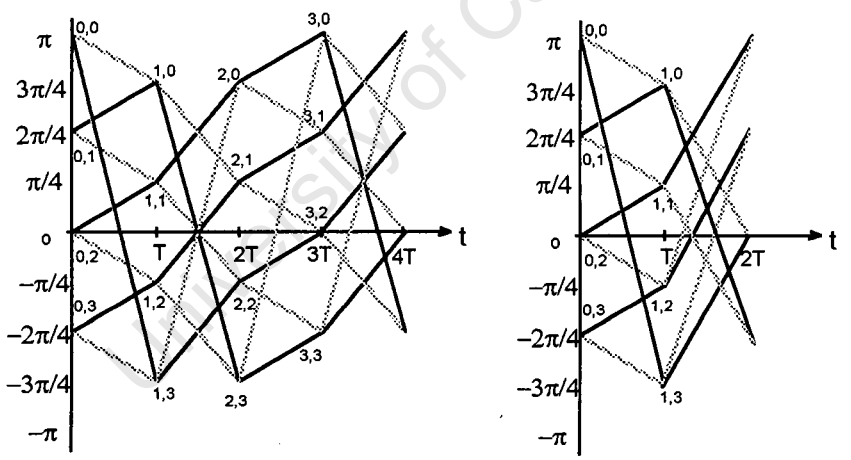
$$q(t) = \int_{-\infty}^{t} g(a)da\,. \tag{2-3}$$



(i) (ii)

Figure 5. (i) Rectangular frequency pulse, (ii) Corresponding phase pulse.

For rational values of h, the phase transitions of the transmitted signal can then be described by a finite state phase trellis diagram (modulo 2π), as shown in Figure 6.



(i) (ii)

Figure 6. Phase trellis diagram for the phase pulse used in Figure 5, with (i) H_2 = (1/4, 2/4), (ii) H_2 = (1/4, 3/4).

In Figure 6, α_i is +1 for the dark lines and -1 for the lighter lines. In the first symbol interval of Figure 6(i), h_i =1/4 and the phase of the carrier changes by $\pm\pi/4$ for α_i =

± 1. In the second symbol interval $h_i = 2/4$, and the phase of the carrier changes by $\pm 2\pi/4$ for $\alpha_i = \pm 1$.

If the phase trellis has a finite number of states, the Viterbi Algorithm (VA) can be used for maximum likelihood detection of the transmitted symbol sequence, and the number of states in the Viterbi decoder is q, according to I. Sasase and S. Mori in [1]. Both q and the period of the trellis diagram influence the complexity of the decoder. The period of the trellis diagram can be found by considering the multi-h set $H_K = (p_1/q, p_2/q, ..., p_K/q)$, in which $h_i = p_i/q$ and i is modulo-K. Let

$$\Gamma = \sum_{i=1}^{K} p_i . \tag{2-4}$$

- If Γ is odd (e.g. in Figure 6(i) T = 1 + 2 = 3), the period of the trellis is 2K symbol intervals and the number of states in a trellis period is 2qK.
- If Γ is even (e.g. Figure 6(ii) T = 1 + 3 = 4), the period of the trellis is K symbol intervals and the number of states in a trellis period is qK.

One last term that needs introduction is the constraint length of a multi-h set. Let X be the minimum number of intervals over which two paths, that have the same origin in the trellis diagram, remain unmerged. For a set of modulation indices H_K (of length K), there exists a certain set such that X = K, provided that $q \geq M^K$ and that no subset of H_K has an integer sum [1]. X + 1 is then the constraint length of the code.

2.2 Multi-h Power Spectra

The power spectra of multi-h schemes are described, amongst others, in [2], [3] and [17].

2.3 Synchronisation Types Required for Decoding Multi-h

Three types of synchronisation are required for decoding multi-h CPM. These are carrier phase, symbol interval (T), and superbaud synchronisation. Superbaud synchronisation keeps track of the position in the trellis and is modulo-K. A q-th power-law device can be used to extract the timing information from the received signal, as described in [3]. In depth studies of carrier synchronisation are reported in [5] by John M. Liebetreu, and in [6] by Brian A. Mazur and Desmond P. Taylor.

2.4 Maximum Likelihood Detection of Multi-h

In practice the received signal will be perturbed by noise. If the noise is AWGN with zero mean, then the symbol sequence, that corresponds to the phase path in the trellis which is closest to the received signal phase path (in an Euclidean distance sense), is the symbol sequence most likely to have been transmitted, say John G Proakis and Masoud Salehi in [13]. It is known that the VA is optimal in finding that path, and hence in finding the most likely transmitted symbol sequence.

When using the VA, the received signal is observed over a period of d symbol intervals, after which an output is produced. d is known as the decision (or decoding) depth, and a rule for selecting a value for d will be given in Chapter 3. If $E_S = 0.5$ Joule and T = 1 second, then Equation 2-1 can be written as

$$s(t,\beta) = \cos(\omega_c t + \phi(t,\beta) + \phi_0). \qquad (2\text{-}5)$$

The above equation then represents the noise free transmitted signal for the symbol sequence β. Let r(t) be the noisy received signal, i.e. r(t) = s(t, β) + noise. It is reported in [13] that the probability of error at the receiver output is minimised by selecting the symbol sequence α, at the receiver, such that α maximises the correlation between s(t, α) and r(t). Note that α = β when no transmission errors occur. The correlation value is calculated as shown in Equation 2-6.

$$C_i(t,\alpha) = \int_{-\infty}^{(i+i)T} r(t)\cos(\omega_c t + \phi\ (t,\alpha) + \phi_0)dt \quad [13] \tag{2-6}$$

Note that the correlation value between s(t,α) and r(t) from minus infinity to time (i+1)T, is equal to the correlation value between s(t,α) and r(t) from minus infinity to time iT, plus the correlation value between s(t,α) and r(t) from time iT to time (i+1)T. Equation 2-6 can therefore be written as

$$\begin{aligned} C_i(t,\alpha) &= C_{i-1}(t,\alpha) + B_i(t,\alpha) \\ &= C_{i-1}(t,\alpha) + \int_{iT}^{(i+i)T} r(t)\cos(\omega_c t + \phi\ (t,\alpha) + \phi_0 + \phi_i)dt, \end{aligned} \tag{2-7}$$

where ϕ_i is the phase of the carrier at time iT. $B_i(t,\alpha)$ is known as the branch metric and is the correlation value over the last symbol interval. Note that there is one branch metric for each branch in the trellis during any given symbol interval. For each of the q starting states there is a ϕ_i, and in the binary case there are two branches emanating from each state. Hence there are 2q branch metrics per symbol interval. For example: If q = 4, there are 2 × 4 = 8 branch metrics per symbol interval. Figure 7 illustrates how a branch metrics can be found.

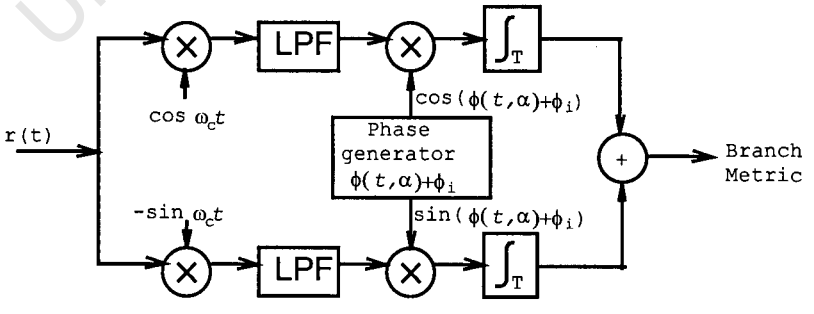


Figure 7. Calculation of multi-h branch metrics [13].

The structure in Figure 7 can be verified as follows:

$$
\begin{aligned}
&\int_{iT}^{(i+1)T} r(t)\cos(\omega_c t)\cos(\phi(t,\alpha)+\phi_i)dt - \int_{iT}^{(i+1)T} r(t)\sin(\omega_c t)\sin(\phi(t,\alpha)+\phi_i)dt \\
&= \int_{iT}^{(i+1)T} r(t)\left[\cos(\omega_c t)\cos(\phi(t,\alpha)+\phi_i) - \sin(\phi(t,\alpha)+\phi_i)\sin(\omega_c t)\right]dt \\
&= \int_{iT}^{(i+1)T} r(t)\frac{1}{2}\{\cos(\omega_c t+\phi(t,\alpha)+\phi_i)+\cos(\omega_c t-\phi(t,\alpha)-\phi_i) \\
&\qquad -\cos(\omega_c t-\phi(t,\alpha)-\phi_i)+\cos(\omega_c t+\phi(t,\alpha)+\phi_i)\}dt \\
&= \int_{iT}^{(i+1)T} r(t)\cos(\omega_c t+\phi(t,\alpha)+\phi_i)dt.
\end{aligned}
\tag{2-8}
$$

An alternative structure has been used by Al-Nasir Premji and Desmond P. Taylor in [7].

It is important to note that all of the 2q functions described by Equation 2-5, also known as the basis functions, are not orthogonal over any symbol interval, and hence there will be some correlation between the outputs of the 2q correlators.

$C_{i-1}(t,\alpha)$ in Equation 2-7 is called the path metric, and is equal to the sum of all the branch metrics along the surviving path that ends at some state at time iT. The surviving path at a state is that path through the trellis (from some initial state) for which the path metric is a maximum. Note that at time iT there are q states, each with its own path metric. The VA is used to ensure that the correct path is selected as the surviving path at each state, and is implemented as follows:

1) Find the path metric of each path entering each state at time (i+1)T, i.e. $C_i(t,\alpha)$, by adding to the branch metric of the branch connecting the state to its predecessor, i.e. $B_i(t,\alpha)$, the path metric of the state's predecessor, i.e. $C_{i-1}(t,\alpha)$.
2) For each state at time (i+1)T, select the incoming path with the maximum path metric as the surviving path, and store its path metric as the path metric for that state. Also store the state sequence or encoder input sequence associated with that state transition.

3) Increment i by 1, and repeat steps 1 and 2 above until the decision depth (d) has been reached. (i = 0 initially.)

At a depth d from the starting state, the decoder state with the maximum path metric is then the most likely state that the encoder would have been in after d symbol intervals. The symbol sequence of the trellis path leading to that state is then the most likely symbol sequence to have been transmitted, and the encoder input that would have caused the first transition along this path is then the output of the decoder. The first state along the surviving path is then the next initial state, and the process (1 to 3 above) is repeated.

If the state transitions are stored, the path can be traced back from the state with maximum path metric to its initial state, and the decoder output can be found. Another approach is to let each one of the q states have an associated register of length d. Each register contains the encoder input sequence for the surviving path terminating at that state. As decoding proceeds, the contents of each register is updated every symbol period in the following manner: After the q surviving paths have been selected, the contents of each register is changed to that of its predecessor. The contents of each register is then shifted by one position so that the oldest input is shifted out. Of these, the output associated with the state with maximum path metric becomes the output of the decoder. The encoder input that would have caused the last state transition for each surviving path, is then inserted into the first (or newest) position of that path's register. In this manner, each register contains the encoder input sequence of the last d symbol intervals for the surviving path ending at that state. This is called the register exchange method.

2.5 Probability of Error for Multi-h

When choosing an h-set to minimise the error rate, the h-values in the set are chosen to delay the merging of two paths in the phase trellis diagram which start at the same state. This is done to maximise what is called the minimum free distance (d_{min}) of the

code. The error correction capabilities of a code increases with d_{min}, which in turn reduces the transmitter power required to achieve a given probability of error for a given bit rate. In general, the actual probability of error formula for optimally detected multi-h CPM in an AWGN channel can not be found [3]. The following estimate was taken from [1].

Let $s(t,\alpha)$ and $s(t,\beta)$ be two signals corresponding to two different symbol sequences (α and β) that split apart at $t = iT$, and let $\phi(t,\alpha)$ and $\phi(t,\beta)$ be their respective phase trajectories. Then the square of the Euclidean distance (D^2) between these two signals, observed over d symbol intervals, is

$$\begin{aligned} D^2 &= \sum_{i=0}^{d-1} \int_{iT}^{(i+1)T} [s(t,\alpha) - s(t,\beta)]^2 \, dt \\ &= \frac{2E_s}{T} \sum_{i=0}^{d-1} \int_{iT}^{(i+1)T} [1 - \cos(\phi(t,\alpha) - \phi(t,\beta))] dt. \end{aligned} \tag{2-9}$$

Let D_{min}^2 be the smallest D^2 when considering all possible pairs of symbol sequences. The square of the minimum free distance is then

$$d_{min}^2 = \frac{D_{min}^2}{2E_b}. \tag{2-10}$$

The bit energy (E_b) and the symbol energy (E_S) are related as follows:

$$E_b = \frac{E_s}{\log_2(M)}. \tag{2-11}$$

The probability of error for large E_b/N_0 in an AWGN channel with a single sided PSD of N_0 is then given by Equation 2-12.

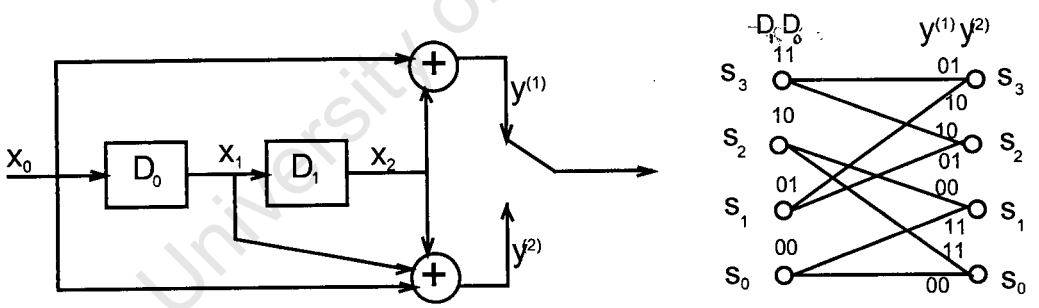
$$P_E \approx Q\left(\sqrt{\frac{d_{min}^2 E_b}{N_0}}\right) \tag{2-12}$$

In general, d_{min}^2 increases with q and proper selection of H_K. However, since the number of states in the trellis (and hence the Viterbi decoder complexity) increase with q, small q-values are desired. The ultimate goal is to get as close to the Shannon limit as possible by proper selection of H_K, while keeping the transmitter and receiver as simple as possible.

3. An Artificial Neural Network (ANN) Viterbi Decoder

This chapter is a report on a study of [12]. In [12] an ANN implementation of the Viterbi Algorithm (VA) is used for the decoding of a convolutional code. Since all the decoding intelligence is contained in the interconnections between the neurons of the neural network, and since these interconnections can be found by inspection of the trellis diagram of the code, this ANN requires no training. All the computation is done by analogue neurons, and in parallel, to allow for very high speed decoding. It is also reported in [12] that this structure is suitable for VLSI implementation. It is expected that this ANN structure can be adapted for the decoding of a multi-h CPM scheme, since both convolutional and multi-h codes are known to be optimally decoded by the VA.

3.1 The Convolutional Encoder and State Trellis Diagram



(i) (ii)

Figure 8. (i) Convolutional encoder, (ii) Trellis diagram [12].

Shown in Figure 8(i) is a convolutional encoder. The two squares are single bit shift registers (delay elements). Every symbol interval the bits at x_0, x_1 and x_2 are shifted one position to the right, and a new input bit is applied at x_0. From these three bits, two output bits, $y^{(1)}$ and $y^{(2)}$, are produced. Every symbol interval one bit enters the encoder and two output bits are produced. It is therefore a rate-1/2 encoder. The

constraint length of the code is equal to the number of intervals over which one input bit can influence the output, and in this case it is 3.

Figure 8(ii) shows the trellis diagram of the encoder. The encoder states are represented by circles. The state of the encoder is given by the contents of the encoder registers. The contents of each encoder register is indicated above the circles. The dark lines between states describe state transitions for an encoder input of 1, while the lighter lines are for an input of 0. The output of the encoder, for each allowable state transition, is shown adjacent to the branch representing that state transition.

The output is the convolutional sum of the input sequence and the impulse response of the encoder. Hence the term "convolutional coding". Let the input bit sequence be (x_0, x_1, x_2, ...). The impulse response of that part of the encoder which leads to the $y^{(1)}$ output is ($g_0^{(1)}$, $g_1^{(1)}$, $g_3^{(1)}$, ... $g_C^{(1)}$). $g_i^{(1)}$ is 1 if there is a connection form x_i to the modulo-2 adder leading to $y^{(1)}$, and 0 otherwise. In Figure 8(i), ($g_0^{(1)}$, $g_1^{(1)}$, $g_2^{(1)}$) = (1,0,1). Similarly the impulse response of that part of the encoder which leads to the $y^{(2)}$ output is ($g_0^{(2)}$, $g_1^{(2)}$, $g_2^{(2)}$) = (1,1,1). Then the output of the top and bottom branches are, according to S. Haykin in [21],

$$y_i^{(1)} = \sum_{k=0}^{C} x_k^{(1)} g_{i-k}^{(1)} \quad i = 0,1,2,... \tag{3-1}$$

and

$$y_i^{(2)} = \sum_{k=0}^{C} x_k^{(2)} g_{i-k}^{(2)} \quad i = 0,1,2,... \tag{3-2}$$

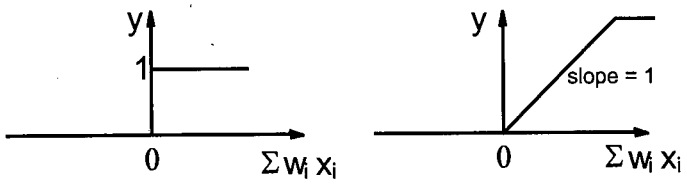
respectively, and the combined output is then

$$\{y_i\} = \{y_0^{(1)}, y_0^{(2)}, y_1^{(1)}, y_1^{(2)}, ...\} . \tag{3-3}$$

The reader is referred to [21] for examples.

3.2 Neurons Used in the ANN Decoder

Two types of neurons are used in the ANN decoder, namely hard-limiter neurons (HL-neurons) and threshold-logic neurons (TL-neurons). Their respective functions are shown in Figure 9.



(i) (ii)

Figure 9. Non-linear functions for the neurons used in [12]: (i) HL-neuron, (ii) TL-neuron.

Note that all TL-neurons operate in the linear region. In this document the output of an HL-neuron is 0 when the total input is smaller or equal to 0, and 1 otherwise.

3.3 The ANN Convolutional Decoder

Figure 10 shows a block diagram of the ANN convolutional decoder. It implements the VA as described in Chapter 2.4. The surviving path metrics of the previous symbol interval are fed back from the output of the surviving path selection unit. The new branch metrics are calculated by correlators, and added to the surviving path metrics of the previous symbol interval. The respective additions are determined from the trellis diagram, and from these values the new surviving path metrics are selected. The register exchange method is used to keep track of the encoder input sequences for the surviving paths. Every symbol interval the maximum surviving path metric is selected, and the oldest value in its register becomes the output of the decoder.

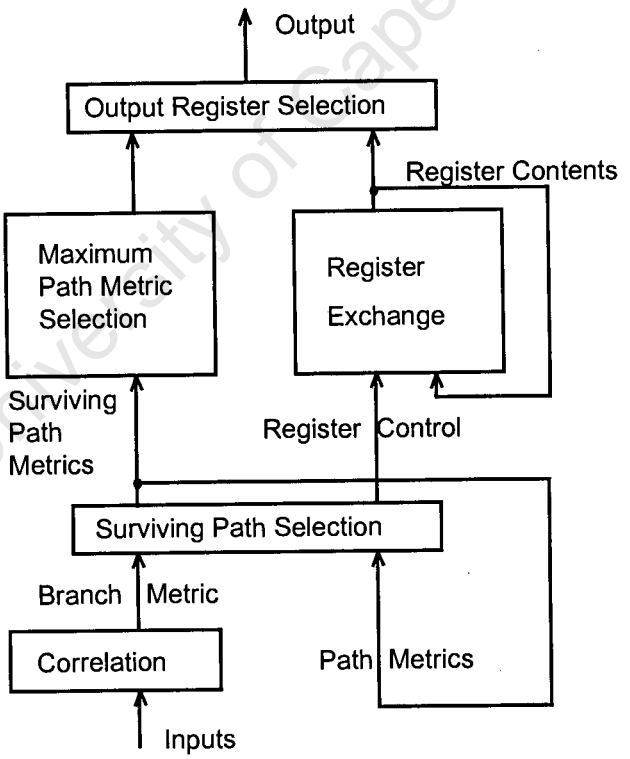


Figure 10. Block diagram of an ANN decoder [12].

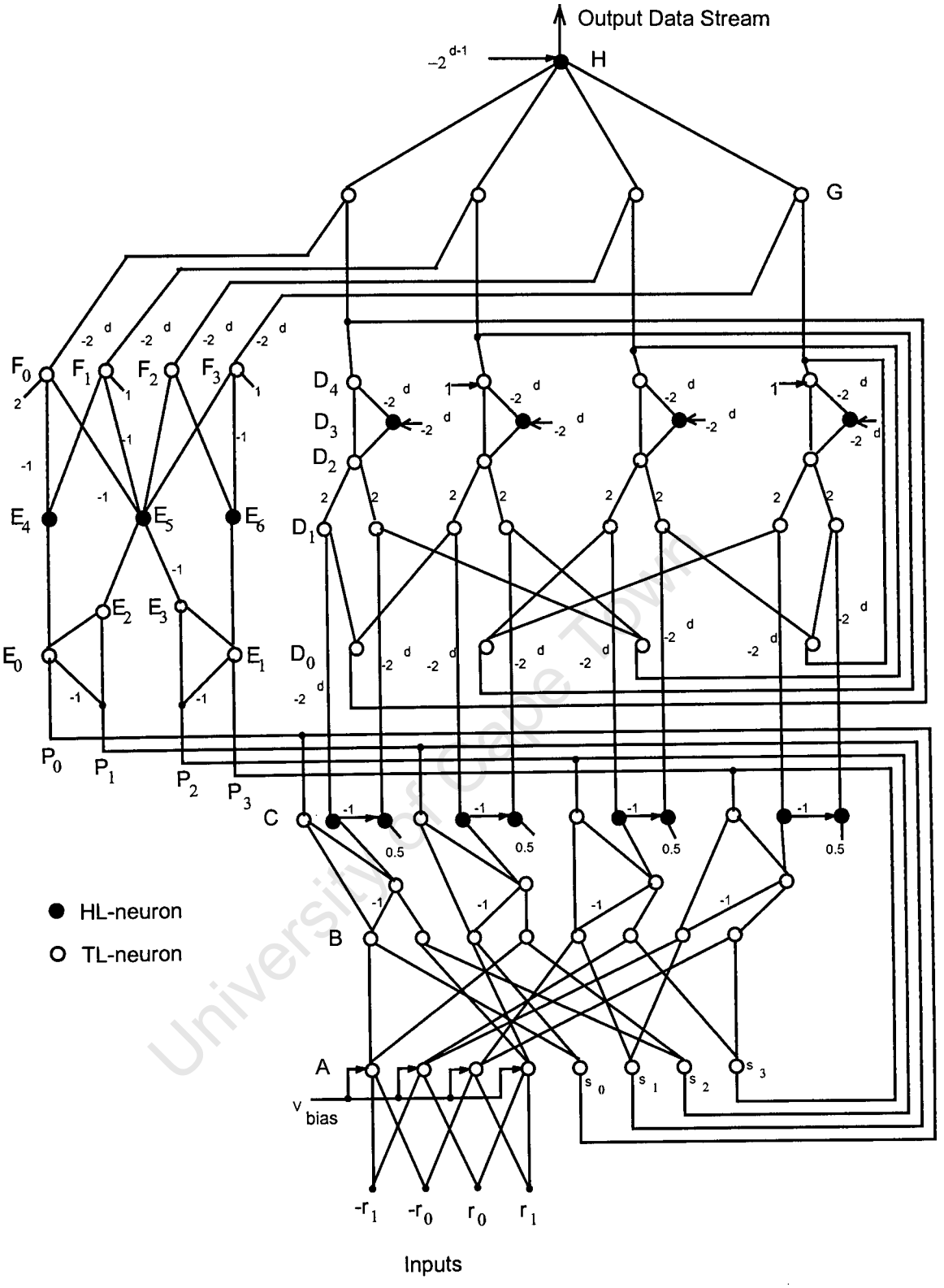


Figure 11. An ANN Viterbi decoder for a rate-1/2, K = 3 convolutional code [12].

Note : The direction of the flow of information is upwards where not specified.

Branches with no weights have weights of 1.

Figure 11 shows how Figure 10 is implemented using the HL- and TL-neurons. The various operations of the ANN decoder are described in detail in the following sections.

3.3.1 Computing the Branch Metrics

The encoder in Figure 8(i) is a rate-1/2 encoder and therefore two bits are transmitted for each encoder input bit. Let the modulator output (refer to Figure 1) be +1 or -1 when the output of the encoder is 1 or 0 respectively. This is known as antipodal baseband. The transmitted signal pair (y_0, y_1) can then be found by replacing the 0's adjacent to the branches of Figure 8(ii) by -1's. The resulting trellis diagram is shown in Figure 12.

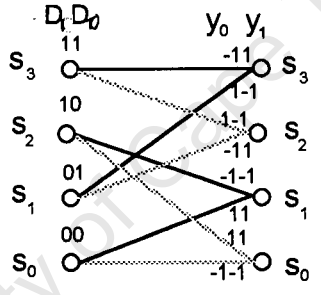


Figure 12. Trellis diagram for the convolutional encoder in Figure 8(i), showing adjacent to the branches, the antipodal baseband transmitted signals.

In an AWGN channel, the received signal pair (r_0, r_1) is equal to the transmitted pair (y_0, y_1) plus the AWGN. The maximum likelihood branch metrics used by the VA are then the inner products of the received signals and the branch labels in Figure 12. The four layer A TL-neurons in Figure 11 compute the eight branch metrics as follows:

-1.r0 + -1.r1 = -r0 - r1 = branch metric from S0 to S0 and from S2 to S1
1.r0 + -1.r1 = r0 - r1 = branch metric from S3 to S2 and from S1 to S3
-1.r0 + 1.r1 = -r0 + r1 = branch metric from S3 to S3 and from S1 to S2
1.r0 + 1.r1 = r0 + r1 = branch metric from S0 to S1 and from S2 to S0 (3-4)

Recall that all the TL-neurons must operate in the linear region. This means that the total input to each TL-neuron must always be positive. Since some of the branch metrics (as calculated above) are negative, a constant bias (V_{bias}) is added to each branch metric before the TL-function is applied. This ensures that the total input to each TL-neuron is always positive.

3.3.2 Surviving Path Selection

Each state in the trellis has two entering paths. The new path metric of each path entering a state is calculated by adding the path metric of the previous state to the branch metric of the branch connecting the two states. These additions are performed by the eight layer B TL-neurons in Figure 11. The path metrics of the previous symbol interval are stored by the TL-neurons labelled S0 to S3 . For each state, the entering path with maximum path metric must be chosen as the surviving path for that state. The selection of the surviving path is done in the following manner: Let p be the maximum path metric selected from path metrics a and b. Then

$$p = \max(a, b) = a + \max(b-a, 0). \tag{3-5}$$

Proof: If $a > b$, $p = a + \max(\text{a negative value}, 0) = a$, and if $a < b$, $p = a + b - a = b$.

Now note that the output of a TL-neuron can be written as the max(input, 0), since the slope of the TL-neurons is one. Hence the maximum path metric can be selected as shown in Figure 13.

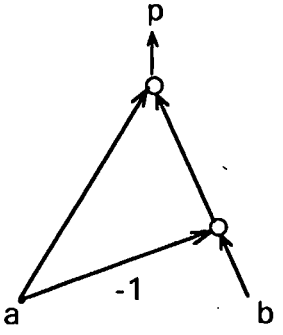


Figure 13. Compare-select subnet [12].

Branches with no weights have weights of 1.

Four copies of the above structure can be seen in Figure 11, between layers B and C (layer C included). The outputs of the layer C TL-neurons are then the path metrics of the four surviving paths, and these are fed back to be used in the following interval. Note that S0 to S3 are set to zero initially.

3.3.3 Register Exchange

Each layer D_4 TL-neuron stores d bits of information in the range $[0, 2^d-1]$. These information bits are also stored by the layer D_0 neurons as inputs for the next symbol interval. These d-bit values are the encoder input bits for the surviving paths. The most significant bit of each of these values is the oldest encoder input bit for that path.

Information about the surviving paths must be passed from the surviving path selection subnet to the register exchange subnet, so that the register values can be exchanged according to the selected surviving paths. Consider one pair of connected layer C HL-neurons in Figure 11. The output of the TL-neuron, leading to the HL-neuron on the left, is positive when the TL neuron's right hand input is larger than its left hand input, and zero otherwise. The operation of the layer C HL-neurons is then summarised by Table 1.

larger path metric entering TL-neuron	**output of TL-neuron leading to HL-neuron**	**left hand HL neuron output**	**right hand HL-neuron**	
			input	**output**
right hand	positive	1	-1+0.5=-0.5	0
left hand	0	0	0+0.5=0.5	1

Table 1. Register exchange control signal generation.

Hence the HL-neuron on the same side as the surviving path has an output of 0, and the other HL-neuron has an output of 1. These HL-neuron outputs are then multiplied by -2^d. The output of the layer D_1 TL-neuron with one input equal to -2^d will be 0, since the other input is always smaller than 2^d (the range of the layer D_0 and D_4 neurons is 0 to 2^d-1). Therefore this TL-neuron will contribute nothing to the output of the layer D_2 neuron. However, the value of the other layer D_1 neuron (with one input equal to 0), will propagate through layer D_1. The connections between layers D_0 and D_1 are determined from the trellis diagram. The resulting effect is that the contents of the layer D_0 register which is associated with the surviving path, propagates through layer D_1, and the contents of the other register is blocked.

Now that the input sequence, associated with the surviving path of the preceding state, has survived, the oldest bit must be removed, and the new bit must be inserted. The following operations would be performed on a digital register.

1) Assuming that the left most bit is the oldest bit, left shift the contents of the register by one bit. The oldest bit is shifted out and becomes an output.

2) Insert a 0 into the least significant bit position if the input to the encoder was a 0 to have caused the last state transition, or insert a 1 if the input was a 1. Note that for this encoder, both paths entering a state are due to the same encoder input.

The equivalent operations are performed on an analogue register (TL-neuron) in the following manner:

1) The following operation is equivalent to the left shift. The outputs of the layer D_1 TL-neurons are multiplied by 2. Then 2^d is subtracted if the output of the layer D_2 TL-neuron is greater or equal to 2^d. (If the output of the layer D_2 TL-neuron is greater or equal to 2^d, the output of the layer D_3 HL-neuron is 1, and -2^d is added to the output of the layer D_2 TL-neuron, at the layer D_4 neuron. If the output of the layer D_2 TL-neuron is smaller than 2^d, the output of the layer D_3 HL-neuron is zero, and the output of the

layer D_2 TL-neuron becomes the output of the layer D_4 TL-neuron. Note that this operation keeps the output of the layer D_4 neuron in the range 0 to 2^d-1.)

2) The new input bit is then inserted into the analogue register by adding 1 or 0 to the input of the layer D_4 TL-neuron.

3.3.4 Extracting the Decoder Output

What remains to be done is to select the maximum path metric from the surviving path metrics, and to extract the most significant bit from its corresponding analogue register. Note that in Figure 11:

If P0 > P1 , then E0 > 0 and hence E4 = 1, else E4 = 0.
If P3 > P2 , then E1 > 0 and hence E6 = 1, else E6 = 0.
E2 = max(P0, P1).
E3 = max(P2, P3).
E5 = 1 if max(P0, P1) > max(P2, P3), else E5 = 0. (3-6)
The output to F0 = -E5 - E4 + 2.
The output to F1 = E4 - E5 + 1.
The output to F2 = E6 + E5.
The output to F3 = E5 - E6 + 1.

Using the above equation, Table 2 summarises the neuron outputs during the selection of the surviving path with maximum path metric. Note that Figure 11 was changes slightly from that in [12] by adding 1 to the input of each layer F neuron.

Path Metric which is maximum	**Output of E_4 E_5 E_6**	**Output of F_0 F_1 F_2 F_3**
P_0	1 1 0	0 1 1 2
P_0	1 1 1	0 1 2 1
P_1	0 1 0	1 0 1 2

P_1	0 1 1	1 0 2 1
P_2	0 0 0	2 1 0 1
P_2	1 0 0	1 2 0 1
P_3	0 0 1	2 1 1 0
P_3	1 0 1	1 2 1 0

Table 2. Maximum path metric selection signals.

From Table 2 and Figure 11 it can be seen that the left hand input is 0 for the layer G neuron with maximum path metric, and the left hand input for all the other layer G neurons is less than -2^d. For each layer G TL-neuron with one input less than -2^d, the output is 0, and these states therefore contribute nothing to the output data stream. The layer G TL-neuron with the 0 input allows its other input value to propagate through to layer H, where 2^{d-1} is subtracted to extract the most significant bit. This bit is then the output of the decoder.

3.4 Modifications to the ANN Decoder

Certain problems arise when implementing the structure shown in Figure 11. Large dynamic ranges are required by some TL-neurons, and in practice the linear range of a TL-neuron is limited. A solution to this problem, and certain steps to reduce the complexity of the decoder, are discussed below.

- TL-neurons with d-bit precision are required for keeping the encoder input sequence for a decoding depth of d bits. It is reported in [12] that for convolutional coding, a decoding depth of greater or equal to 5.8m results in a negligible degradation in the decoding performance, when compared to observing the decoder input over an infinite duration. m is the total number of memory elements in the encoder. A depth of 5 to 10 times the constraint length of the convolutional code is used in practice, and similar results are expected for multi-h CPM decoding. Therefore d is usually chosen between 5 and 10 times the constraint

length of the code, and it becomes impossible to implement analogue TL-neurons which are linear over the required dynamic range. One solution is to use a digital register exchange system, with digital registers of length d.

- Another problem is that a path metric increases indefinitely. One solution is to subtract the minimum path metric from all the path metrics, every symbol interval, in which case the maximum value of the path metrics is bounded. Another method is to select an arbitrary state, and to set a threshold of say 10 times the typical branch metric. When the path metric of the selected state exceeds the threshold, the difference is subtracted from all the path metrics, so that all the path metrics are always in the region of the threshold. Note that the latter approach does not require the selection of the minimum path metric.

- The structure in Figure 11 can be simplified by considering the following: It is reported in [12] that as the surviving paths are traced back through the trellis, a point is reached beyond which all the paths are the same. If that point occurs within the decoding depth (d), all the register exchange registers will contain the same oldest bit, and hence the selection of the surviving path with maximum path metric for determining the output of the decoder is unnecessary. This makes layers E, F and G of Figure 11 redundant.

The resulting decoder is shown in Figure 14.

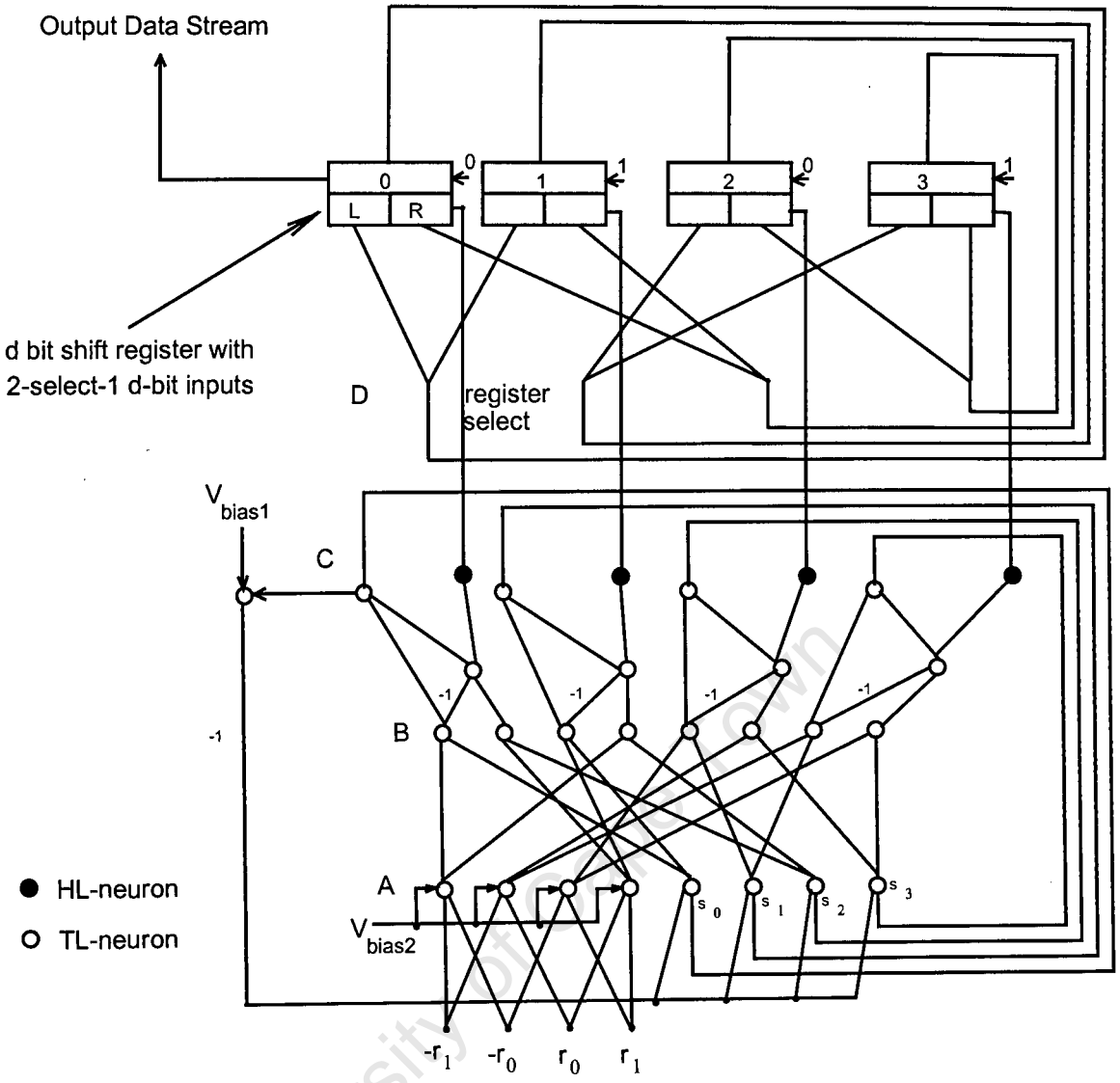


Figure 14. Hybrid ANN Viterbi decoder [12].

Note : The direction of the flow of information is upwards into a node where not specified.
Branches with no weights have weights of 1.

The decoder is now much simpler. In order to select the correct register during register exchange, consider the two left most layer B TL-neurons. If the output of the right hand neuron is greater than that of the left hand neuron, the output of the HL-neuron (layer C) is 1, and the right hand input (R) is to be selected for register 0. Otherwise L is selected. The new input bit is added to each register by left shifting a 0 or a 1 into the first position of each register, while shifting out the most significant bit. Any one of the most significant bits can be chosen as the output of the decoder (since d is assumed to be large enough to include the point beyond which all the surviving paths

are the same). In Figure 14, V_{bias1} could be chosen to be -10 times the typical branch metric, and V_{bias2} could be 4 times the typical branch metric to allow for a relatively large negative noise contribution.

It has been shown that all the necessary decoding information can be found by inspection of the trellis diagram, and therefore no training of the neural network is required. This decoding structure therefore reduces to a hybrid digital-analogue implementation of the VA. The VA is known to be optimal and therefore no trained neural network will perform better than this decoder. Therefore no in depth analysis of how and why neural networks work, or of their computational powers, will be done. For a comprehensive study of neural networks, see [19] and [20].

4. ANN Multi-h Decoder.

This chapter describes how the ANN decoder in Chapter 3 can be adapted for the decoding of a multi-h CPM scheme.

4.1 The Multi-h Code and Phase Trellis Diagram

The trellis diagram of $H_2 = \{1/4, 2/4\}$ is shown in Figure 15. Next to each phase state there are two numbers. The first is the depth or position in the trellis, in the range 0 to 3. The second is the state number, also in the range 0 to 3. Since it is a binary scheme, there are two branches leaving each state, and these are labelled A through H (per period). Note that this code is a poorly performing code, but has been chosen because there are only four phase states per symbol interval. This makes the trellis and decoder diagrams clear and easier to understand.

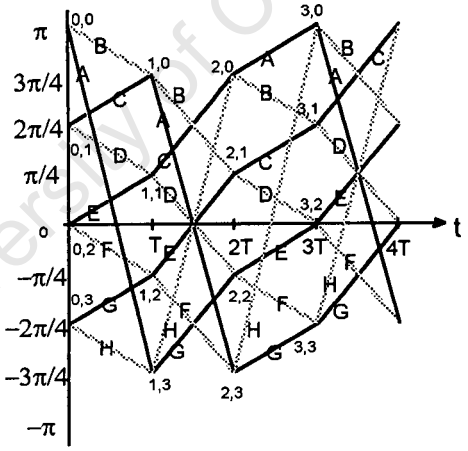


Figure 15. Trellis diagram for $H_2 = \{1/4, 2/4\}$.

Table 3 shows for each input and depth, the previous states of each state, and the branches connecting the two states.

		input = 0			input =1		
depth	**current state**	**previous state**	**branch**		**previous state**	**branch**	
0	0	3	H	r	1	C	l
1	0	0	B	l	1	C	l
2	0	3	H	r	1	C	l
3	0	3	H	r	0	A	r
0	1	0	B	r	2	E	r
1	1	1	D	l	2	E	r
2	1	0	B	r	2	E	r
3	1	0	B	r	1	C	l
0	2	1	D	l	3	G	r
1	2	2	F	r	3	G	r
2	2	1	D	l	3	G	r
3	2	1	D	l	2	E	l
0	3	2	F	r	0	A	l
1	3	3	H	l	0	A	l
2	3	2	F	r	0	A	l
3	3	2	F	r	3	G	r

Table 3. State transitions for trellis diagram in Figure 15.

It can be seen that each state has three predecessors, of which only two are valid during any symbol interval. For example: If the current state is state 0 and the input is 0 and the depth is 0, then the previous state is state 3. But if the depth is 1, then the previous state is state 0. Hence the decoder has to keep track of which state transitions are allowed at any point in time.

4.2 The Multi-h Decoder

Figure 16 shows the multi-h decoder for the code described by the phase trellis diagram in Figure 15.

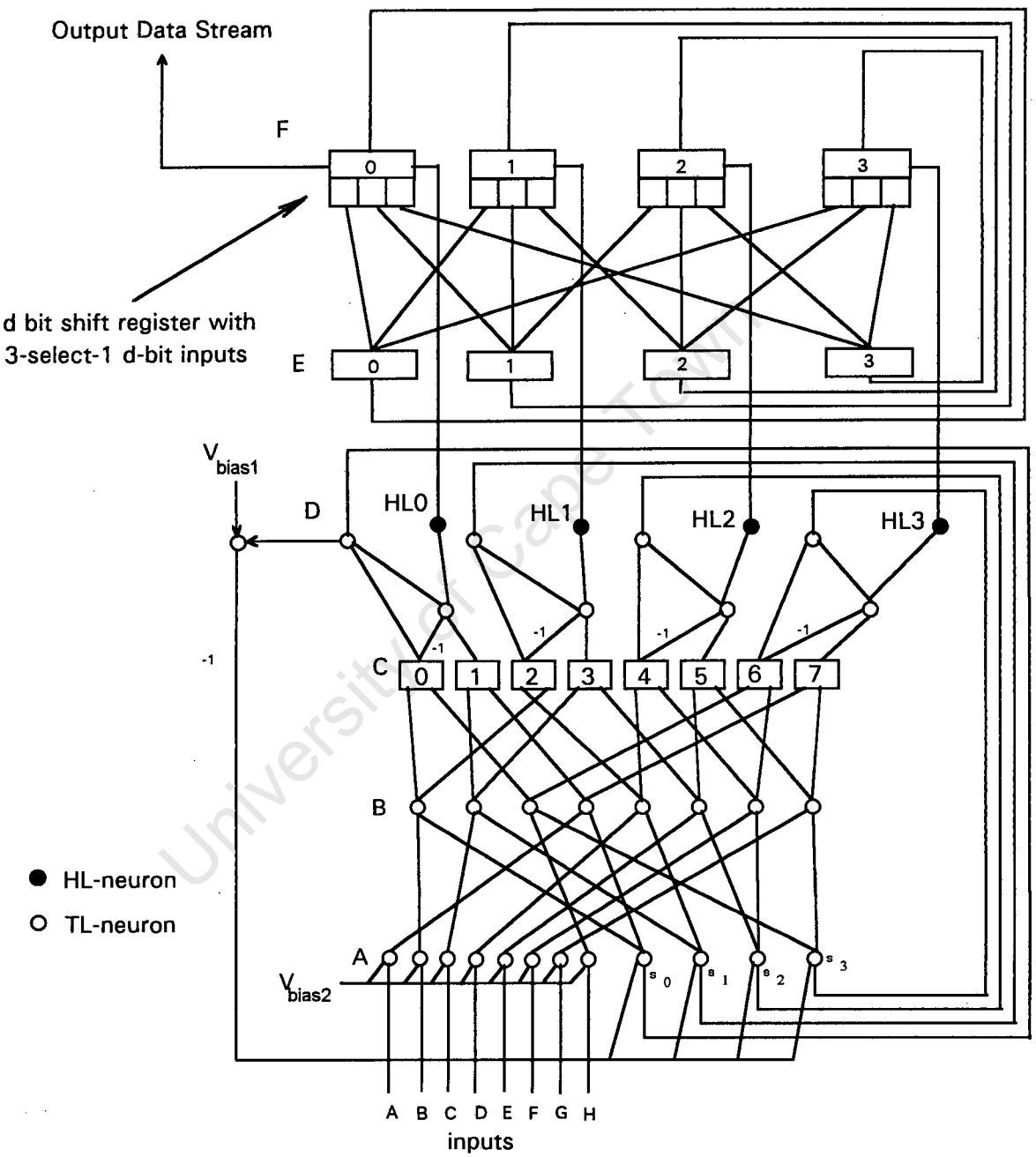


Figure 16. ANN multi-h decoder.

Note : The direction of the flow of information is upwards into a node where not specified.
Branches with no weights have weights of 1.

The structure in Figure 16 resembles that shown in Figure 14, but there are some important differences:

1) The branch metrics are calculated as described in the latter part of Equation 2-7. Eight of the structures shown in Figure 7 are required, and their outputs are the inputs to the decoder.

2) The new path metrics are calculated by the layer B TL-neurons. From Table 3 it is evident that only eight different summations of old path metrics (or state metrics) and branch metrics are required. The eight metrics are: S3 + H, S1 + C, S0 + B, S2 + E, S1 + D, S3 + G, S2 + F and S0 + A, where SX is the path metric of previous state X. The interconnections between layers A and B ensure that only these values are calculated.

3) To the right of label C are eight selection blocks. Each selection block selects the input which is valid for the current position in the trellis. There are two selection blocks for each of the four (q = 4) states, with the left most pair belonging to state 0. For each pair the inputs are arranged so that the transitions associated with an encoder input of 0 enter the left hand selection block, and those associated with a 1 enter the right hand selection block. This is done so that if the new path metric at the output of the left hand selection block is larger than that at the output of the right hand one, the left hand path will be selected as the survivor, and the encoder input will be assumed to have been a 0. And if the new path metric at the output of the right hand selection block is larger than that at the output of the left hand one, the right hand path will be selected as the survivor, and the encoder input will be assumed to have been a 1. For example: Consider only the left most pair of selection blocks in Figure 16. From Table 3 it can be seen that if the encoder input is 0, and the current state is state 0, then the new path metric for state 0 is either S3 + H or S0 + B, depending on the position (depth) in the trellis. These are the inputs to the left hand selection block. Similarly, if the encoder input is 1, and the current state is state 0, then the new path metric for state 0 is either S1 + C or S0 + A, and these are the inputs to the right hand selection block.

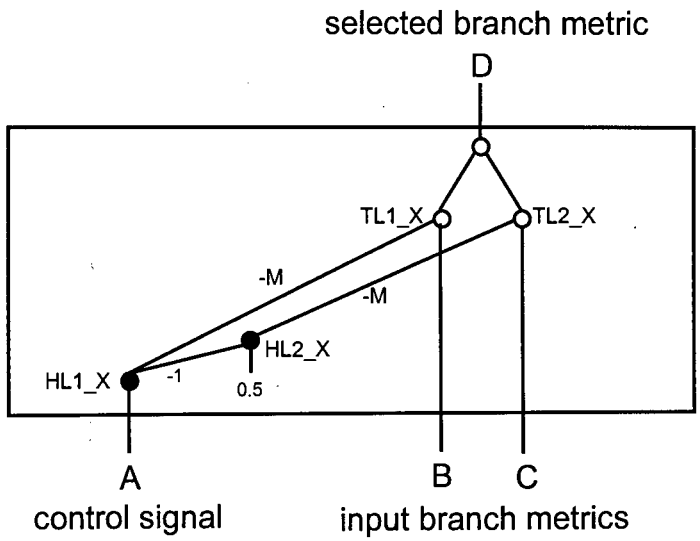


Figure 17. Selection unit.

Note : The direction of the flow of information is upwards into a node where not specified. Branches with no weights have weights of 1.

In Figure 17, A is the control signal, B and C are the path metric inputs, and D is the output. M is a constant and must always be greater than the path metric inputs (B and C). The operation of the selection unit is summarised in Table 4.

Control signal A	HL1_X's total input	HL1_X's output	HL2_X's total input	HL2_X's output	output D
0	0	0	0.5	1	D = B
1	1	1	-0.5	0	D = C

Table 4. Operation of the selection unit.

The control signals (A in Table 4) need to be generated for each selection unit and for each position in the trellis. In Table 3 there is a "r" or "l" to the right of each branch entry. A "r" or "l" means that the branch is the left hand input (B in Figure 17) or right hand input (C in Figure 17) to one of the selection units in Figure 16. From Tables 3 and 4 the control signals can be found for each of the eight selection units, at each point in the trellis, and these are given in Table 5. Note that AX is the control signal

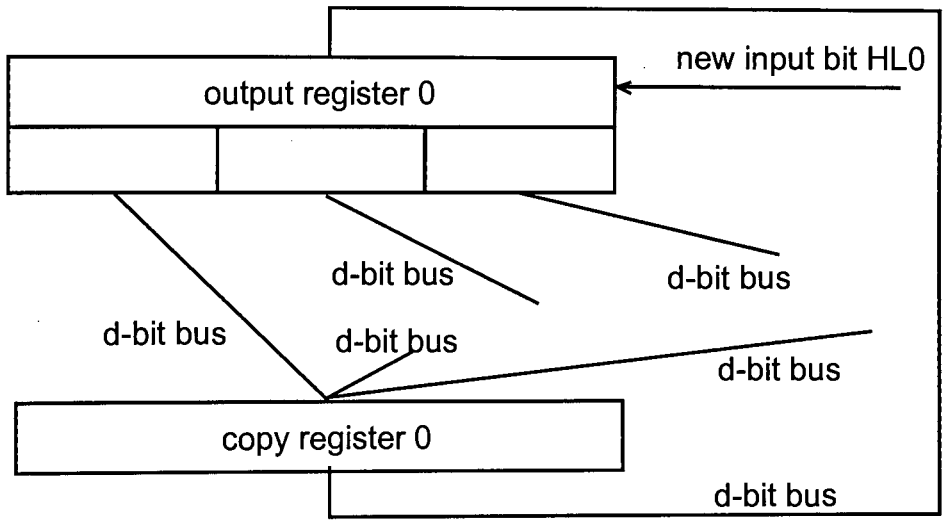
for selection unit X, and that if $AX \leq 0$, then the left hand input (B in Figure 17) propagates through to the output. Otherwise input C propagates through to the output.

	Selection block control signal							
depth	**A0**	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**
0	1	0	1	1	0	1	1	0
1	0	0	0	1	1	1	0	0
2	1	0	1	1	0	1	1	0
3	1	1	1	0	0	0	1	1

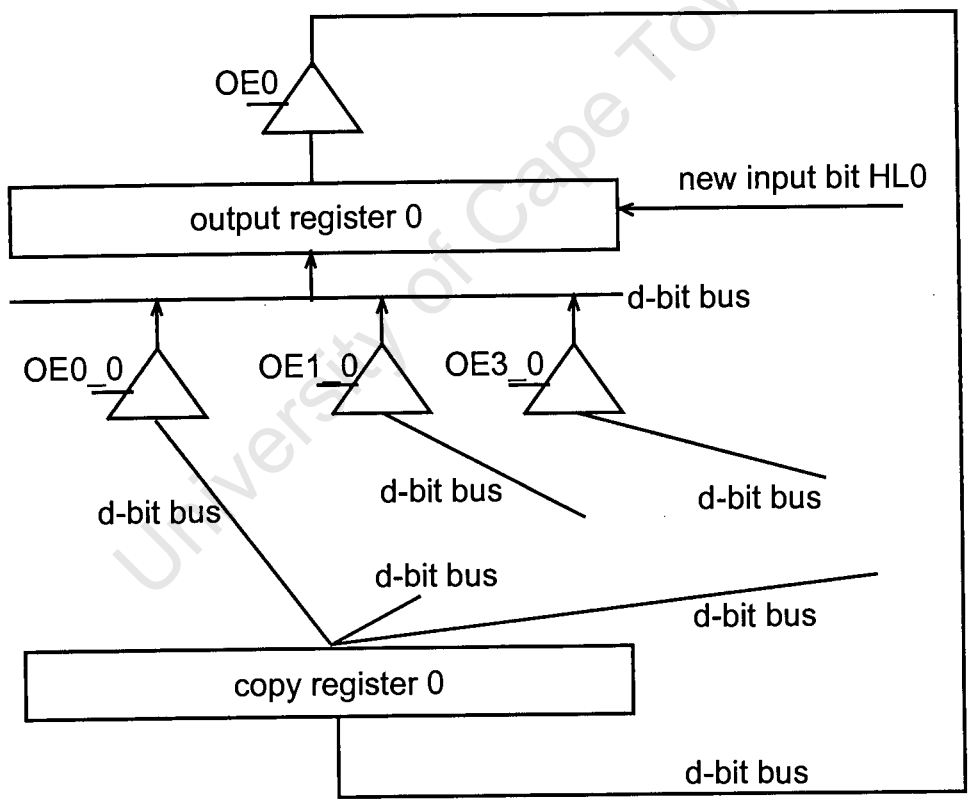
Table 5. Control signals for the selection units.

4) The TL-neurons between layers C and D then select the surviving path for each state, as described in Chapter 3. The surviving path metrics are stored by the layer D TL-neurons and are fed back to S0, S1, S2 and S3, where they become the old path metrics (or state metrics) for the following symbol interval. Recall that if the left hand path metric (e.g. output of selection block 0 in Figure 16) is larger, then the output of the layer D HL-neuron is 0, and if the right hand path metric (e.g. the output of selection block 1) is larger, then the HL-neuron output is 1. The HL-neuron output is therefore the same as the encoder input associated with the selected path, and becomes the new input to the register.

5) To ensure that the contents of each register exchange register is updated with that from the register of the preceding state along the surviving path, certain control signals are required. It is necessary to first make a copy of the information in each register, and these copies are to be used for updating the output registers. This is done to ensure that the contents of a register is not updated before it is to be used elsewhere. Shown in Figure 18(i) is the left most output register of Figure 16, and the register used to keep a copy of its contents. Figure 18(ii) shows how Figure 18(i) can be implemented. Note that the triangular shaped objects are d-bit buffers.



(i)



(ii)

Figure 18. Register exchange control signals for state 0.

Each output register will have a similar configuration. Note that none of the busses in Figure 18 are connected to each other. Each set of buffers and output register has an independent bus, connecting only those buffers and output register. This allows simultaneous updating of all output registers.

The control signals required for driving the output enable signals of the buffers, can be generated from the control signals of the selection units (Tables 4 and 5) and the layer D HL-neuron outputs, by using simple digital logic. Consider the following example (refer to Figure 16):

- Recall that if the left most layer D HL-neuron output is 0, then the surviving path enters selection unit 0. If A0 is 0, the left hand input to selection unit 0 is selected, and the previous state is state 0. If A0 is 1, the right hand input to selection unit 0 is selected, and the previous state is state 3.

- Recall that if the left most layer D HL-neuron output is 1, then the surviving path enters selection unit 1. If A1 is 0, the left hand input to selection unit 1 is selected, and the previous state is state 1. If A1 is 1, the right hand input to selection unit 1 is selected, and the previous state is state 0.

By following the above example, all the register exchange control signals can be found, and these are given in Table 6.

state	layer D HL-neuron	layer D HL-neuron output	left hand selection unit control signal	right hand selection unit control signal	previous state	output enable signal
0	HL0	0	A0 = 0	A1 = x	0	OE0_0
0	HL0	0	A0 = 1	A1 = x	3	OE3_0
0	HL0	1	A0 = x	A1 = 0	1	OE1_0
0	HL0	1	A0 = x	A1 = 1	0	OE0_0
1	HL1	0	A2 = 0	A3 = x	1	OE1_1
1	HL1	0	A2 = 1	A3 = x	0	OE0_1
1	HL1	1	A2 = x	A3 = 0	1	OE1_1
1	HL1	1	A2 = x	A3 = 1	2	OE2_1
2	HL2	0	A4 = 0	A5 = x	1	OE1_2
2	HL2	0	A4 = 1	A5 = x	2	OE2_2
2	HL2	1	A4 = x	A5 = 0	2	OE2_2
2	HL2	1	A4 = x	A5 = 1	3	OE3_2
3	HL3	0	A6 = 0	A7 = x	3	OE3_3
3	HL3	0	A6 = 1	A7 = x	2	OE2_3
3	HL3	1	A6 = x	A7 = 0	0	OE0_3
3	HL3	1	A6 = x	A7 = 1	3	OE3_3

Table 6. Register exchange control signals.

Note : x means don't care.

From Table 6 the register exchange functions for state 0 are as follows:

$$\begin{aligned} OE0\_0 &= (\overline{HL0}\ \text{AND}\ \overline{A0})\ \text{OR}\ (HL0\ \text{AND}\ A1) \\ OE1\_0 &= HL0\ \text{AND}\ \overline{A1} \\ OE3\_0 &= \overline{HL0}\ \text{AND}\ A0 \end{aligned} \qquad (4\text{-}1)$$

Note that in Equation 4-1, an overscore means NOT. The implementation of Equation 4-1 is shown in Figure 19.

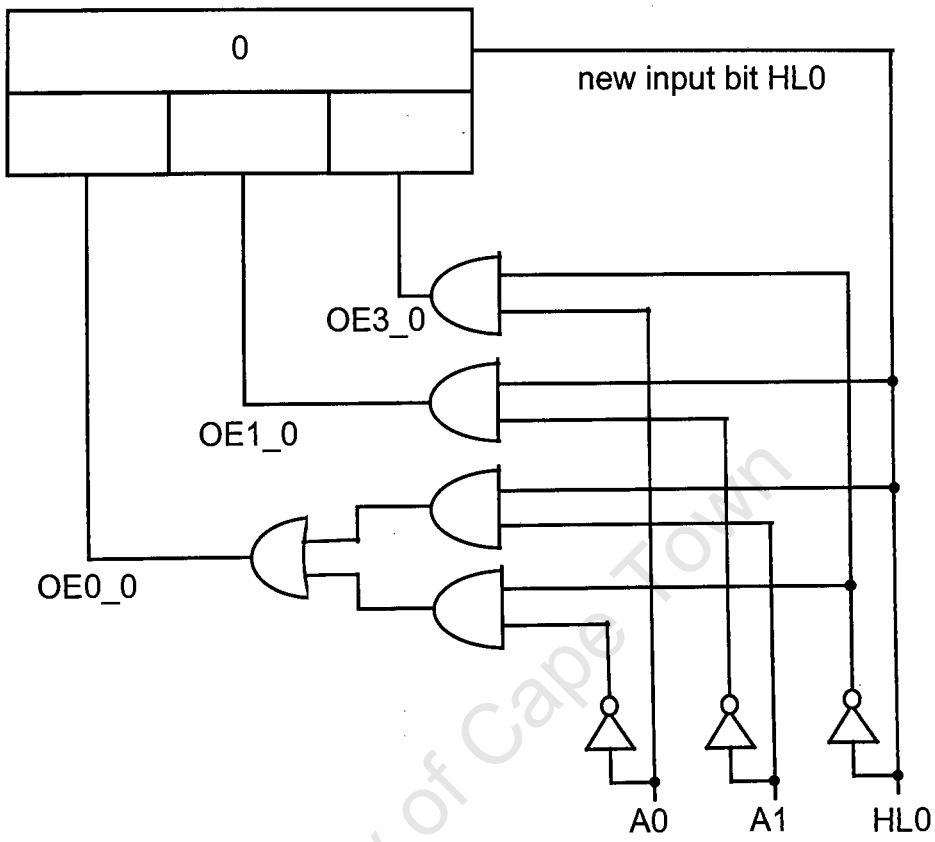


Figure 19. Register exchange control logic for state 0.

Similarly the functions for the other states are as follows:

state 1 :

$$\begin{aligned} OE0\_1 &= \overline{HL1}\ \text{AND}\ A2 \\ OE1\_1 &= (\overline{HL1}\ \text{AND}\ \overline{A2})\ \text{OR}\ (HL1\ \text{AND}\ \overline{A3}) \\ OE2\_1 &= HL1\ \text{AND}\ A3 \end{aligned} \qquad (4\text{-}2)$$

state 2 :

$$\begin{aligned} OE1\_2 &= \overline{HL2}\ \text{AND}\ \overline{A4} \\ OE2\_2 &= (\overline{HL2}\ \text{AND}\ A4)\ \text{OR}\ (HL2\ \text{and}\ \overline{A5}) \\ OE3\_2 &= HL2\ \text{AND}\ A5 \end{aligned} \qquad (4\text{-}3)$$

state 3 :

$$\begin{aligned} OE0\_3 &= HL3 \text{ AND } \overline{A7} \\ OE2\_3 &= \overline{HL3} \text{ AND } A6 \\ OE3\_3 &= (HL3 \text{ AND } A7) \text{ OR } (\overline{HL3} \text{ AND } \overline{A6}) \end{aligned} \qquad (4\text{-}4)$$

Again it is assumed that the point beyond which all the surviving paths are the same, is within the decoding depth (d), and therefore any one of the registers' oldest values can be used for the output of the decoder.

4.3 Decoder Complexity

The complexity of the decoder is characterised by the number of neurons and registers, and the number of branches and busses between them. The following analysis is done for the binary multi-h decoder in Figure 16. The number of neurons and branches can be calculated as shown in Table 7.

Position and function of neurons	**number of neurons**	**number of branches entering neurons**
layer A TL-neurons for ensuring that the branch metrics are always positive	2q	2q from branch metrics (1 per neuron) 2q from V_{bias2} (1 per neuron)
layer A TL-neurons for keeping path metrics in the region of V_{bias1}	q	q from V_{bias1} neuron (1 per neuron) q from layer D neurons (1 per neuron)
layer B TL-neurons for computing new path metrics	2q	4q from layer A neurons (2 per neuron)
layer C TL-neurons (3 for each of the 2q selection units)	6q	4q from layer B neurons (2 per selection unit) 8q internally (4 per selection unit)
layer C HL-neurons (2 for each of the 2q selection units)	4q	6q (3 per selection unit)

layer D TL-neurons for selecting the surviving path metrics	2q	4q (2 per neuron)
layer D HL-neurons for generating new input bits for the registers and register selection control signals	q	q (1 per neuron)
layer D TL-neuron for calculating the difference between V_{bias1} and the layer D reference path metric	1	2
Totals	**18q + 1**	**33q+2**

Table 7. Number of neurons and interconnections in the ANN multi-h decoder (Figure 16).

For the register exchange mechanism there are q output registers and q registers to keep the copies. There are q d-bit busses for feeding back the contents of the output registers, 3q d-bit busses for distributing the information to the buffers and q d-bit busses for connecting the buffers to the output registers. Hence the total number of registers is 2q and there are 5q d-bit busses.

It is important to note that slight variations of this structure are possible. However the number of neurons, branches, registers and d-bit busses are all linear functions of q, and therefore the complexity of the decoder is Order q. Therefore codes with small q values are desired.

5. Simulations

In order to ensure correct interpretation of the ANN decoder by Xiao-an Wang in [12] (as described in Chapter 3), and to verify the multi-h decoder structure as described in Chapter 4, both these decoders were implemented in software on an IBM compatible PC, using Borland C. This chapter describes how the bit error rate performances of the decoders were found for an AWGN channel. The results are compared to those found in [1] and [2]. Perfect phase, bit and superbaud synchronisation were assumed. The AWGN channel was modelled with a Gaussian distributed random number generator, by William H. Press *et al*, in [15]. The information bit sequence was generated by a 31-bit pseudo random number generator.

5.1 Additive White Gaussian Noise Source

To ensure correct implementation of the noise source described in [15], the statistics of the noise generator output was found through simulation, and compared to that described in the literature. The Borland C source code for the noise source is given in Appendix A, and that of the simulation is given in Appendix B.

Consider the receiver structure shown in Figure 20 below.

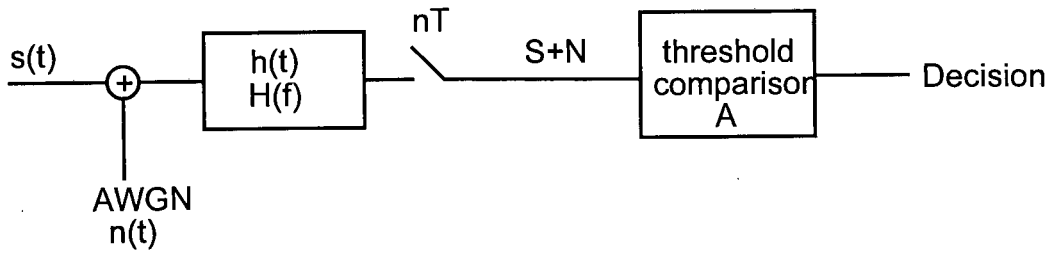


Figure 20. Maximum likelihood receiver.

Let n(t) be an AWGN signal with zero mean and double sided power spectral density $N_0/2$. When only n(t) is applied to the filter's (h(t)'s) input, the output at time nT is a Gaussian random variable N, with zero mean and variance (σ^2) given by

$$\sigma^2 = \int_{-\infty}^{\infty} |H(f)|^2 \frac{N_0}{2} df \,. \tag{5-1}$$

It is assumed that the initial conditions of the filter are set to zero at the start of each symbol interval.

Let s(t) be one of two possible signals, $s_1(t)$ or $s_2(t)$, and let the output of the filter at time nT be S_1 or S_2 when the input is only $s_1(t)$ or only $s_2(t)$ respectively. Let $S_1 > S_2$. If $s_1(t)$ and $s_2(t)$ have the same probability of occurring, then the decision boundary (A) that minimises the probability of making a decision error, is given by

$$A = \frac{S_1 + S_2}{2}, \tag{5-2}$$

for the following decision strategy:

If S + N > A, decide that $s_1(t)$ was sent.
If S + N < A, decide that $s_2(t)$ was sent.
If S + N = A, randomly decide on $s_1(t)$ or $s_2(t)$ with equal probability.

The probability of making a symbol error is then given by

$$P_E = Q\left(\frac{S_1 - S_2}{2\sigma}\right) . \tag{5-3}$$

If $S_1 = +1$ and $S_2 = -1$, then

$$P_E = Q\left(\frac{1}{\sigma}\right) . \qquad (5\text{-}4)$$

To simulate the sampled output of the filter, a random variable with standard deviation σ was added to a random sequence of S_1 (+1) and S_2 (-1). The resulting error rates are shown in Table 8.

σ	P_E (theory) = Q(1/σ)	P_E (simulation)
4	0.4013	0.4010
2	0.3085	0.3092
1	0.1587	0.1596
0.9	0.1357	0.1340
0.8	0.1056	0.1050
0.7	0.0764	0.0761
0.6	0.0475	0.0470
0.5	0.0228	0.0227
0.4	0.00621	0.00634
0.3	0.000404	0.000400

Table 8. A comparison of the theoretical and simulated error rates for a Gaussian noise source.

The Gaussian random variable generator is therefore accurate and can be used to simulate an AWGN channel.

5.2 Verification of the Results in [12]

The ANN decoder in Figure 14 was implemented using Borland C, as described in Appendix C. This is a functional implementation, i.e. the functions of the various blocks were implemented.

The standard way of presenting error rate results for a particular modulation or coding scheme, is to plot the bit error rate (BER) versus E_b/N_0 (in dB). When comparing bit error rate curves, it is required that the same bandwidth, and hence the same transmission bit rate (T_b) be used. Antipodal baseband will be used for transmitting the information. The amplitude of the antipodal baseband signal will be ±1V and T_b will be 1 second.

For the rate-1/2 convolutional encoder in Figure 8(i), 2 bits are transmitted per symbol, hence

$$T = 2T_b = 2\,\text{sec} \tag{5-5}$$

and

$$E_S = 2E_b. \tag{5-6}$$

For the uncoded case, one bit is transmitted per symbol and hence

$$T = T_b = 1\,\text{sec}. \tag{5-7}$$

In both cases the first stage of the optimal receiver is an ideal integrator. The output at the end of each bit interval (see Figure 20) is +1 or -1 for a noise free input of +1V or -1V respectively. For the uncoded case the variance of the random variable (N) at the output of the integrator is then $\sigma^2 = N_0/2$. The proof is as follows: N is given by

$$N = \int_{t}^{t+T} n(t)dt. \tag{5-8}$$

From [14] by Michel C. Jeruchim *et al*, this process is equivalent to passing n(t) through a linear time-invariant causal system with an impulse response of

$$h(t) = \begin{cases} 1 & for\ 0 < t < T \\ 0 & elsewhere \end{cases}. \tag{5-9}$$

The corresponding transfer function of the filter is given by

$$H(f) = Te^{-j\pi fT}\left(\frac{\sin \pi fT}{\pi fT}\right). \tag{5-10}$$

Then from Equation 5-1,

$$\begin{aligned} \sigma^2 &= \int_{-\infty}^{\infty} |H(f)|^2 \frac{N_0}{2} df \\ &= \int_{-\infty}^{\infty} T^2 \left(\frac{\sin \pi fT}{\pi fT}\right)^2 \frac{N_0}{2} df \\ &= \frac{T^2 N_0}{2} \times \frac{1}{T} = \frac{N_0 T}{2} = \frac{N_0}{2}, \end{aligned} \tag{5-11}$$

which is the desired result.

The probability of error for uncoded antipodal baseband in an AWGN channel is given by Equation 5-12 [2]. Note that E_S = 1J for the uncoded case.

$$P_E = Q\left(\sqrt{\frac{2E_s}{N_0}}\right) = Q\left(\sqrt{\frac{2 \times 1}{2\sigma^2}}\right) = Q\left(\frac{1}{\sigma}\right). \tag{5-12}$$

For the coded simulation T = 2 seconds and E_S = 2J, and therefore the standard deviation (σ) of the noise source used in the uncoded case is multiplied by the square root of two, before being added to the coded antipodal baseband signal.

The simulation results are shown in Figure 21, and the curves are labelled as follows:

- A: uncoded theoretical antipodal baseband.

- B: uncoded simulated antipodal baseband.
- C: coded simulated antipodal baseband.

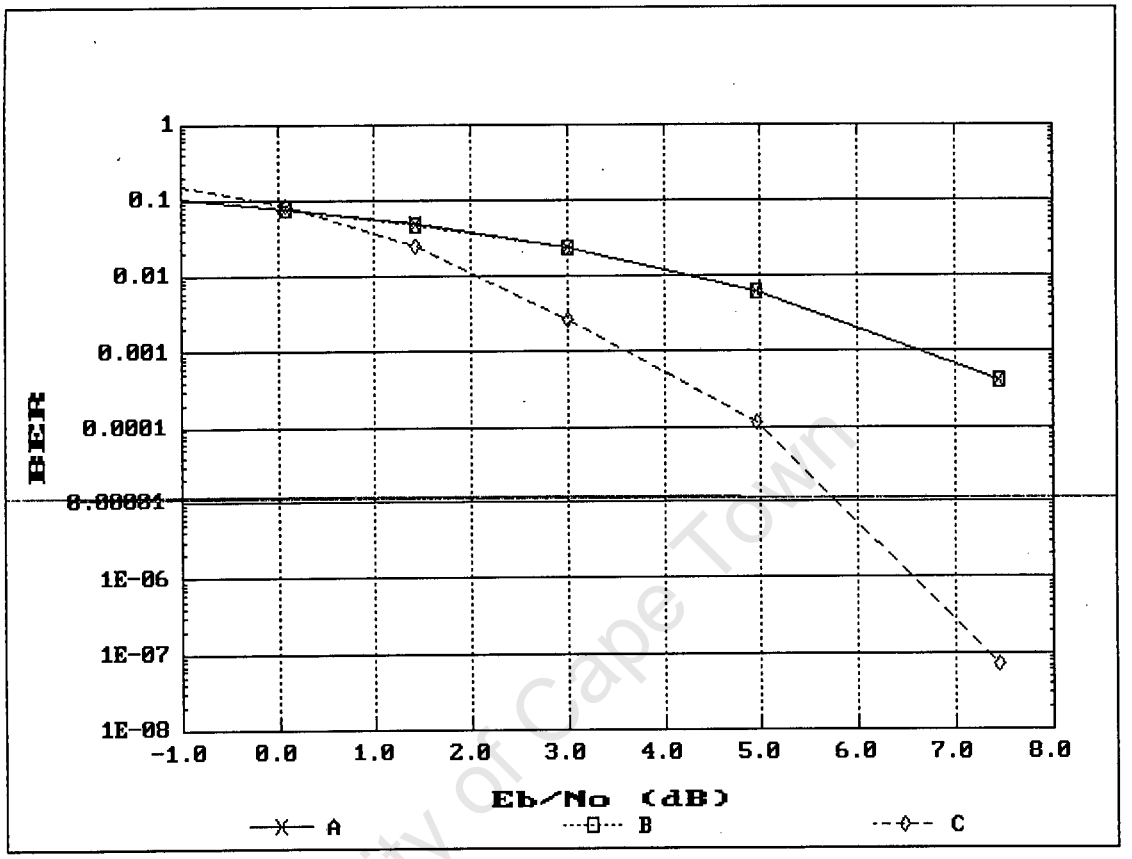


Figure 21. Bit error rate curve for the ANN convolutional decoder in [12], found by simulation.

In Figure 21, curves A and B are indistinguishable, again confirming that the noise source is accurate. Curve C is almost exactly that obtained in [12], and all three curves can be verified in the literature, e.g. in [2].

5.3 Multi-h Simulation

In this section the simulation results are presented for the multi-h ANN decoder in Figure 16. The Borland C source code is given in Appendix D. The latter part of Equation 2-7 is used for calculating the branch metrics. Eight branch metrics are

calculated every symbol interval for this decoder. The basis functions are not orthogonal (as discussed in Chapter 2), and therefore the contributions of the noise at the output of the correlators are not statistically independent. For this reason the noise must be applied to the input of the correlators. In other words, a Gaussian random variable cannot simply be added at the output of the correlators. The noise must be added to the transmitted signal. To compute the branch metrics, the now noisy received signal must be sampled at least twice per carrier period (Nyquist rate). This means that the noise signal is also sampled, and the PSD of the sampled noise process is, according to [2], given by

$$\frac{N_0}{2} = T_S\sigma^2. \tag{5-13}$$

In Equation 5-13, T_S is the sampling period and σ is the standard deviation of the noise process. Note that equation 5-13 is valid only for $T_S << T$, in which case the sampled noise appears to be white. The mean square value of the noise random variable N is then

$$E\left(N^2\right) = \frac{N_0}{2T_s} = \sigma^2. \tag{5-14}$$

Recall that MSK is used as a reference for multi-h error rate curves. For MSK the basis functions are orthogonal [2], and the AWGN channel can be simulated by adding a Gaussian random variable at the output of the MSK demodulator. If T = 1 second, the mean square value of the noise process (N) at the output of the MSK demodulator is, from Equation 5-11,

$$\sigma^2 = \frac{N_0}{2}B = \frac{N_0}{2T} = \frac{N_0}{2}. \tag{5-15}$$

Note that B = 1/T. From Equations 5-14 and 5-15, the standard deviation σ of the Gaussian random variable at the input of the multi-h correlator must therefore be

equal to $1/\sqrt{T_S}$ times that added at the output of the MSK demodulator. The branch metrics are then calculated as shown in Figure 22, which is a numerical implementation of the latter part of Equation 2-7.

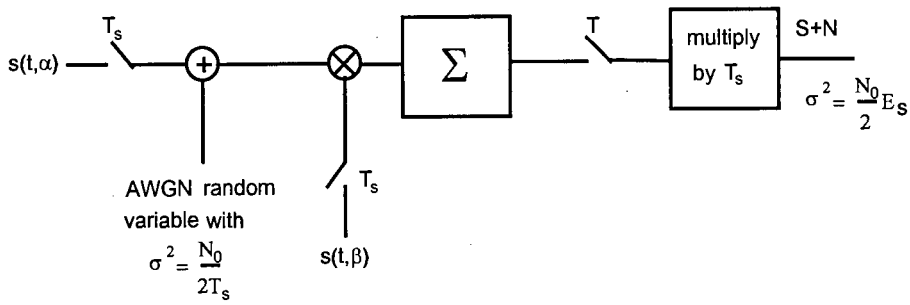


Figure 22. Numerical calculation of multi-h branch metrics.

Let the amplitude of the multi-h signal be 1V and recall that T = 1 second. Then the symbol energy of the sinusoidal multi-h signal is $E_S = 0.5$ Joule. Since only one bit is transmitted per symbol, the bit energy E_b is given by

$$E_b = E_s = 0.5\ Joule\,. \tag{5-16}$$

Then from Equations 5-14 and 5-16,

$$\frac{E_s}{N_0} = \frac{E_b}{N_0} = \frac{0.5}{2T_S\sigma^2} = \frac{1}{4T_S\sigma^2}\,. \tag{5-17}$$

Recall from section 2.5 that the approximate probability of error for multi-h is given by

$$P_E \approx Q\left(\sqrt{\frac{d_{\min}{}^2 E_b}{N_0}}\right). \tag{5-18}$$

d_{min}^2 was found to be 1.09 for $H_2 = (1/4, 2/4)$ by using numerical integration. This value was verified from a graph in [16], by T. Aulin *et al*, and by James Cuthbert, author of [11]. Then

$$P_E \approx Q\left(\sqrt{\frac{1.09E_b}{N_0}}\right) = Q\left(\sqrt{\frac{1.09}{4T_S\sigma^2}}\right). \qquad (5\text{-}19)$$

In this simulation a T_S of 1/100 was used to ensure that $T_S << T$. The carrier frequency ω_c is 100 radians per second, which is equal to $100/2\pi$ Hertz, and the Nyquist sampling rate is therefore satisfied. A decoding depth of 100 was used, which is much greater than 10 times the constraint length of 3, and therefore the results should be independent of the decoding depth. The simulation results are shown in Figure 23, and the curves are labelled as follows:

- A: theoretical error rate curve for MSK.
- B: theoretical multi-h error rate curve for $H_2 = (1/4, 2/4\}$.
- C: simulated multi-h error rate curve for $H_2 = (1/4, 2/4\}$.

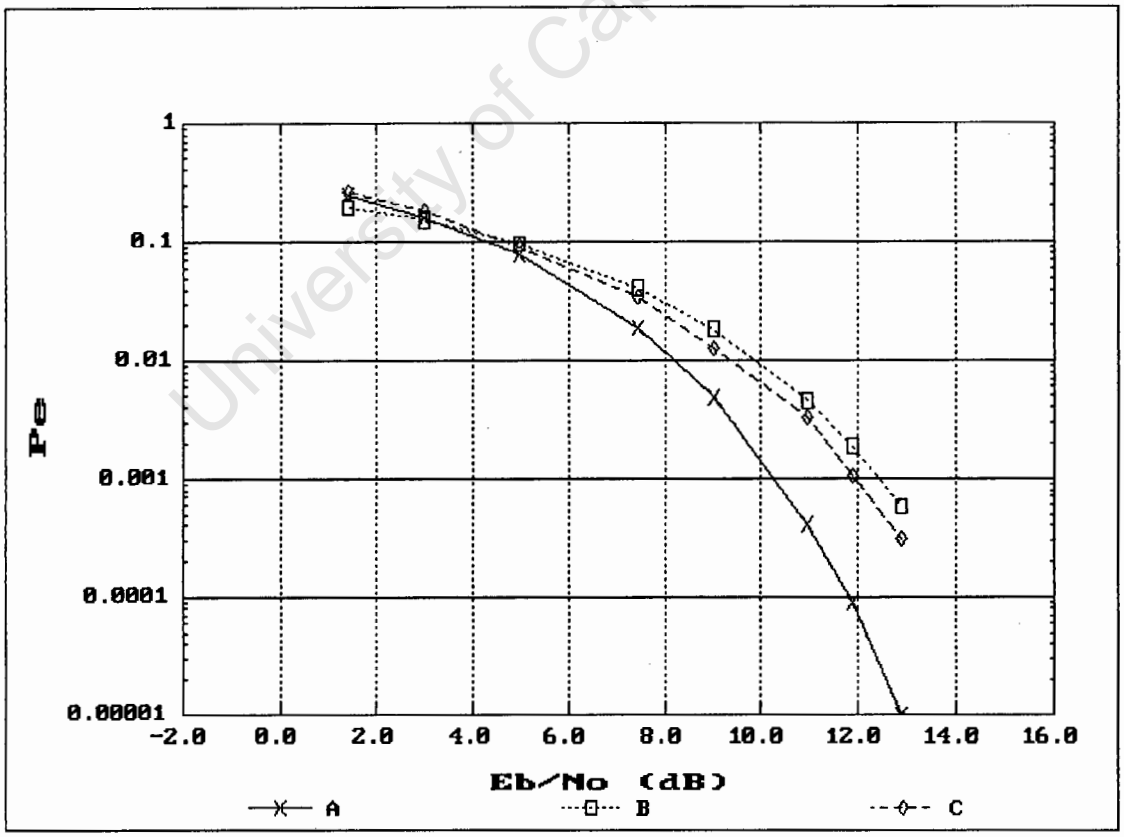


Figure 23. Multi-h probability of error curve for $H_2 = \{1/4, 2/4\}$.

From Figure 23, the theoretical error rate curve for the multi-h set $H_2 = \{1/4, 2/4\}$ (curve B) is approximately 2.15 dB worse than that for MSK (curve A). Hence the expected coding gain is -2.15 dB. Note again that the objective of this simulation was to verify the working of the decoder, and not to show that large gains can be achieved over MSK. The simulated multi-h curve (curve C) is approximately 0.5 dB better that expected (curve A), which is equivalent to an error in E_b/N_0 of 12%. The decoding structure in Figure 16 has therefore been verified.

6. Conclusions

The knowledge gained through studies of multi-h CPM and the ANN in [12], was combined to produce a ANN multi-h decoder. The error rate performance of the decoder was investigated by simulating the effect of an AWGN channel on the multi-h transmitted signal. The error rate performance of the decoder was found to be very close to that suggested by the literature [1], proving that the structure in [12] could be adapted for the decoding of a multi-h CPM scheme.

This decoder is not a trainable neural network, as was search for initially, but reduces to a hybrid analogue-digital implementation of the VA. However, since the VA is known to be optimal, no trainable neural network will perform better. As was the case for the ANN in [12], all computation is done by analogue neurons, and all digital operations can be performed by shift registers and simple feed forward digital logic. The multi-h decoder is therefore also suitable for VLSI implementation and high speed decoding.

All the goals of this thesis, as set out in the Terms of Reference, were therefore achieved.

Future work in this area may include a hardware implementation of the structure to highlight possible implementation problems.

7. Bibliography

[1] I. Sasase and S. Mori, "Multi-h Phase-Coded Modulation," IEEE Communications Magazine, Vol. 29, No. 12, pp. 46-56, December 1991.

[2] R. E. Ziemer and R. L. Peterson, *Introduction to Digital Communication,* New York, Macmillan, 1992.

[3] R. E. Ziemer and R. L. Peterson, *Digital Communication and Spread Spectrum Systems*, New York, Macmillan, 1985, pp. 228-250.

[4] Carl-Erik Sundberg, "Continuous Phase Modulation", IEEE Communications Magazine, Vol. 24, No. 4, pp. 25-38, April 1986.

[5] John M. Liebetreu, "Joint Carrier Phase Estimation and Data Detection Algorithms for Multi-h CPM Data Transmission", IEEE Transactions on Communications, Vol. Com-34, No. 9, pp. 873-881, September 1986.

[6] Brian A. Mazur and Desmond P. Taylor, "Demodulation and Carrier Synchronisation of Multi-h Phase Codes", IEEE Transactions on Communications, Vol. Com-29, No. 3, pp. 257-266, March 1981.

[7] Al-Nasir Premji and Desmond P. Taylor, "Receiver Structures for Multi-h Signalling Formats", IEEE Transactions on Communications, Vol. Com-35, No. 4, pp. 439-451, April 1987.

[8] Al-Nasir Premji and Desmond P. Taylor , "A Practical Receiver Structure for Multi-h CPM Signals", IEEE Transactions on Communications, Vol. Com-35, No. 9, pp. 901-908, September 1987.

[9] T. Aulin and Carl-Eric W. Sundberg, "Continuous Phase Modulation - Part I: Full Response Signalling", IEEE Transactions on Communications, Vol. Com-29, No. 3, pp. 196-209, March 1981.

[10] T. Aulin, N. Rydbeck and Carl-Eric W. Sundberg, "Continuous Phase Modulation - Part II: Path Response Signalling", IEEE Transactions on Communications, Vol. Com-29, No. 3, pp. 210-225, March 1981.

[11] James Cuthbert, "High Performance Multi-h CPFSK Modulator and Demodulator Design using Population Based Incremental Learning Search Methods", Ph.D. Thesis, University of Cape Town, submitted December 1996.

[12] Xiao-an Wang and Stephen B. Wicker, "An Artificial Neural Net Viterbi Decoder", IEEE Transactions on Communications, Vol. 44, No. 2, pp. 165-171, February 1996.

[13] John G. Proakis and Masoud Salehi, *Communication Systems Engineering,* New Jersey, Prentice-Hall, 1994.

[14] Michel C. Jeruchim, Philip Balaban and K. Sam Shanmugan, *Simulation of Communication Systems*, New York, Plenum Press, 1992.

[15] William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numerical Recipes in C*, Cambridge University Press, 1992.

[16] T. Aulin, N. Rydbeck and Carl-Eric W. Sundberg, "On the Optimum Euclidean Distance for a Class of Signal Space Codes", IEEE Transactions on Information Theory, Vol. IT-28, No. 1, pp. 43-55, January 1982.

[17] Stephen G. Wilson and Richard C. Gaus, "Power Spectra of Multi-h Phase Codes", IEEE Transactions on Communications, Vol. 29, pp. 250-256, No. 3, March 1981.

[18] G. David Forney, "The Viterbi Algorithm", Proceedings of the IEEE, Vol. 61, pp. 268-278, No. 3, March 1973.

[19] Simon Haykin, *Neural Networks: a comprehensive foundation*, Macmillan, 1994.

[20] J. Hertz, A. Kroagh and R. G. Palmer, *Introduction to the theory of Neural Computation*, Addison-Wesley, 1991.

[21] Simon Haykin, *Digital Communications*, New York, John Wiley and Sons, 1988.

[22] Ferrel G. Stremler, Introduction to *Communication Systems*, Reading, Massachusetts, Addison-Wesley Publishing Company, 1982.

8. Appendix A: Noise Source [15]

(Source Code in Borland C)

Coded by Jaz Schoonees , University of Cape Town .

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

float ran1( long *idum )

/* Returns a random number between 0.0 and 1.0.
   Call with idum a negative number to initialize.                */


{
  int j;
  int k;
  static long iy=0;
  static long iv[NTAB];
  float temp;

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

```
/* Returns a normally distributed deviate with zero mean and unit
   variance, using ran1(idum) as the source of uniform deviates
   Source : Numerical Recipes in C -
                    Press, Teukolsky, Vetterling, Flannery                    */



#include<math.h>

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

 if( iset==0 ){
        do{
                v1 = 2.0*ran1(idum)-1.0;
                v2 = 2.0*ran1(idum)-1.0;
                rsq = v1*v1+v2*v2;
                } while ( rsq>=1.0||rsq==0.0 );
        fac = sqrt( -2.0*log(rsq)/rsq );
        gset = v1*fac;
        iset = 1;
        return v2*fac;
        }
 else{
        iset=0;
        return gset;
        }
}
```

9. Appendix B: Noise Source Validation

(Source Code in Borland C)

```
/*standard include files*/
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h>

/*noise generator include files*/
#include <gasdev.c>
#include <ran1.c>

/*constants*/
#define K 100000 /*length of simulation*/

/*global variables*/
long *seed;         /*noise generator seed*/
long PNGEN;         /*pn-generator memory*/
float STDDEV;       /*standard deviation*/

/*-------------------------------------------------------------------------*/
/* initialise seed for noise generator                                     */
/*-------------------------------------------------------------------------*/
void InitSeed()
{
  *(seed)=-1234;
}

/*-------------------------------------------------------------------------*/
/* 31 bit pseudo noise generator for generating transmitted bit sequence :*/
/*returns a bit                                                            */
/*-------------------------------------------------------------------------*/
int GetBit()
{
int bit,newbit,j;
        if(PNGEN==0) PNGEN=1234;
        bit=PNGEN&0x01;
        newbit=(PNGEN&0x00000001)^((PNGEN>>3)&0x00000001);
        PNGEN=PNGEN>>1;
        PNGEN=PNGEN|(newbit<<30);
return bit;
}


/*-------------------------------------------------------------------------*/
/* Find probability of making a decision error when adding AWGN with */
/* entered standard deviation to a random sequence of  S1=+ and S2=-1  */
/*-------------------------------------------------------------------------*/
void Simulate()
{
int input,r;
```

```
double I=0,Q,i,q,T=0,n;
        while(I<K) /*until end of simulation*/
        {
                I++;
                /*get a bit*/
                input=GetBit();
                if(input==0)
                input=-1;

                /*get a noise value*/
                n=STDDEV*gasdev(seed);

                /*compute error rate*/
                if(input==1)
                {
                  if(n<-1)
                  {
                          T++;
                  }
                }
                else
                {
                  if(n>+1)
                  {
                          T++;
                  }
                }

                if(n==0)
                {
                        r= rand() % 2;  /*result is 0 or 1*/
                        T=T+r;
                }
        }
        /*print results to screen*/
        printf("\n%f",(float)(T/(I*1.0)));
        getch();
}

/*---------------------------------------------------------------------------*/
/* function main: read standard deviation from the console                   */
/*---------------------------------------------------------------------------*/
int main()
{
float stddev;

        clrscr();

        printf("\n stddev: ");
        scanf("%f",&stddev);
        STDDEV=(double)stddev;

        InitSeed();
        Simulate();

return 0;
}
```

10. Appendix C: ANN Convolutional Decoder [12]

(Source Code in Borland C)

This is a functional implementation of Figure 14.

```
/*standard include files*/
#include <graphics.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>

/*noise generator include files*/
#include <ran1.c>
#include <gasdev.c>

/*global constants*/
#define Q 4                     /*denominator of modulation index */
#define BM 8                    /*number of branch metrics*/
#define DEPTH 100               /*decoding depth*/
#define VBIAS1 10.0             /* Vbias1*/
#define SQRT2 1.4142136

/*global variables*/
double STDDEV;                  /*standard deviation of noise*/
int Register[Q][DEPTH];         /*register exchange output registers*/
int NewRegister[Q][DEPTH];      /*register exchange registers for copying*/
int States[Q];                  /*used to keep register exchange info*/
int EState[3];                  /*encoder memory elements*/
int InBit[DEPTH];               /*circular shift register for keeping last
                                    encoder input bits (DEPTH of them)*/
float InputStates[Q];           /*state metrics for current symbol period
                                    level A (S0 to S3)*/
float NewInputStates[Q];        /*path metrics of previous symbol period
                                    level C*/
float BranchMetrics[Q];         /*Branch metric inputs */
float PartialMetrics[BM];       /*Path metrics */
float Selectors[BM];            /*copy of path metrics*/
float Comparators[Q];           /*HL neuron outputs*/
int bit0,bit1;                  /*noise free encoder output bits*/
double av0,av1;                 /*noisy encoder output bits, ie received bits*/
int REGPOS;                     /*circular list position indicator*/
int input;                      /*symbol period counter*/
int OldBit;                     /*encoder input, DEPTH symbol periods ago*/
long PNGEN;                     /*pn generator memory elements*/
long COUNT,ERRORS,BER;          /*counters: ERRORS for coded simulation
                                    and BER for uncoded simulation*/
long *seed;                     /*noise source seed*/
```

```
/*-------------------------------------------------------------------------------*/
/* initialise seed for noise generator                                           */
/*-------------------------------------------------------------------------------*/
void InitSeed()
{
        *(seed)=-1234;
}

/*-------------------------------------------------------------------------------*/
/* initialise encoder state                                                      */
/*-------------------------------------------------------------------------------*/
void InitState()
{
        EState[0]=0;
        EState[1]=0;
        EState[2]=0;
}

/*-------------------------------------------------------------------------------*/
/* initialise registers to zero                                                  */
/*-------------------------------------------------------------------------------*/
void InitRegisters()
{
int i,j;
        for(i=0;i<DEPTH;i++)
        {
                for(j=0;j<Q;j++)
                {
                        Register[j][i] = 0;
                }
        }
}

/*-------------------------------------------------------------------------------*/
/* initialise decoder output bits to zero                                        */
/*-------------------------------------------------------------------------------*/
void InitInputStates()
{
int i;
        for(i=0;i<Q;i++)
        {
                InputStates[i] = 0;
        }
}
/*-------------------------------------------------------------------------------*/
/* 31 bit pseudo random generator for generating encoder input sequence:*/
/*returns a bit                                                                  */
/*-------------------------------------------------------------------------------*/
int GetBit()
{
int bit,newbit;
        if(PNGEN==0) PNGEN=1234;
        bit=PNGEN&0x01;
        newbit=(PNGEN&0x00000001)^((PNGEN>>3)&0x00000001);
        PNGEN=PNGEN>>1;
        PNGEN=PNGEN|(newbit<<30);
return bit;
}
```

```
/*-------------------------------------------------------------------------------*/
/* calculate branch metrics and add noise                                        */
/*-------------------------------------------------------------------------------*/
void GetBranchMetrics()
{
float noise0,noise1;
        /*find encoder output bits*/
        EState[2]=EState[1];
        EState[1]=EState[0];
        OldBit=InBit[REGPOS];
        InBit[REGPOS]=EState[0]=GetBit();
        bit0=(EState[0]&01)^(EState[2]&0x01);
        bit1=(EState[0]&01)^(EState[1]&0x01)^(EState[2]&0x01);
        bit0=bit0*2-1;
        bit1=bit1*2-1;

        /*get noise values*/
        noise0=gasdev(seed)*STDDEV;
        noise1=gasdev(seed)*STDDEV;

        /*count number of uncoded errors*/
        if(bit0==1&&noise0<-1) BER++;
        if(bit0==-1&&noise0>1) BER++;
        if(bit1==1&&noise1<-1) BER++;
        if(bit1==-1&&noise1>1) BER++;

        /*multiply noise standard deviation by root(2) for coded simulation
                and add noise to transmitted encoder bits*/
        av0=bit0+noise0*SQRT2;
        av1=bit1+noise1*SQRT2;

        /*compute branch metrics*/
        BranchMetrics[0]=(-av0-av1);
        BranchMetrics[1]=(av0-av1);
        BranchMetrics[2]=(-av0+av1);
        BranchMetrics[3]=(av0+av1);

}

/*-------------------------------------------------------------------------------*/
/* calculate path metrics                                                        */
/*-------------------------------------------------------------------------------*/
void CalcPartialMerics()
{

        PartialMetrics[0]=InputStates[0]+BranchMetrics[0];
        PartialMetrics[1]=InputStates[2]+BranchMetrics[3];
        PartialMetrics[2]=InputStates[0]+BranchMetrics[3];
        PartialMetrics[3]=InputStates[2]+BranchMetrics[0];

        PartialMetrics[4]=InputStates[1]+BranchMetrics[2];
        PartialMetrics[5]=InputStates[3]+BranchMetrics[1];
        PartialMetrics[6]=InputStates[1]+BranchMetrics[1];
        PartialMetrics[7]=InputStates[3]+BranchMetrics[2];

}
```

```
/*-------------------------------------------------------------------------*/
/* copy path metrics form PartialMetrics to Selectors                      */
/*-------------------------------------------------------------------------*/
void IncomingMetrics()
{
int i;
        for(i=0;i<BM;i++)
                Selectors[i]=PartialMetrics[i];
}

/*-------------------------------------------------------------------------*/
/*select maximum path metric for each state                                */
/*-------------------------------------------------------------------------*/
void SelectMaxMetrics()
{
int i;
        for(i=0;i<Q;i++)
        {
                if(Selectors[i*2]>Selectors[2*i+1]) /*left is greater*/
                {
                        Comparators[i]=0;
                        NewInputStates[i]=Selectors[i*2];
                }
                else                                /*right is greater*/
                {
                        Comparators[i]=1;
                        NewInputStates[i]=Selectors[i*2+1];
                }
        }
}


/*-------------------------------------------------------------------------*/
/* register exchange                                                       */
/*-------------------------------------------------------------------------*/
void RegisterExchange()
{
int i,j;
        /*copy registers*/
        for(i=0;i<DEPTH;i++)
        {
                for(j=0;j<Q;j++)
                {
                        NewRegister[j][i]=Register[j][i];
                }
        }

        /*find register exchange sequence (States)*/
                if(Comparators[0]==0) /*select left*/
                {
                        States[0]=0;
                }
                else
                {
                        States[0]=2;    /*select right*/
                }
                if(Comparators[1]==0)
                {
```

```
                States[1]=0;
        }
        else
        {
                States[1]=2;
        }
        if(Comparators[2]==0)
        {
                States[2]=1;
        }
        else
        {
                States[2]=3;
        }
        if(Comparators[3]==0)
        {
                States[3]=1;
        }
        else
        {
                States[3]=3;
        }

    /*exchange registers*/
    for(i=0;i<DEPTH;i++)
    {
        for(j=0;j<Q;j++)
        {
                Register[j][i] = NewRegister[ States[j] ][i];
        }
    }
}
/*-------------------------------------------------------------------------------*/
/* determine decoder output error count and insert new bits into registers*/
/*-------------------------------------------------------------------------------*/
void UpdateRegisters()
{
int I;
    /*print output bits to screen for observation*/
    printf("\n %d %d %d %d ",Register[0][REGPOS],Register[1][REGPOS],
        Register[2][REGPOS],Register[3][REGPOS]);
    if(input<DEPTH) input++;

    /*count number of coded simulation errors*/
    if(input==DEPTH)
     {
        if(OldBit!=Register[0][REGPOS])
        {
                ERRORS++;
        }
     }
    /*print input bit of DEPTH symbol intervals ago and biased branch metrics
        to screen for observation*/
    printf(" %d :",OldBit);
    printf(" %2.3f %2.3f %2.3f %2.3f : ",InputStates[0],InputStates[1],InputStates[2],
    InputStates[3]);

    /*insert new input bits*/
```

```
        Register[0][REGPOS]=0;
        Register[1][REGPOS]=1;
        Register[2][REGPOS]=0;
        Register[3][REGPOS]=1;
}

/*-------------------------------------------------------------------------------*/
/* feed back path metrics for next symbol period and keep path metrics   */
/* near Vbias1                                                           */
/*-------------------------------------------------------------------------------*/
void UpdateInputs()
{
int i;
float diff;
        if(NewInputStates[0]>VBIAS1)
        {
                diff=NewInputStates[0]-VBIAS1;
                for(i=0;i<Q;i++)
                {
                        NewInputStates[i]=NewInputStates[i]-diff;
                }
        }

        for(i=0;i<Q;i++)
        {
                InputStates[i]=NewInputStates[i];
        }
}
/*-------------------------------------------------------------------------------*/
/* simulation procedure                                                  */
/*-------------------------------------------------------------------------------*/
void Simulate()
{
int i;
  ERRORS=0;
  REGPOS=0;
  input=0;
  COUNT=0;
  BER=0;
  while(!kbhit())
  {
          /*get the inputs*/
          GetBranchMetrics();

          /*calculate path metrics*/
          CalcPartialMerics();

          /*copies path metrics*/
          IncomingMetrics();

         /*select max path metric*/
          SelectMaxMetrics();

          /*register exchange*/
          RegisterExchange();
```

```
            /*insert register inputs */
            UpdateRegisters();

            /*update input state path metrics*/
            UpdateInputs();

            /*update register position indicator*/
            REGPOS++;
            if(REGPOS==DEPTH) REGPOS=0;

            /*print results to screen*/
            COUNT++;
            printf(" %.0f %.0f %.0f",(float)ERRORS,(float)COUNT,(float)BER);
          }
}

/*-------------------------------------------------------------------------------*/
/* function main: get output filename and standard deviation  and print  */
/* results to file                                                               */
/*-------------------------------------------------------------------------------*/
int main()
{
float stddev;
FILE *out;
char filename[20];

          clrscr();
          printf("Output file: ");
          scanf("%s",filename);
          if ((out = fopen(filename, "a"))== NULL)
          {
            fprintf(stderr, "Cannot open output file.\n");
            return 1;
          }

          printf("\n stddev: ");
          scanf("%f",&stddev);
          STDDEV=(double)stddev;
          clrscr();

          InitInputStates();
          InitRegisters();
          InitSeed();
          InitState();
          Simulate();

          /*print results to file*/
          fprintf(out, "\n%f %f %f %f %f %f %f",(float)STDDEV,(float)COUNT,(float)ERRORS,
                    (float)BER,(float)ERRORS/(float)COUNT,(float)BER/(2*(float)COUNT),
                    (4.0/6.0)*(float)ERRORS/(float)COUNT);
          fclose(out);
          return 0;
}
```

11. Appendix D: ANN Multi-h Decoder

(Source Code in Borland C)

Implementation of ANN multi-h decoder in Figure 16.

```
/*standard include files*/
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>

/*noise generator include files*/
#include <ran1.c>
#include <gasdev.c>

/*global constants*/
#define Q 4                    /*denominator of multi-h set*/
#define M 100                  /*constant used in selection unit*/
#define BM 8                   /*number of branch metrics*/
#define DEPTH 100              /*decoding depth*/
#define POINTS 100             /*number of sampling points for calculation branch metrics*/
float Vbias1=-10;              /*Vbias1*/
float Vbias2=2;                /*Vbias2*/
#define pi 3.1415927

/*global variables*/
double STDDEV;                 /*standard deviation*/
double BER,ERRORS,COUNT,COUNT2,count; /*error rate counters*/
float SN[Q];                   /*level A state neurons*/
float BN[BM];                  /*level B neurons*/
float CN[BM];                  /*level C neurons*/
float DN[BM];                  /*level D TL neurons*/
int HLN[Q];                    /*level D HL neurons*/
int HL1[BM];                   /*level C selection units' HL1 neurons*/
int HL2[BM];                   /*level C selection units' HL2 neurons*/
int AC[Q][BM];                 /*selection unit control signals*/
float TL1[BM];                 /*level C selection units TL1 neurons*/
float TL2[BM];                 /*level C selection units TL2 neurons*/
float VBN;                     /*level D Vbias1 neuron*/
int Outputs[Q];                /*oldest bit of each of the q output registers*/
int FNEW[Q][DEPTH];            /*copy of output registers*/
int F[Q][DEPTH];               /*output registers*/
float Branch[BM];              /*branch metric inputs on level A*/
int PERIOD;                    /*position in tellis: used for calculating branch metrics and indicating
                                 decoding position*/
int STATE;                     /*encoder state*/
int prevreg[Q];                /*keeps register exchange information*/
int Bit[DEPTH],OldBit;         /*circular list for keeping last DEPTH encoder input bit, oldest bit*/
int REGPOS;                    /*position indicator in Bit[DEPTH]*/
int bit;                       /*encoder input*/
float path1[BM][2];            /*phase trajectories of multi-h code*/
long *seed;                    /*seed for noise generator*/
long PNGEN;                    /*memory for maximum length sequence generator */
```

```
/*-------------------------------------------------------------------*/
/* initialise seed for noise generator                               */
/*-------------------------------------------------------------------*/
void InitSeed()
{
        *(seed)=-123497432;
}

/*-------------------------------------------------------------------*/
/* initialise encoder state for generating a transmitted signal*/
/*-------------------------------------------------------------------*/
void InitState()
{
        STATE=2;
}
/*-------------------------------------------------------------------*/
/* initialise selection unit control signals A                       */
/*-------------------------------------------------------------------*/
void InitAC()
{
AC[0][0]=1;
AC[0][1]=0;
AC[0][2]=1;
AC[0][3]=1;
AC[0][4]=0;
AC[0][5]=1;
AC[0][6]=1;
AC[0][7]=0;
AC[1][0]=0;
AC[1][1]=0;
AC[1][2]=0;
AC[1][3]=1;
AC[1][4]=1;
AC[1][5]=1;
AC[1][6]=0;
AC[1][7]=0;
AC[2][0]=1;
AC[2][1]=0;
AC[2][2]=1;
AC[2][3]=1;
AC[2][4]=0;
AC[2][5]=1;
AC[2][6]=1;
AC[2][7]=0;
AC[3][0]=1;
AC[3][1]=1;
AC[3][2]=1;
AC[3][3]=0;
AC[3][4]=0;
AC[3][5]=0;
AC[3][6]=1;
AC[3][7]=1;
}
```

```
/*-------------------------------------------------------------------*/
/* initialise trellis branches  (depth = 0)                          */
/*-------------------------------------------------------------------*/
void InitPath0()
{
/*A*/
        path1[0][0]=pi;
        path1[0][1]=-(pi*3)/4;
/*B*/
        path1[1][0]=pi;
        path1[1][1]=(pi*3)/4;
/*C*/
        path1[2][0]=pi/2;
        path1[2][1]=(pi*3)/4;
/*D*/
        path1[3][0]=pi/2;
        path1[3][1]=pi/4;
/*E*/
        path1[4][0]=0;
        path1[4][1]=pi/4;
/*F*/
        path1[5][0]=0;
        path1[5][1]=-pi/4;
/*G*/
        path1[6][0]=-pi/2;
        path1[6][1]=-pi/4;
/*H*/
        path1[7][0]=-pi/2;
        path1[7][1]=-(pi*3)/4;
}

/*-------------------------------------------------------------------*/
/* initialise trellis branches  (depth = 1)                          */
/*-------------------------------------------------------------------*/
void InitPath1()
{
/*A*/
        path1[0][0]=(pi*3)/4;
        path1[0][1]=-(pi*3)/4;
/*B*/
        path1[1][0]=(pi*3)/4;
        path1[1][1]=pi/4;
/*C*/
        path1[2][0]=pi/4;
        path1[2][1]=(pi*3)/4;
/*D*/
        path1[3][0]=pi/4;
        path1[3][1]=-pi/4;
/*E*/
        path1[4][0]=-pi/4;
        path1[4][1]=pi/4;
/*F*/
        path1[5][0]=-pi/4;
        path1[5][1]=-(pi*3)/4;
/*G*/
        path1[6][0]=-(pi*3)/4;
        path1[6][1]=-pi/4;
```

```
/*H*/
        path1[7][0]=-(pi*3)/4;
        path1[7][1]=(pi*3)/4;
}

/*-------------------------------------------------------------------*/
/* initialise trellis branches  (depth = 2)                          */
/*-------------------------------------------------------------------*/
void InitPath2()
{
/*A*/
        path1[0][0]=(pi*3)/4;
        path1[0][1]=pi;
/*B*/
        path1[1][0]=(pi*3)/4;
        path1[1][1]=pi/2;
/*C*/
        path1[2][0]=pi/4;
        path1[2][1]=pi/2;
/*D*/
        path1[3][0]=pi/4;
        path1[3][1]=0;
/*E*/
        path1[4][0]=-pi/4;
        path1[4][1]=0;
/*F*/
        path1[5][0]=-pi/4;
        path1[5][1]=-pi/2;
/*G*/
        path1[6][0]=-(pi*3)/4;
        path1[6][1]=-pi/2;
/*H*/
        path1[7][0]=-(pi*3)/4;
        path1[7][1]=pi;
}

/*-------------------------------------------------------------------*/
/* initialise trellis branches  (depth = 3)                          */
/*-------------------------------------------------------------------*/
void InitPath3()
{
/*A*/
        path1[0][0]=pi;
        path1[0][1]=-pi/2;
/*B*/
        path1[1][0]=pi;
        path1[1][1]=pi/2;
/*C*/
        path1[2][0]=pi/2;
        path1[2][1]=pi;
/*D*/
        path1[3][0]=pi/2;
        path1[3][1]=0;
/*E*/
        path1[4][0]=0;
        path1[4][1]=pi/2;
```

```
/*F*/
        path1[5][0]=0;
        path1[5][1]=-pi/2;
/*G*/
        path1[6][0]=-pi/2;
        path1[6][1]=0;
/*H*/
        path1[7][0]=-pi/2;
        path1[7][1]=pi;
}

/*-------------------------------------------------------------------*/
/* 31 bit pseudo noise generator for generating transmitted  */
/*bit sequence : returns a bit                                       */
/*-------------------------------------------------------------------*/
int GetBit()
{
int bit,newbit,j;
        if(PNGEN==0) PNGEN=1234;
        bit=PNGEN&0x01;
        newbit=(PNGEN&0x00000001)^((PNGEN>>3)&0x00000001);
        PNGEN=PNGEN>>1;
        PNGEN=PNGEN|(newbit<<30);
return bit;
}

/*-------------------------------------------------------------------*/
/* TL-neuron function                                                */
/*-------------------------------------------------------------------*/

float TL(float left,float right)
{
float output=0;
        if(left+right>0)
        output=left+right;
return output;
}
/*-------------------------------------------------------------------*/
/* HL-neuron function                                                */
/*-------------------------------------------------------------------*/

int HL(float in1,float in2)
{
int output=0;
        if((in1+in2)>0)
        output=1;
return output;
}
```

```
/*-------------------------------------------------------------------------*/
/*Calculates branch metric for current position in trellis     */
/*-------------------------------------------------------------------------*/
void GetBranchMetrics()
{
int i,j,branch,position;
enum branches {A=0,B=1,C=2,D=3,E=4,F=5,G=6,H=7};
double noise,Noise,corr,corr1,corr2;
float slope[BM],sloperef,Corr[BM],phase[BM],phaseref;

/*.GENERATE ENCODER INPUT BIT */
        bit=GetBit();
        if(REGPOS==DEPTH-1)OldBit=Bit[0];
        else OldBit=Bit[REGPOS+1];
        Bit[REGPOS]=bit;

/*.FIND ENCODER NEXT STATE AND THE BRANCH*/
        if(PERIOD==0)/*first period*/
        {
         InitPath0();
                if(bit==0)
                {
                        switch(STATE)
                        {
                                case 0 : branch = B;  STATE = 0;
                                break;
                                case 1 : branch = D;  STATE = 1;
                                break;
                                case 2 : branch = F;  STATE = 2;
                                break;
                                case 3 : branch = H;  STATE = 3;
                                break;
                                default : printf("state error");
                        }
                }
                else
                {
                        switch(STATE)
                        {
                                case 0 : branch = A; STATE = 3;
                                break;
                                case 1 : branch = C; STATE = 0;
                                break;
                                case 2 : branch = E; STATE = 1;
                                break;
                                case 3 : branch = G; STATE = 2;
                                break;
                                default : printf("state error");
                        }
                }
        }
```

```
else
if(PERIOD==1)
{
        InitPath1();
        if(bit==0)
        {
                switch(STATE)
                {
                        case 0 : branch = B;  STATE = 1;
                        break;
                        case 1 : branch = D;  STATE = 2;
                        break;
                        case 2 : branch = F;  STATE = 3;
                        break;
                        case 3 : branch = H;  STATE = 0;
                        break;
                        default : printf("state error");
                }
        }
        else
        {
                switch(STATE)
                {
                        case 0 : branch = A; STATE = 3;
                        break;
                        case 1 : branch = C; STATE = 0;
                        break;
                        case 2 : branch = E; STATE = 1;
                        break;
                        case 3 : branch = G; STATE = 2;
                        break;
                        default : printf("state error");
                }
        }
}
else
if(PERIOD==2)
{
InitPath2();

        if(bit==0)
        {
                switch(STATE)
                {
                        case 0 : branch = B;  STATE = 1;
                        break;
                        case 1 : branch = D;  STATE = 2;
                        break;
                        case 2 : branch = F;  STATE = 3;
                        break;
                        case 3 : branch = H;  STATE = 0;
                        break;
                        default : printf("state error");
                }
        }
```

```
        else
        {
                switch(STATE)
                {
                        case 0 : branch = A; STATE = 0;
                        break;
                        case 1 : branch = C; STATE = 1;
                        break;
                        case 2 : branch = E; STATE = 2;
                        break;
                        case 3 : branch = G; STATE = 3;
                        break;
                        default : printf("state error");
                }
        }
}
else
if(PERIOD==3)
{
        InitPath3();

        if(bit==0)
        {
                switch(STATE)
                {
                        case 0 : branch = B;  STATE = 1;
                        break;
                        case 1 : branch = D;  STATE = 2;
                        break;
                        case 2 : branch = F;  STATE = 3;
                        break;
                        case 3 : branch = H;  STATE = 0;
                        break;
                        default : printf("state error");
                }
        }
        else
        {
                switch(STATE)
                {
                        case 0 : branch = A; STATE = 3;
                        break;
                        case 1 : branch = C; STATE = 0;
                        break;
                        case 2 : branch = E; STATE = 1;
                        break;
                        case 3 : branch = G; STATE = 2;
                        break;
                        default : printf("state error");
                }
        }
}
```

```
/*.CALCULATE BRANCH METRICS*/
        /*calculate input signal slope*/
        sloperef=path1[branch][1]-path1[branch][0];
        /*calculete slope of referece signals (basis functions)*/
        for(i=0;i<BM;i++)
        {
                slope[i]=path1[i][1]-path1[i][0];
        }
        /*initialise correlation values to zero*/
        for(i=0;i<BM;i++)
        {
                Corr[i]=0;
        }
        for(i=0;i<POINTS;i++)
        {
                /*calculate input signal phase*/
                phaseref=path1[branch][0]+(sloperef*i/POINTS);

                /*get noise sample*/
                noise=gasdev(seed)*STDDEV;

                /*calculate bit error rate for check on noise generator*/
                if(bit==0&&noise>0.7071067)
                BER++;
                if(bit==1&&noise<-0.7071067)
                BER++;
                COUNT2++;

                /*change noise value for sampling*/
                noise=noise*sqrt(POINTS);

                /*calculate branch metrics*/
                for(j=0;j<BM;j++)
                {
                        phase[j]=path1[j][0]+(slope[j]*i/POINTS);
                        corr1=cos(phaseref+i)+noise;
                        corr2=cos(phase[j]+i);
                        corr=corr1*corr2;
                        Corr[j]=Corr[j]+corr;
                }
        }

       /*scale branch metrics (multiply by Ts)*/
        for(i=0;i<BM;i++)
        {
                Corr[i]=Corr[i]/POINTS;
        }

        /*add Vbias2  to branch metrics*/
        for(i=0;i<BM;i++)
        {
                Branch[i]=TL(Corr[i],Vbias2);
        }

}
```

```
/*-------------------------------------------------------------------*/
/* simulation procedure                                              */
/*-------------------------------------------------------------------*/
void Simulate()
{
 int i,j,wait=0;   /*counters*/
 int depth;        /*depth in trellis = PERIOD-1 (modulo-3) */
 int s0,s1,s2,s3;  /*counters*/
 int OE0_0,OE1_0,OE3_0,OE0_1,OE1_1,OE2_1,OE1_2,OE2_2,OE3_2,OE0_3,OE2_3,OE3_3;
                   /*register output enable signals*/
 double Noise,corr;  /*gaussian random variable, correlation value*/
 PERIOD=0;
 REGPOS=0;
 ERRORS=0;
 COUNT=0;
 COUNT2=0;
 while(!kbhit())
 {
        COUNT++;
/*.CALCULATE DIFFERENCE BETWEEN D[0] AND Vbias1*/
        VBN=TL(Vbias1,DN[0]);

/*.SUBTRACT DIFFERENCE FROM ALL OLD PATH METRICS*/
        SN[0]=TL(DN[0],-VBN);
        SN[1]=TL(DN[2],-VBN);
        SN[2]=TL(DN[4],-VBN);
        SN[3]=TL(DN[6],-VBN);

/*.CALCULATE BRANCH METRIC INPUTS*/
        GetBranchMetrics();

/*.CALCULTE NEW PATH METRICS */
        BN[0]=TL(Branch[1],SN[0]); /*B0*/
        BN[1]=TL(Branch[2],SN[1]); /*C1*/
        BN[2]=TL(Branch[7],SN[3]); /*H3*/
        BN[3]=TL(Branch[0],SN[0]); /*A0*/
        BN[4]=TL(Branch[3],SN[1]); /*D1*/
        BN[5]=TL(Branch[4],SN[2]); /*E2*/
        BN[6]=TL(Branch[5],SN[2]); /*F2*/
        BN[7]=TL(Branch[6],SN[3]); /*G3*/

/*.INPUT SELECTION (depth dependent)*/
/*..CALCULATE depth*/
        if(PERIOD==3)depth=0;
        else depth=PERIOD+1;

/*..SELECT INPUTS USING SELECTION UNIT CONTROL SIGNALS (AX)*/
        HL1[0]=HL(AC[depth][0],0);
        HL2[0]=HL(-HL1[0],0.5);
        TL1[0]=TL(-M*HL1[0],BN[0]);
        TL2[0]=TL(-M*HL2[0],BN[2]);
        CN[0]=TL(TL1[0],TL2[0]);
```

```
        HL1[1]=HL(AC[depth][1],0);
        HL2[1]=HL(-HL1[1],0.5);
        TL1[1]=TL(-M*HL1[1],BN[1]);
        TL2[1]=TL(-M*HL2[1],BN[3]);
        CN[1]=TL(TL1[1],TL2[1]);

        HL1[2]=HL(AC[depth][2],0);
        HL2[2]=HL(-HL1[2],0.5);
        TL1[2]=TL(-M*HL1[2],BN[4]);
        TL2[2]=TL(-M*HL2[2],BN[0]);
        CN[2]=TL(TL1[2],TL2[2]);

        HL1[3]=HL(AC[depth][3],0);
        HL2[3]=HL(-HL1[3],0.5);
        TL1[3]=TL(-M*HL1[3],BN[1]);
        TL2[3]=TL(-M*HL2[3],BN[5]);
        CN[3]=TL(TL1[3],TL2[3]);

        HL1[4]=HL(AC[depth][4],0);
        HL2[4]=HL(-HL1[4],0.5);
        TL1[4]=TL(-M*HL1[4],BN[4]);
        TL2[4]=TL(-M*HL2[4],BN[6]);
        CN[4]=TL(TL1[4],TL2[4]);

        HL1[5]=HL(AC[depth][5],0);
        HL2[5]=HL(-HL1[5],0.5);
        TL1[5]=TL(-M*HL1[5],BN[5]);
        TL2[5]=TL(-M*HL2[5],BN[7]);
        CN[5]=TL(TL1[5],TL2[5]);

        HL1[6]=HL(AC[depth][6],0);
        HL2[6]=HL(-HL1[6],0.5);
        TL1[6]=TL(-M*HL1[6],BN[2]);
        TL2[6]=TL(-M*HL2[6],BN[6]);
        CN[6]=TL(TL1[6],TL2[6]);

        HL1[7]=HL(AC[depth][7],0);
        HL2[7]=HL(-HL1[7],0.5);
        TL1[7]=TL(-M*HL1[7],BN[3]);
        TL2[7]=TL(-M*HL2[7],BN[7]);
        CN[7]=TL(TL1[7],TL2[7]);

/*.SELECT MAX PATH METRIC PER STATE (SURVIVING PATHS)*/
        DN[1]=TL(-CN[0],CN[1]);
        DN[3]=TL(-CN[2],CN[3]);
        DN[5]=TL(-CN[4],CN[5]);
        DN[7]=TL(-CN[6],CN[7]);
        DN[0]=TL(CN[0],DN[1]);
        DN[2]=TL(CN[2],DN[3]);
        DN[4]=TL(CN[4],DN[5]);
        DN[6]=TL(CN[6],DN[7]);

/*.FIND HL NEURON OUTPUTS (LEVEL D)*/
        HLN[0]=HL(DN[1],0);
        HLN[1]=HL(DN[3],0);
        HLN[2]=HL(DN[5],0);
        HLN[3]=HL(DN[7],0);
```

```
/*.REGISTER EXCHANGE*/

/*..GENERATE OUTPUT ENABLE SIGNALS AND EXCHANGE SEQUENCES*/
        OE0_0=(!HLN[0]&&!AC[depth][0]) || (HLN[0] && AC[depth][1]);
        OE1_0=(HLN[0]&&!AC[depth][1]);
        OE3_0=(!HLN[0]&&AC[depth][0]);

        s0=0;
        if(OE0_0)
        {
                prevreg[0]=0;
                s0++;
        }
        if(OE1_0)
        {
                prevreg[0]=1;
                s0++;
        }
        if(OE3_0)
        {
                prevreg[0]=3;
                s0++;
        }
        if(s0>1)
        {
                printf("ERROR: bus 0 contention");  /*chech that two registers are not enabled at
                                                        once*/
                getch();
        }

        OE0_1=(!HLN[1]&&AC[depth][2]);
        OE1_1=(!HLN[1]&&!AC[depth][2]) || (HLN[1] && !AC[depth][3]);
        OE2_1=(HLN[1]&&AC[depth][3]);

        s1=0;
        if(OE0_1)
        {
                prevreg[1]=0;
                s1++;
        }
        if(OE1_1)
        {
                prevreg[1]=1;
                s1++;
        }
        if(OE2_1)
        {
                prevreg[1]=2;
                s1++;
        }
        if(s1>1)
        {
                printf("ERROR: bus 1 contention");
                getch();
        }
```

```
OE1_2=(!HLN[2]&&!AC[depth][4]);
OE2_2=(!HLN[2]&&AC[depth][4]) || (HLN[2] && !AC[depth][5]);
OE3_2=(HLN[2]&&AC[depth][5]);
s2=0;
if(OE1_2)
{
        prevreg[2]=1;
        s2++;
}
if(OE2_2)
{
        prevreg[2]=2;
        s2++;
}
if(OE3_2)
{
        prevreg[2]=3;
        s2++;
}
if(s2>1)
{
        printf("ERROR: bus 2 contention");
        getch();
}

OE0_3=(HLN[3]&&!AC[depth][7]);
OE2_3=(!HLN[3]&&AC[depth][6]);
OE3_3=(HLN[3]&&AC[depth][7]) || (!HLN[3] && !AC[depth][6]);

s3=0;
if(OE0_3)
{
        prevreg[3]=0;
        s3++;
}
if(OE2_3)
{
        prevreg[3]=2;
        s3++;
}
if(OE3_3)
{
        prevreg[3]=3;
        s3++;
}
if(s3>1)
{
        printf("ERROR: bus 3 contention");
        getch();
}
```

```
/*..COPY REGISERS, DO EXCHANGE AND SHIfT BY ONE BIT*/
        for(i=0;i<Q;i++)
        {
                for(j=1;j<DEPTH;j++)
                {
                        FNEW[i][j]=F[prevreg[i]][j-1];
                }
        }

/*..COLLECT OUPUTS*/
        for(i=0;i<Q;i++)
        {
                Outputs[i]=FNEW[i][DEPTH-1];
        }
        printf("\n%d%d%d%d",Outputs[0],Outputs[1],Outputs[2],Outputs[3]);

/*..INSERT NEW INPUTS*/
        for(i=0;i<Q;i++)
        {
                FNEW[i][0]=HLN[i];
        }

/*..COPY BACK TO OUTPUT REGISTERS*/
        for(i=0;i<Q;i++)
        {
                for(j=0;j<DEPTH;j++)
                {
                        F[i][j]=FNEW[i][j];
                }
        }

/*.SIMULATION ERROR COUNTS*/
        if(wait<DEPTH)wait++;
        if(wait==DEPTH)
        {
                if(Outputs[0]!=OldBit)
                ERRORS++;
                printf(" %d %.0f %.0f %.0f %f f",OldBit,(float)ERRORS,(float)BER,(float)COUNT,
                (float)ERRORS/(float)COUNT, (float)BER/(float)COUNT2);
        }

/*.UPDATE POSITION IN TRELLIS (DEPTH)*/
        PERIOD++;
        if(PERIOD==4)PERIOD=0;

/*.UPDATE POSITION INDICATOR FOR INPUT BIT SEQUENCE CYCLIC REGISTER*/
        REGPOS++;
        if(REGPOS==DEPTH)REGPOS=0;

 }

}
```

```
/*-------------------------------------------------------------------*/
/* function main: read output file name and standard                 */
/*deviation from console and print results to file                   */
/*-------------------------------------------------------------------*/
int main()
{
float stddev;
FILE *out;
char filename[20];
        clrscr();
        printf("Output file: ");
        scanf("%s",filename);
        if ((out = fopen(filename, "a"))
                == NULL)
        {
          fprintf(stderr, "Cannot open output file.\n");
          return 1;
        }
        printf("\n stddev: ");
        scanf("%f",&stddev);
        STDDEV=(double)stddev;
        clrscr();

        InitSeed();
        InitAC();
        InitState();
        Simulate();
        count=COUNT-DEPTH;
        fprintf(out, "\n%f %f %f %f %f %f",(float)STDDEV,(float)COUNT,(float)ERRORS,
                (float)BER,(float)ERRORS/(float)COUNT,(float)BER/(float)COUNT2);
        fclose(out);

return 0;
}
```