Design of a Simulation for Trellis Coded Modulation

by

Theo Lindebaum

B.Sc(Eng), University of Cape Town, 1990

A Dissertation Submitted to the Faculty of Engineering,
University of Cape Town,
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering

Cape Town, South Africa

1992

Design of a Simulation for Trellis Coded Modulation

by

Theo Lindebaum

I hereby certify that this is my own original work and has not been submitted before for any other degree.

Signed by candidate

Signature removed

Candidate

5/10/92

Date

Recommended for Submission:

Supervisor

5/10/92

Date

PREFACE

The last decade has shown a very rapid increase in the application of digital computers to virtually all walks of life. This has resulted in a demand for better and faster digital communication links, at reasonable prices. Trellis coded modulation is a fairly recent development in the technology of digital communications that has helped to meet an aspect of this demand.

The Department of Electrical Engineering at the University of Cape Town had not undertaken any research in the field of trellis coded modulation at the time this thesis was started. From what was understood of the topic at the time, it was deemed important to develop a foundation of skills and tools for working with trellis coded modulation. It was hoped that such a foundation will give current and future members of the department sufficient background to spot potential applications for the principles of trellis coded modulation.

It was felt that the development of a simulation would provide a forum for study on the topic of trellis coded modulation as well as provide some basic simulation tools for future investigations into the topic. This thesis is a direct result. Some implementation goals were set to provide both direction and a defined scope for the thesis.

- The format of the implementation should be such that the simulation can be considered a re-usable tool.

- The simulation aspect should focus on the trellis coded modulation aspects of the overall system.

- The simulation should be applied to a known system to verify the results of the simulation against known data.

ACKNOWLEDGEMENTS

The author wishes to thank the following people for their part in the presentation of this thesis.

Dr Robin Braun - for willingly agreeing to be the supervisor of this thesis. He provided plenty of ideas and support during the implementation of this thesis.

Jaz Schoonees - a fellow student. Jaz provided an excellent sounding board for my ideas, and helped with a few ideas of his own. His many questions and answers helped ensure that the author thought about the problems at hand. Jaz also helped with proof reading of this document.

Johan van de Groenendaal - a fellow student. who had a sharp eye for errors when proof reading this document

Contents

List of Figures

List of Tables

INTRODUCTION

Trellis Coded Modulation (TCM) is a technique where the performance of digital communication systems can be improved over what was previously possible. The key feature of TCM is that it *combines* convolutional coding with signal space mapping. This is achieved by expanding the signal constellation that is used to carry the information on the channel. Practical implementations of TCM systems must develop suitable codes and signal space mappings in order to achieve coding gains. Other requirements such as rotational invariance place further constraints on the design of TCM systems.

TCM is a fairly well established technology, and has been applied to real life systems. An example of such a system is a modem supporting the CCITT[1] recommendation V.32. The V.32 recommendation is a document specifying a standard for modems that can operate at data rates up to 9600 bits/s on a standard telephone line. Included in the V.32 recommendation are the specifications for an implementation of trellis coded modulation system and an alternative 16-QAM modulation system.

This document discusses a simulation developed to evaluate the performance of TCM implementations. The concepts and basics of TCM are discussed in chapter 1. The relevant parts of the CCITT recommendation V.32 are discussed in chapter 3.

The TCM system of the CCITT recommendation V.32 is used as an example to evaluate the simulation programs. This allowed the simulation results to be com-

[1]CCITT - International Consultative Committee for Telegraphy and Telephony.

pared with previously published results. In 1984 Wei[12] detailed an implementation of a TCM system that had the desirable characteristic of being rotationally invariant. Chapter 2 provides details of such rotationally invariant codes.

The simulation has been implemented using a modular structure where each module is a stand-alone program that performs some specific processing function within the overall simulation. This was done to allow the various program modules to be re-used for future simulations using TCM type of structures. Chapter 4 discusses the implementation details of the simulation.

The simulation was applied to the TCM and the 16-QAM alternative systems specified in the V.32 recommendation. Chapter 5 details the simulation, the expected results and the results obtained using the simulation. In addition, a simulation of the effects of phase jitter is implemented to illustrate the application of the simulation tools for different simulations.

Finally, some general comments and conclusions are made in chapter 6. The implementation of the simulation is deemed to meet the goals and requirements set out in the preface of this document.

The appendices provide specific information about the programs developed for the simulation. Readers who are going to use the simulation tools must read appendices C and D. Readers interested in implementing new program modules are recommended to read appendix B and to consult [3]. A diskette is provided with this dissertation, and contains the source code for all of the program modules that were implemented.

CHAPTER 1

FUNDAMENTALS OF TCM

Trellis coded modulation (TCM) is the combination of information coding with modulation in order to achieve a better system reliability without requiring more bandwidth or signal power. This chapter introduces some of the concepts necessary for understanding why TCM is used, and how TCM achieves better performance.

Consider a digital transmission scheme that transmits 2 bits of information with every symbol, during an interval T, using PSK modulation. There are several options to achieve this goal :

1. Use a 4-PSK modulation scheme with no coding. One symbol is transmitted every T seconds and carries two bits of information. This is the reference system for the other coded techniques.

2. Apply a convolution code to the digital data with (for example) a 2/3 rate and modulate with a 4-PSK signal. Two bits of digital information is packaged into 3 bits resulting in each symbol transmitted containing 2/3 * 2 = 4/3 bits of information. Thus to keep the same data rate, the symbols must be transmitted every 2/3 T seconds. This requires extra bandwidth.

3. Apply a convolutional code with a rate of say 2/3, and modulate with an 8-PSK signal. Each symbol on the 8-PSK still contains 2 bits of information, maintaining the symbol interval of T seconds to carry the data at the required rate.

There are conceivably other options, but they shall not be considered here.

Option (3) would require an increase in power to maintain the same error performance if there was no benefit obtained by the coding. This is because for an uncoded scheme the error performance is related to the distance between adjacent symbols in the PSK constellation. To maintain the same spacing as 4-PSK, an 8-PSK constellation would require 3dB of extra power.

In uncoded modulation the error performance is determined by the distance between valid mappings in the signal constellation. The demodulator must estimate the true symbol represented by the received signal. This is done by mapping the measured received point to the nearest valid point in the constellation. But how is the nearest point measured? In the case of PSK, the actual distance of the measured point and the nearest valid constellation point is required. This is known as the *Euclidian distance*.

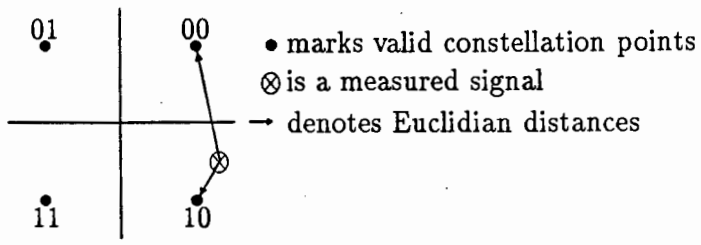


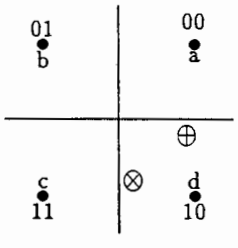
Figure 1.1: *Constellation Diagram for 4-PSK*

In this uncoded modulation scheme, the nearest point is chosen to be the best estimate of the transmitted symbol. There exists a minimum Euclidian distance between any combination of two points of the symbol mappings. It is this distance that defines the error performance of the modulation system. The greater the distance, the larger the noise contribution can be tolerated before incorrect decisions are made. The only way to increase the minimum Euclidian distance between

constellation points is to either increase the size of the constellation (which requires more power), or to reduce the number of points in the constellation (which reduces the information carrying capacity of each symbol).

When coding is introduced, the measure of the received signal and a valid signal is no longer done on a symbol by symbol basis but rather on a *sequence* of symbols. In the case of the convolutional code of option (2) the measure of the 'distance' between a proposed sequence and the measured sequence is done via either Hamming[1] or Euclidian distances to the received data. When decoding convolutional codes, the quantisation of the received signal to received symbols (as required for Hamming distance measure) is known as *hard decision decoding.* The use of Euclidian distance when decoding the convolutional code is known as *soft decision decoding.*

Notice that the Hamming distance is not linearly related to the Euclidian distance between received sequences of points. Thus there is a irrecoverable loss of information using hard decision decoding. To illustrate this consider Figure (1.2) below:



• valid consellation points

⊗, ⊕ received points

Points ⊕ and ⊗ would both be taken to represent symbol d in a memoryless demodulator. The hamming distance between symbols c and d is the same as the hamming distance between a and d. However point ⊗ has a much smaller euclidian distance to symbol c than it does to symbol a.

Figure 1.2: *Illustration of Information loss using hard decision decoding*

Effectively, the hard decision decoding quantises the received signal into the range of values defined by the valid signal points. This coarse quantisation loses

[1]Hamming distance between two binary numbers is the count of the number of bits in which the numbers differ.

all the information contained by true position of the received point being between quantisation levels.

Trellis codes (such as example 3) *combine the coding and modulation into a single entity.* The demodulation and decoding is also combined. A key advantage of this system over the convolutional coding of option (2) is that the coding information has been added without increasing the bandwidth of the signal. As with the coding scheme of option (2), the decoding is done on a *sequence* of symbols. The measure of a proposed sequence is done using the Euclidian distance (i.e. soft decision decoding) between a proposed sequence of symbol points and the measured sequence of symbol points.

An essential part of using trellis codes is that valid sequences have to be chosen from a *subset* of all possible combinations. If this was not the case, every possible symbol would be a valid option at each sampling interval. This would effectively reduce the decoding to the symbol by symbol decoding of option 1). In order to choose a subset of symbols at any one time a memory element is required. The contents, or *state* of this memory will depend upon the previous data received.

The improvement in performance discussed here is achieved by changing the assumption that consecutive signals are independent of each other. This is done by restricting the valid sequences of data to a subset of all the possible sequences that would be allowed if the signals *were* independent.

Consider $\boldsymbol{q}$, a sequence of k symbols, each chosen out of a set $\boldsymbol{A} = \{0, 1, \ldots M-1\}$. $\boldsymbol{q}$ may be represented as a vector $\boldsymbol{q} = (a_1, a_2, \ldots a_k)$ where a_i is an element of $\boldsymbol{A}$. If a_n is independent of any a_{n-1} there would be a total of M^k possible sequences.

By introducing a dependence on previous signal elements we introduce a new set of sequences that are a subset of the one above. However, rather than reduce the information carried by the subset, the subset is designed to keep the same number of elements, M, chosen out of a larger set of symbols. Hence the need to go to a higher order of modulation.

The selection of the larger set of symbols is achieved by using a convolutional code of rate $m/(m+\alpha)$ where $\alpha \geq 1$. TCM in practice is limited to using $\alpha = 1$ because of the difficulties involved with demodulating higher order signal constellations.

In summary, consider a sequence of source symbols $a_1, a_2, \ldots a_k$. A convolutional encoder generates a new symbol dependant on a finite number of previous input symbols L, denoted as

$$x_n = f(a_{n-L}, a_{n-L+1}, \ldots a_n) \tag{1.1}$$

where $x_n \in \boldsymbol{X}$. Alternatively we can define a state σ_n for the encoder that is a function of the L previous input elements.

$$\sigma_n = h(a_{n-L}, a_{n-L+1}, \ldots a_{n-1}) \tag{1.2}$$

This allows us to simplify equation (1.1) to

$$x_n = f(a_n, \sigma_n) \tag{1.3}$$

The next state of the convolutional encoder depends only on the current state and the new input symbol.

$$\sigma_{n+1} = g(\sigma_n, a_n) \tag{1.4}$$

The number of symbols of x_n in $\boldsymbol{X}$ is larger than the number of symbols required to represent all the a_n in $\boldsymbol{A}$. The output signal element for x_n is chosen out of a set

of signals Ω' that has a one to one mapping on $\boldsymbol{X}$. The signals in Ω' are *partitioned* to allow a signal selection that is dependent on the current state σ_n of the encoder.

The state and selection of symbols can be represented graphically by using *trellis diagrams*. Each possible state σ_n at time unit n is represented by the vertical position of the dots or *nodes* in the trellis diagram. *Branches* join the nodes from a state at time n to a state at time $(n+1)$ and are labelled according to value $f(a_n, \sigma_n)$ described above. The state transitions are defined by the function $g(\sigma_n, a_n)$ defined above.

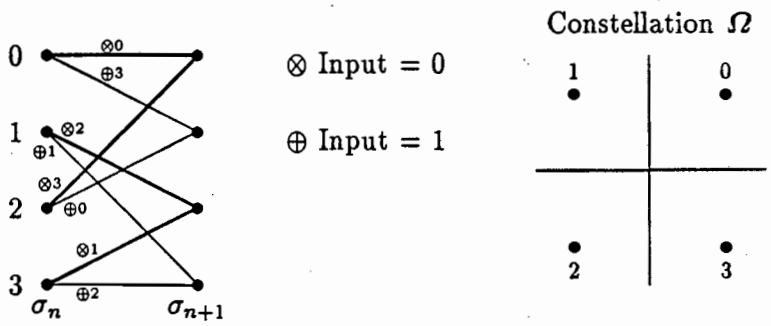


Figure 1.3: *Example of a 4 state trellis describing a TCM code with 4 states and transmitting from a source set of binary symbols.*

A trellis diagram for a code of M source symbols will *always* have M branches leaving each node, because all possible inputs $a_n \in \boldsymbol{A}$ have to be accommodated. Thus it is possible to have two or more branches joining the same pair of states (but representing different source symbols). These are known as *parallel transitions*. Each branch originating from a node is associated with a source symbol. (The symbols associations for the binary source in the above example are indicated by the $\oplus$'s and $\otimes$'s on the trellis diagram.) These indicate which source symbol is required to generate the state transition for that branch. To illustrate the trellis diagrams and the generation of TCM codes, consider the circuit of figure(1.4) :

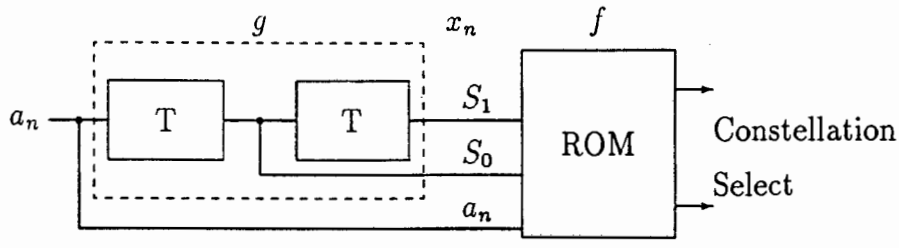
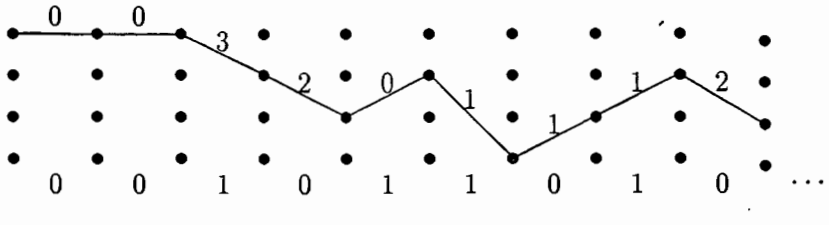


Figure 1.4: *Example of simple Convolutional Encoder*

The two unit delays implement the convolutional encoder for the code illustrated in the trellis diagram of figure(1.3). This implements the function $g(\sigma_n, a_n)$ for this encoder. σ_n is the contents of the two unit delays (memory) and defines the state of the encoder. The ROM implements the function f(), and defines the constellation point for each branch on the trellis diagram. For this example, the contents of the ROM would be :

S_1	S_0	a_n	Constellation point
0	0	0	0
0	0	1	3
0	1	0	2
0	1	1	1
1	0	0	3
1	0	1	0
1	1	0	1
1	1	1	2

Assuming that the convolutional encoder starts in state 0, for the binary sequence 001011010 ...the following trellis diagram would be generated.



1.1 SELECTION OF THE CONSTELLATION MAPPING FUNCTION F

Here we pause formally to define two terms:

The *minimum distance* $d_{\min}$ is the minimum Euclidian distance between any pair of possible signals in a constellation Ω.

The *minimum free distance* d_{free} is the minimum Euclidian distance between all possible different sequences of signals in a constellation Ω' that both start in one state and both end in some other state.

$$d_{\text{free}} = \min_{\{a_n\} \neq \{b_n\}} \left(\sum_n d^2 (a_n, b_n) \right)^{\frac{1}{2}} \tag{1.5}$$

$\{a_n\}$ and $\{b_n\}$ are sets of all the possible channel sequences that the encoder can generate. $d^2(a_n, b_n)$ is the Euclidian distance squared between a_n and b_n. Note also that any parallel transitions also define valid paths and must be included in the definition of the free distance.

The whole aim of trellis codes is to introduce a sequence of symbols in the modulated format (i.e. the constellation mappings) that have a large as possible distance between valid sequences. Ungerboeck described in his paper [10] some of the requirements for such a selection of mapping. He called his ideas "Mapping by set Partitioning." Ungerboeck pursues design methods that are aimed at maximizing the free Euclidian distance. The channel signal set is broken down into subsets, with each subset being broken down into further subsets such that $\Delta_0 < \Delta_1 < \Delta_2 \ldots$ where Δ_0 is the minimum distance for the full signal set and Δ_1 is the minimum distance of elements of the 1st etc. This is illustrated in figure(1.5).

The subsets themselves are labelled A_0 for the complete constellation, B_0 and B_1 for the first set of partitions, $C_0, \ldots C_3$ for the second set of partitions etc.

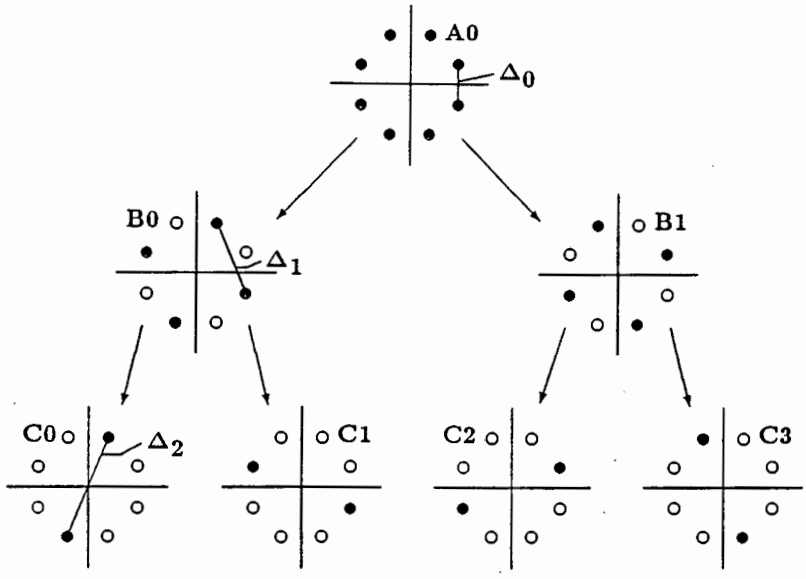
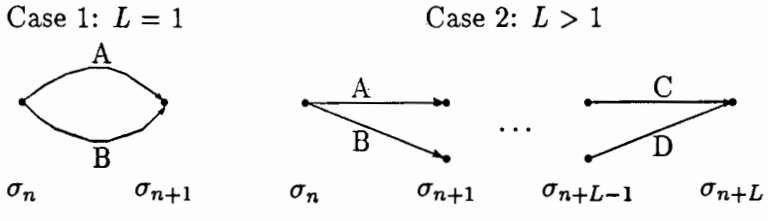


Figure 1.5: *Set Partitioning for an 8-PSK constellation*

Consider a trellis code with signals chosen from four subsets A, B, C and D. Assume the free distance d_{free} is generated on a sequence of L bits.

For the case $L = 1$, there are *parallel transitions*. Here the free distance d_{free} is the same as the minimum distance d_{min} of the subset of signals used. For the case $L > 1$, d_{free} now becomes defined as $d^2_{\text{free}} = d^2_{\text{min}}(A, B) + \ldots d^2_{\text{min}}(C, D)$



For good codes, parallel transitions must be defined from the subsets that have the largest minimum distance between them. This is necessary since parallel transitions set the upper boundary to the minimum free distance d_{free}. Subsets originating from or ending in the same state should have the largest possible minimum distance

between the elements to keep the distances between adjacent paths high as possible.

Ungerboeck [10] established a set of rules in his paper to achieve such requirements. These are known as the *Ungerboeck Rules*.

1. All signals should be used equally often.
2. Parallel transitions are assigned signal members from the same partition.
3. Adjacent transitions are assigned signal members from the next larger partition.

If an $m/(m+1)$ code is used, and the partitioning of figure (1.5) is used, these rules can be restated for this example as follows :

- All signals are used equally often.
- Parallel transitions are assigned numbers from either subsets C0 or C1 or C2 or C3.
- Transitions joining the same state or leaving the same state should be assigned from subset B0 or B1.

1.2 Calculating the Coding Gain of a Trellis Code

After establishing the free distance d_{free} of a code, it is possible to calculate the coding gain of the code. As mentioned earlier, any change in the average power of the resulting signal has to be considered when looking at the code's performance. The *coding gain* is given by

$$\gamma = \frac{d_{\text{free}}^2}{E} / \frac{d_{\text{min}}^2}{E'}$$

where E and E' are the average energies of the coded and uncoded signals respectively.

1.2.1 Error Events

There is another factor to be considered when looking at bit error probabilities. In uncoded transmission, each consecutive symbol is independent of the previous one, and has a finite probability of being decoded incorrectly that is dependant only on the signal to noise ratio of the signal[1]. This implies that a decoding error or *error event* will be independent of other error events. If the signal constellation is mapped so that the nearest neighbours have a low Hamming distance from the adjacent symbols they represent (eg. Gray code) then an error event will *most likely* be a single bit error. (In practice this is not always possible because of other constraints, such as a requirement for rotational invariance).

In coded modulation, an error occurs when a decoded *sequence* differs from the transmitted sequence. This error sequence must be at least L symbols long where L is the minimum sequence length to generate two different paths that diverge and converge again at L symbols later. The sequence of symbols in error form an *error event.* An error event results in a sequence of bit errors. The resulting bit error rate from error events is given by

$$P(e) = N \times P(Ev)$$

where N is the average length in bits of the error event[2] and $P(Ev)$ is the probability of the error event. The value of N is a function of both the average length of the error event (in units of symbols), and the actual mapping of the symbols in the constellation. The mapping affects the average number of bits in error for each symbol that is in error.

[1]For these error probabilities we consider only ideal channels with additive Gaussian noise.

[2]If the upper bound of the minimum free distance is limited by parallel transitions in the trellis code, single event errors will be the most likely of the possible error events.

Notice that $P(e)$ is not simply related to the coding gain. The probability of an error event *is* related to free distance d_{free} of the trellis code. Consequently the coding gain is generally accepted as the most important measure of the performance of a trellis code.

1.3 Decoding Trellis Codes

The decoding of TCM codes is significantly more complicated than the symbol by symbol estimation used in uncoded systems. In a TCM system, the decoder must consider sequences of data. Consider a message of k binary bits. All combinations of data will generate 2^k possible different messages which in turn must generate 2^k possible different sequences from the TCM encoder. Ideally, the decoder of the TCM system will for each of the 2^k possible sequences accumulate the distance (or distance^2) between that sequence and the received sequence. *The accumulated distances between a valid sequence and the measured sequence is known as the branch metric.* The branch metric is derived mathematically from the maximum likelihood function of the trellis code for a signal with additive Gaussian noise. Refer to [1, pages 87–90] for further information. To implement the above procedure is known as *maximum likelihood decoding.*

In practice, the direct implementation of maximum likelihood decoding is impractical, if not impossible, for all but short message lengths. A direct implementation of the above procedure would require the computation and storage of 2^k possible paths and branch metrics. In addition, the data estimate only becomes available after the complete message of k bits has been received.

1.3.1 THE VITERBI ALGORITHM

The Viterbi Algorithm[1][5][4] provides a practical means of implementing maximum likelihood decoding described above.

Consider a trellis code defined by the trellis diagram of figure (1.6). At any

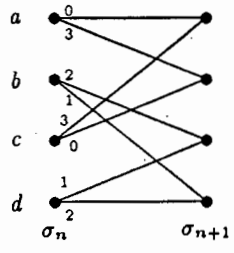


Figure 1.6: *Basic Trellis Diagram*

arbitrary time $n+1$ of the trellis diagram there are two branches entering every state σ_{n+1} from nodes at time = n. Each of these joining branches defines two alternative paths that converge at time $n+1$. All future branches of each path originating from a node at time n are defined only by the current state and the input data arriving at that point in time, as indicated earlier in equation(1.4). Consequently the path metric of all possible future paths passing through a node at time n will be minimum if the path up to the node at n is chosen to have the minimum metric. Thus there is a recursive relationship for computing the minimum metric. The minimum path metric to a node σ_{n+1} can be formed by choosing the branch to σ_{n+1} that has the lowest path metric. The lowest path metric is found by adding the branch metric of the transition to the path metric of the origin node of that transition. At any stage $n+1$ the only branch entering a node that need be stored is the branch that has the lowest metric entering the node at $n+1$. The stored branch is known as the *survivor*. The selection of survivors at each stage

forms the basis of the *Viterbi Algorithm.*

Viterbi decoding of a message of l symbols in length transmitted using a trellis code with α states requires the storage of paths of length l for each of the α states. This is a considerably less than the 2^k paths required for a direct implementation described earlier.

The Viterbi algorithm as described here also has the problem of a delay of l symbols before the data is available. To overcome this problem the *Truncated Viterbi Algorithm* is used. In the truncated Viterbi algorithm a decision is forced before the entire message of l symbols is received, at some length, say d symbols. d is known as the *Truncation Depth.* At stage n of the algorithm, the path with the minimum metric at stage n is backtracked to stage $n - d$ to find the most likely symbol for the current trellis. The binary data (x_{n-d}) for that branch is output from the decoder.

This process generates sub-optimal decoding. However for sufficiently long truncation depths, computer simulation results have shown the loss to be negligible[8]. As the truncation depth d increases, the truncated Viterbi algorithm asymptotically approaches the ideal behaviour of the Viterbi algorithm.

There are two major benefits to the truncated Viterbi algorithm :

- The delay problem is solved. For a message of any length the data becomes available after an initial delay of at most d symbol periods, and thereafter one symbol for every symbol period.
- The memory requirements for the path storage is quantised to α paths each of at most length d.

The Viterbi algorithm can be stated as follows :

Consider a message of L symbols with an encoder of memory M, assuming an initial state of zero.

Step 1: Starting at level (time unit) $j = M$, compute the metric for the single path entering each state of the encoder trellis. Store the path and its associated metric for each of the states.

Step 2: Increment level j by 1. Compute the metric for all the paths entering each state by adding the metric of the incoming branches to the metric of the connecting survivor from the previous time unit. For each state, identify the path with the lowest metric as the survivor of step 2. Store the survivor and its associated metric.

Step 3: If $j < M + L$, repeat step 2, otherwise select the state with the minimum metric at time $L + M$. Backtrack along the stored path of that metric until level time. The decoded data is then found by selecting the data that would follow the stored trellis path.

In order to illustrate the operation of the Viterbi algorithm, we shall deviate from trellis codes for a while and consider a convolutional coded implementation that uses a Viterbi algorithm that applies the *Hamming* distance for estimating the branch metrics of the proposed paths. The code will be assumed to be generated from the code of figure(1.7) :

Now suppose an all zero sequence was transmitted from the encoder but the sequence (00 01 00 10 00 00) was received. Knowing what was transmitted, we can see two errors in the received sequence. Unfortunately the receiver does not have our prior knowledge of the transmitted data, and will have to do its best to decode correctly. To illustrate the workings of the Viterbi algorithm we will build

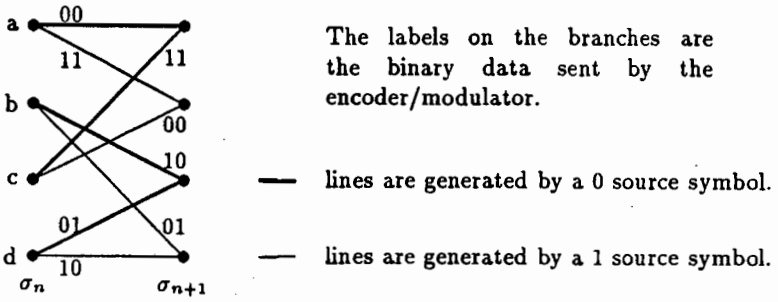


Figure 1.7: *Trellis code used for Illustration of Viterbi Algorithm*

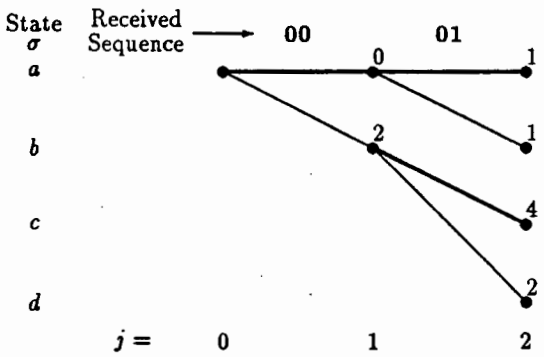


Figure 1.8: *Step 1: build up the trellis to each valid state for the code.*

up a trellis diagram for the decoder. For step 1, the starting state is assumed to be a (i.e. state 0). Using the notation a_0–b_1 to indicate the branch from the node at state a at $j = 0$ to the node at state b at $j = 1$. From figure 1.7, the transmitted symbol for a_n–a_{n+1} is **00**. A **00** was received giving a partial metric for the branch a_0–a_1 of 0. Add this to the metric of node a_0 to give the metric for a_1 of 0. Likewise, the transmitted symbol for a_n–b_{n+1} is **11**. A zero was received giving a partial metric for the branch a_0–b_1 of 2. Add this to the metric of node a_0 (the origin of the branch) to give a metric for b_1 of 2. There are no valid branches to nodes c_1 and d_1, so their metrics cannot be computed.

The process is repeated for time $j = 2$, except now there are valid branches into all the states. This completes the initialising of the trellis in preparation for step 2.

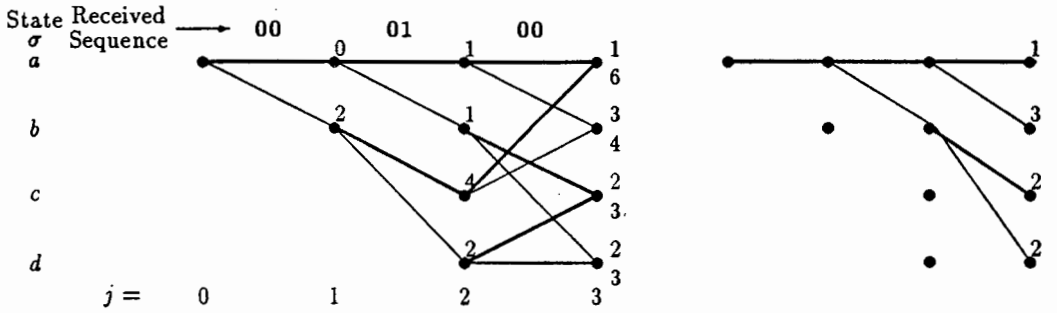


Figure 1.9: *Trellis for first iteration of step 2*

The main part of the Viterbi algorithm now starts. Consider node a_3 for the first iteration of step 2. From figure 1.7 we can see there are two possible entry routes to a_3, viz. c_2–a_3 and a_2–a_3. For the branch a_2–a_3, the transmitted data required is **00**. This has a Hamming distance of 0 from the received sequence of **00**. Add this 0 to the metric for a_2 to give a metric for that branch (known as the *branch metric*). Store the branch metric. Similarly, for the branch c_2–a_3, the transmitted data required is **11**. This has a Hamming distance of 2 from the received sequence of **00**. Add this to the metric for c_2 (which is 4) to give a branch metric 6. Store this branch metric. This completes the computation of the partial metrics for all the nodes entering node a_3. Now select and keep the branch with the *minimum* branch metric, discarding all other branch metrics for that node. This is the crucial step of the Viterbi algorithm. The remaining branch for the node is known as the *survivor path.*

Repeat these steps for the rest of the nodes of each state. This leaves the survivors as indicated in figure 1.9.

This process must be repeated for steps $j = 4, 5, \ldots$ etc. At $j = 6$, the resulting trellis is indicated in figure(1.10) would remain.

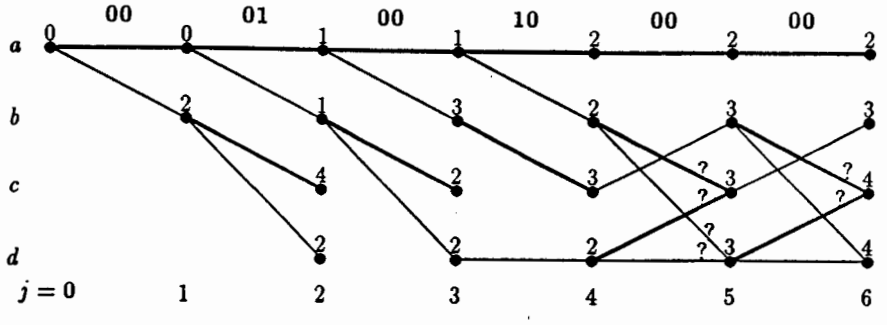


Figure 1.10: *Trellis at end of Received Sequence*

At stage $j = 6$, the node a_6 clearly has the lowest metric. So we can backtrack down the trellis and conclude that an all zero sequence was actually transmitted.

The trellis diagram of figure 1.10 shows a potential problem with the Viterbi algorithm. What if the branch metrics have the same value? (As is was the case at nodes c_5 and d_5, c_6 and d_6.) The easiest solution to this problem is to flip a fair coin and choose a branch randomly. Alternatively, a fixed decision rule can be applied. This however, must be done with caution since applying a fixed decision rule *may* introduce a statistical bias. This would depend upon the exact nature of the encoder.

The Viterbi algorithm as applied in this example is exactly the same as would be applied to trellis codes except for that the 'distance' is calculated differently. The Viterbi algorithm as applied to trellis codes will use the Euclidian distance between the measured received signal and the constellation point that would have been transmitted for that branch. The result for the distance is a positive real number rather than a positive integer obtained when using Hamming distance. This significantly reduces the likelihood of nodes having identical path metrics. (The magnitude of the reduction is a function of how finely the received signal is quantised and how much precision is used by the arithmetic.)

The truncated Viterbi algorithm is a minor modification to the Viterbi algorithm. For example, assume the trellis will be truncated at a length of 5. At $j = 5$, after the survivors have been computed, a symbol needs to be removed from the trellis. To do this, choose the node with the smallest metric at $j = 5$ and backtrack through the trellis to the symbol at $j - 5$. Output this symbol from the decoder. Discard the trellis at $j - 5$.

This process would be continued until all the received symbols have been processed. The remaining symbols in the trellis would then be output from the decoder by selecting and backtracking the path with the lowest metric.

1.4 Ungerboeck Representation

The Ungerboeck representation of TCM systems model the memory part of the encoder as a convolutional encoder. The author found this method better for obtaining an initial understanding of trellis codes over more analytical approaches. More analytical representations have been developed such as in [1].

If a_i, representing a source symbol at time i can take on 2^k values, it may be represented as a set of k binary digits $b_i^{(1)}, b_i^{(2)}, \ldots b_i^{(k)}$ which are simultaneously presented to the encoder. Let c_i represent the encoder output at time i. In general c_i will depend on the previous v_j bits (where $v_j \geq 0$) of the j^{th} binary input stream, $j = 1 \ldots k$.

$$c_i = \mathrm{f}(b_i^{(1)}, b_{i-1}^{(2)}, \ldots b_{i-v_j}^{(1)}, \ldots b_i^{(k)}, \ldots b_{i-v_j}^{(k)})$$

From this model, we can represent the trellis encoder as being composed of two parts :

1. A *binary convolutional encoder* with k input bits $b_i^{(1)}, \ldots b_i^{(k)}$ and n output bits $c_i^{(1)}, \ldots c_i^{(n)}$ where $n > k$. The sum

$$v = \sum_{j=1}^{k} v_j$$

is the memory of the encoder and determines the number of states of the encoder (which is 2^v).

2. A *modulator part* which associates the binary n–tuple c_i at the output of the encoder to a signal element x_i. The modulator is memoryless.

The *constraint length* of the encoder is defined as $v + 1$.

1.5 Ungerboeck Codes

Ungerboeck codes have the following properties :

- $M = M'$; that is M signals are used to transmit $\log_2 M'$ bits per signal.
- The convolutional code used is linear.

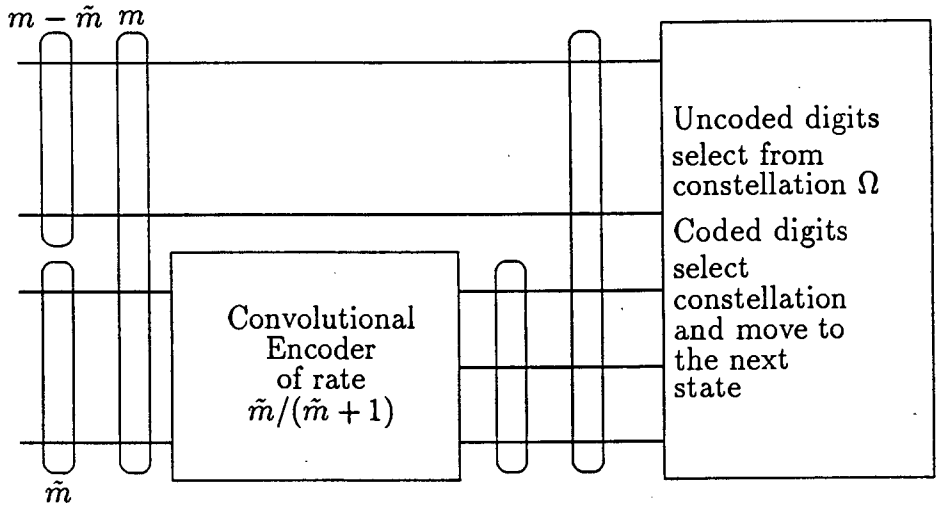


Figure 1.11: *Graphical representation of an Ungerboeck code*

Ungerboeck codes can be represented graphically as in figure (1.11). $\tilde{m}$ bits feed the $\tilde{m}/(\tilde{m}+1)$ rate convolutional encoder. The $\tilde{m}+1$ bits select a signal from the M signals defined in the constellation signal set Ω. The output of the encoder selects a subconstellation (or partition) from which the signal x_i is selected. The $m-\tilde{m}$ uncoded bits select one of the $2^{m-\tilde{m}}$ signal elements from the partition. These bits constitute parallel transitions, and according to the Ungerboeck Rules described earlier, must be selected from the same partition.

Chapter 2

Rotationally Invariant Codes

This chapter represents a synthesis of descriptions of rotationally invariant codes (RIC's) from various sources [11][12][1]. It was found by the author that all the references poorly described (or did not describe) some important aspect of rotationally invariant codes which is crucial to the full understanding of such codes. The author hopes that the description presented here addresses these shortcomings. The understanding of RIC's requires a thorough understanding of how basic TCM is applied in practice. The explanations provided in this chapter supplement the description of chapter 1, and provide a worked example.

Wei presented basic rules for designing rotationally invariant codes capable of 180° rotational invariance and 90° rotational invariance in two companion papers in 1984 [11][12]. A simplified presentation was given by Z. Chu and A. Clark in 1987.[13]

In this chapter we will cover the basic requirements for rotationally invariant codes (RIC) as applied to Ungerboeck Codes. The ideas explained here are implemented in the V.32 recommendations. In fact one of the codes presents by Wei in his 1984 paper[12] forms the basis of the V.32 recommendation for trellis codes.

Digital transmission systems often use modulation systems with suppressed carriers. For modems[1] various combinations of amplitude and phase modulations are often used.[2] The general format for such a signal is given by

$$s(t) = A(t)\cos(\omega_c t + \theta_m(t) + \phi) \tag{2.1}$$

ω_c is the carrier frequency of the signal. $\theta_m(t)$ is the phase change, and $A(t)$ is an amplitude change, due to the modulation of the signal. ϕ is some constant.

At the receiver, the exact value of the parameters of equation(2.1) are not known and have to be estimated from the received signal. The demodulator circuit applies some form of carrier recovery operation to the signal in order to provide an estimate of the carrier frequency $\hat{\omega}_c$. The recovered carrier normally has some phase uncertainty ϕ_u where ϕ_u is some integer fraction of 360°. The value of ϕ_u remains constant as long as the carrier recover circuit remains in lock.[3] Typical values of ϕ_u are 90° or 180° for practical circuit implementations. Such circuits are said to have a phase ambiguity of 90° and 180° respectively.

Consider a signal representing a point in a signal constellation. Say the demodulator measured the signal to have a phase of say 60°. A receiver with a phase ambiguity of 90° does not know if the signal is actually at 60° or at $60° + n90°$ where $n \in \{1, 2, 3\}$. However, if the next received signal element is measured at say 130°, the receiver will know the phase has advanced from the previous signal element by $130° - 60° = 70°$. The receiver will always be able to determine how much the latest signal element has moved in phase from the previous signal element, provided the carrier recovery circuit does not lose track or 'lock' of the carrier.

[1] MOdulator and DEModulator circuits designed for general switched telephone networks or leased circuits

[2] Frequency modulation is also used in early modems, but have fallen out of favour because of their bandwidth requirements.

[3] The term 'lock' is applied in terms of phase locked loops which are normally used in carrier recovery circuits.

There still remains the problem of determining the initial phase ambiguity. This can be achieved by sending a known string of data at the beginning of any message. This will allow the phase error of the local carrier estimate to be measured. The data can then be recovered by subtracting this error phase from the measured phase of each signal element. There are several disadvantages to this system of synchronisation :

- The initial message must be re-transmitted every time a new connection is made between the send and receive circuits.
- If carrier lock is momentarily lost, all subsequent data in the message will be incorrect if the new phase error is different from that of the previous carrier lock.

A better solution is to design the modulator / demodulator circuits to be insensitive to phase ambiguities that exist in the carrier recovery circuits. This can be achieved by applying differential encoding to the data before it is applied to the modulator circuits and corresponding reverse operation at the output of the demodulator circuit. Differential encoding causes the data to generate a signal change that corresponds to a value change of the original data. The demodulator can correctly measure the changes of the signal elements as described earlier. The original data is recovered by applying the inverse operation of the differential encoder to the receiver's output.

This method still requires on extra signal element to be transmitted before the phase ambiguity is removed from the data. A major advantage over the previous method is that in the event of loss of carrier lock, the differential encoder/decoder circuits automatically recover after carrier lock is regained and one signal element has been received.

Trellis codes do not allow such a simple solution as described above. The convolutional coding and the resulting signal elements within the signal constellation are inseparably combined. At first it was thought to be very difficult if not impossible to develop trellis codes insensitive to phase ambiguities. In fact, realising that the constellation and the convolutional code are indeed inseparably combined is the first step in developing *rotationally invariant codes.*

As eloquently stated in the book by Biglieri et al[1]

> *If the inputs, encoder states, and outputs are invariant under a* 90° *rotation, the code must be so invariant, because every entity used to define the code is so-invariant.*

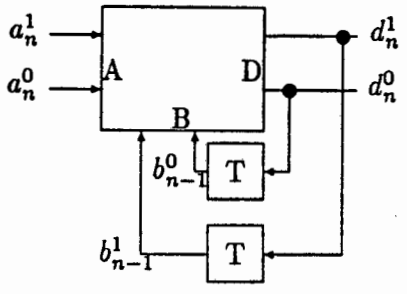
In the following sections the descriptive approach given by Biglieri[1] will be used to describe a simple trellis code scheme. It illustrates the points that need to be understood in order to understand the requirements for rotational invariant trellis codes.

2.1 Differential Encoding of the Inputs for Rotationally Invariant Codes

90° rotational invariance requires that one of 4 possible rotations must be catered for. This may achieved by applying a modulus 4 operation to the sum of 2 least significant binary bits of the input symbol and the previous (or delayed) least significant 2 bits from the differential encoder. This is shown in figure (2.1).

In practice, the modulo-4 adder can be implemented by using a 16 entry ROM[4] or combinational logic. The differential encoder operates by computing the sum of the previous output (B) and moving that value forward by the number of positions

[4]Read Only Memory



$$D = (A + B) \bmod 4$$
$$b_n^1 = d_{n-1}^1$$
$$b_n^0 = d_{n-1}^0$$

Figure 2.1: *Differential encoder*

at the input (A). For example, an input binary pair 10_2 will rotate the previous input by 2 positions. The rotation corresponds to modulo-4 addition in binary rotation and to the selection of one of the four quadrants in the constellation representation. This operation is reversible since the modulo-4 addition is reversible. If $D = (A + B) \bmod 4$ then $A = (D + -A) \bmod 4$ where $-A$ is defined as the 2's complement of A as indicated below :

A	$-A$
00	00
01	11
10	10
11	01

2.2 Code Requirements for Rotationally Invariant Codes

In the following discussions relating to 90° rotational invariance, the binary numbers relate to phase as defined by this table.

00	0
01	$\pi/2$
10	π
11	$3\pi/2$

Consider an encoder shown in figure (2.2).

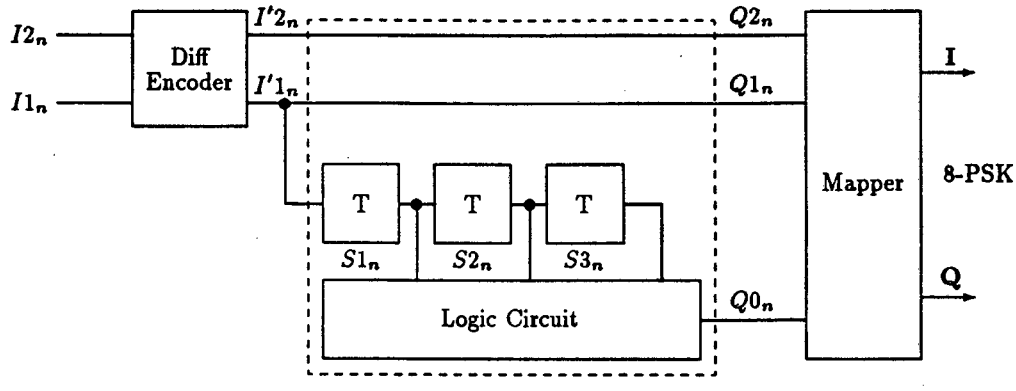


Figure 2.2: *Diagram of encoder example*

In this example, the state of the encoder is determined only by the history of the bit $I'1_n$. The state transition diagram for this encoder is represented by figure(2.3).

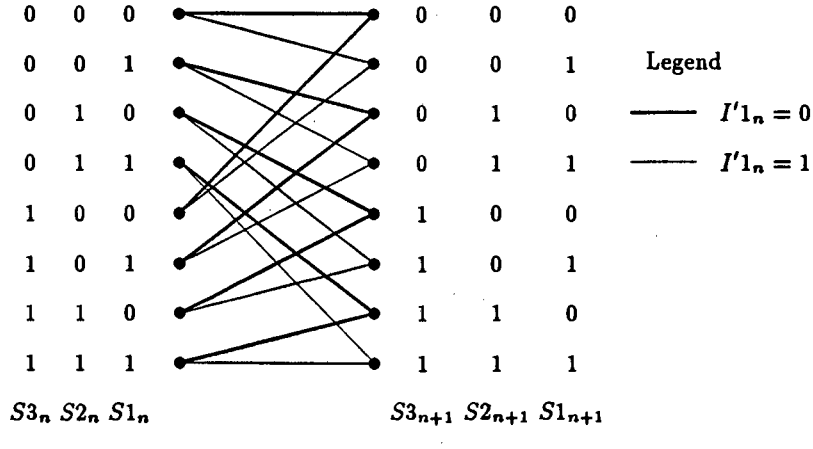


Figure 2.3: *State Transition Diagram for encoder of example*

The first step in developing a rotationally invariant code is to establish the effect of phase rotations of the input data on the state of the encoder.

In the example of figure(2.2) the state is determined by the previous three values of the bit $I'1_n$. The vector $(I'2_n, I'1_n)$ is altered by a phase rotation as defined by table (2.1). Notice that the resultant phase advanced vector is simply the original vector added (modulo-4) to the binary representation of the phase advance. It can

Input		Phase Advance			
$I'2_n$	$I'1_n$	0	$\pi/2$	π	$3\pi/2$
0	0	00	01	10	11
0	1	01	10	11	00
1	0	10	11	00	01
1	1	11	00	01	10

Table 2.1: *Table of phase rotated vectors $(I'2_n, I'1_n)$ for various values of phase rotation.*

be seen from table (2.1) that $I'1_n$ is unchanged for phase rotations of 0 and π but is complemented for phase rotations of $\pi/2$ and $3\pi/2$.

A point worth emphasising here is that once the receiver has established lock onto the carrier, the phase error of the receiver will be at some *fixed* multiple of the phase ambiguity. Thus all measurements of the received signal will have the *same* phase error.

Consider a sequence 01011 ... for $I'1_n$ applied to the state transition diagram of figure(2.3). A phase rotation of $\pi/2$ for the input data would result in the complement sequence 10100 .... The encoder will be in some state defined by the previous 3 values of the input $I'1$. For a different starting state, but the same sequence of

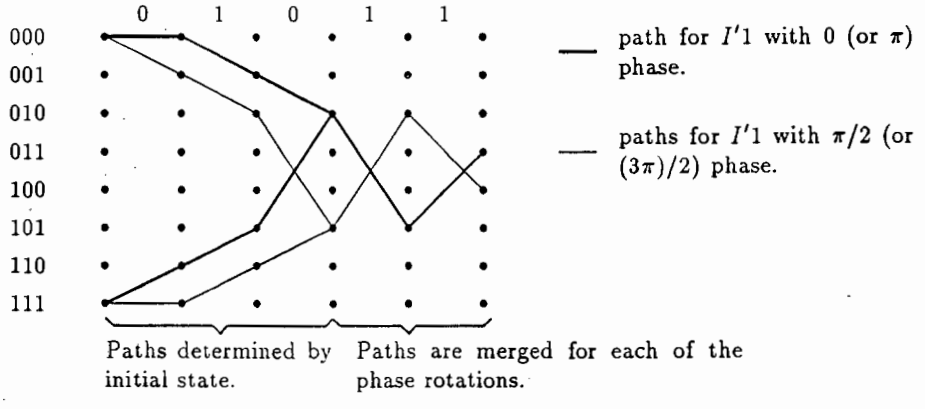


Figure 2.4: *Trellis diagram showing state transitions for example sequence described in text*

data, the trellis path may be different for the l bits that are used to define the state of the encoder. For this example, $l = 3$. After not more than l bits, the paths for

the same input data sequence *will* merge, as illustrated in figure(2.4). Figure(2.4) also shows the path for the $\pi/2$ phase shifted version of the data sequence for $I'1$. This sequence also converges in at most l bits, but the final path converged to is different to that of the data with no phase shift.

Two important concepts are illustrated by figure(2.4) :

1. The trellis paths formed for input data that is phase shifted follows some unique path that is generally different from the path trellis formed by the original (unshifted) input data.

2. The different trellis paths for data sequences with various phase offsets can only be established after the phase difference has been present at the encoder input for l bits.

Using the information established thus far, we can now generate a table showing the effect of phase rotations on the state vector of the encoder. The procedure is as follows :

For each of the states of the encoder :

1. Establish an input sequence of l bits required to reach the state.

2. For each of the required phase rotations :

   (a) Advance the input sequence by the phase rotation.

   (b) Apply the resultant sequence to the encoder and tabulate the resultant state of the encoder.

For example, consider state 0. The sequence $I'1 = 000$ will place the encoder in state 0. ($I'2$ does not affect the state vector in this example, and is excluded.) If the sequence is advanced by $\pi/2$, the new input sequence is $I'1 = 111$. This results in a state vector of 111_2. Add this value to the table for the results.

Completing this process for all unique multiples of $\pi/2$ and all states gives table(2.2). From this table we can find the equivalent code sequences for the encoder

Target State	Phase Rotation (rad)			
	0	$\pi/2$	π	$3\pi/2$
000	000	111	000	111
001	001	110	001	110
010	010	101	010	101
011	011	100	011	100
100	100	011	100	011
101	101	010	101	010
110	110	001	110	001
111	111	000	111	000

Table 2.2: *Table of phase rotated state vectors for the encoder example*

when phase rotated data is applied. For example, consider the transition from state 010_2 to 101_2. If there is a $\pi/2$ phase rotation on the input data, the new code sequence would be 101_2 to 010_2. These values are obtained by looking up the entries under the $\pi/2$ column in table(2.2) for the states 010_2 and 101_2 respectively. The results correspond with the trellis diagram illustration of figure(2.4).

2.3 Mapping requirements for Rotationally Invariant Codes

The final requirement for generating a rotationally invariant code is to ensure that the encoder output is correctly mapped to the signal constellation for the rotationally invariant code. The requirements of RIC's add further restrictions to the mapping procedure set by the Ungerboeck Rules. For rotational invariance the mapping must meet the following requirements :

1. Let U be the set of signals associated with the transition from state $S1$ to state $S2$. let ϕ be a phase rotation that is some integer multiple of the phase ambiguity. (i.e. $\phi = \pi/2, \pi$ or $3\pi/2$ for 90° RIC). Let $S1_r$ and $S2_r$ be the equivalent states resulting from the phase rotation ϕ of the states $S1$ and

$S2$ respectively. A rotationally invariant code requires that the set of signal elements associated with the state transitions from $S1_r$ to $S2_r$, be the set of signal elements obtained by rotating the signal elements U by ϕ. This rule is known as *the rule of equal rotation.*

2. The signal elements that are ϕ radians apart must be associated with the same encoder output bit combinations that do not affect the state of the encoder.

In some cases, these requirements for the mapping of RIC conflict with the Ungerboeck rules. If this occurs, some optimality of the code must be sacrificed to achieve rotational invariance. However, in general, it is possible to find another code that *does* allow all the Ungerboeck rules to be applied together with the requirements for RIC.

To illustrate the application of the Ungerboeck Rules and the mapping requirements for RIC, we will consider applying the mapping rules to an 8-PSK constellation for the encoder of figure(2.2).

First, note the encoder has one bit that does not affect the state of the encoder. This implies that there will be parallel transitions for each of the possible state transitions. The first step is to partition the 8-PSK constellation into 4 partitions, each have two signal elements (one for each of the possible parallel transitions). A suitable partitioning is shown in figure(2.5). Note the application of Ungerboeck's guidelines for set partitioning[10]. Starting at some arbitrary state in the state transition diagram of the encoder, assign some signal sets to that state. According to the Ungerboeck rule 2, signals leaving the same state must be assigned signal elements from the next higher sub-constellation. For this example, assign the sub-constellation $B0$ to state 0. ($B1$ would have been equally valid at this stage.) Assign the signal set (2,6) to the transition from state 0 to state 0. The first element of

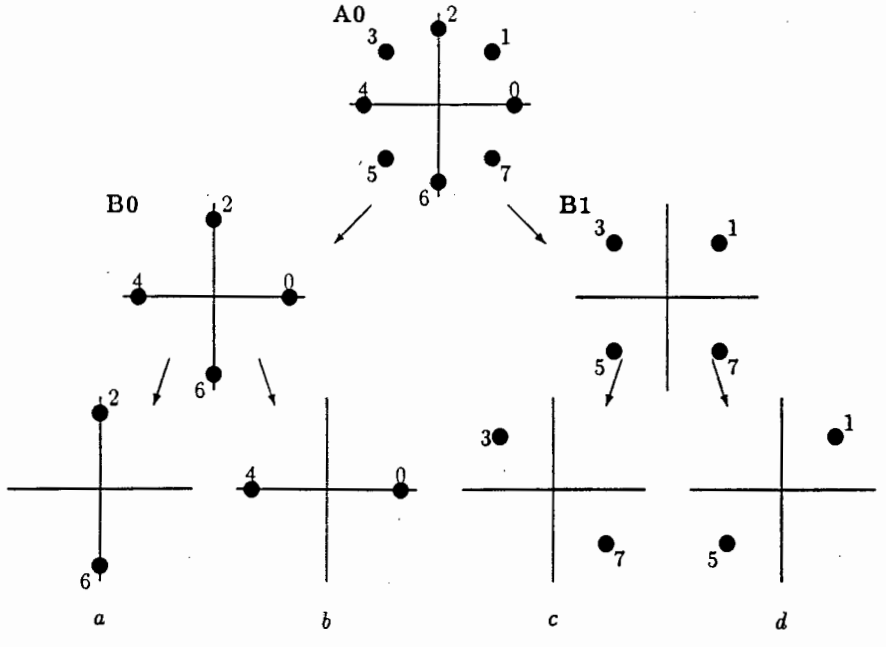


Figure 2.5: *Set partitioning for 8-PSK constellation*

the set is taken to be for $I'2 = 0$ and the second value for $I'1 = 1$. These signal alternatives constitute the parallel transitions for this code. After this assignment the signal set for transition from state 0 to state 1 is required to be b. Assign the set (0,4) to this transition. The assignments can be represented as in figure(2.6) by the ordered set of numbers to the left of the node for state 0. Likewise, all further signal mapping assignments discussed here have been added to figure(2.6). The associated input bits $I'2$ and $I'1$ are labelled above the corresponding signal assignments. In keeping with Ungerboeck's design rules 2 and 3, assign the signals elements of sub-constellation $B1$ to the state transitions originating from state 1. So far, note that state 1 is *not* one of the equivalent phase rotated states of state 0 (the destination states of the previously assigned state transitions).

Any states that are one of the phase rotated states from a previous transition assignment must *be assigned signal elements according to the rules for rotationally invariant codes rather than the Ungerboeck rules.* Understanding this is essential to combining the Ungerboeck rules with the requirements for RIC's.

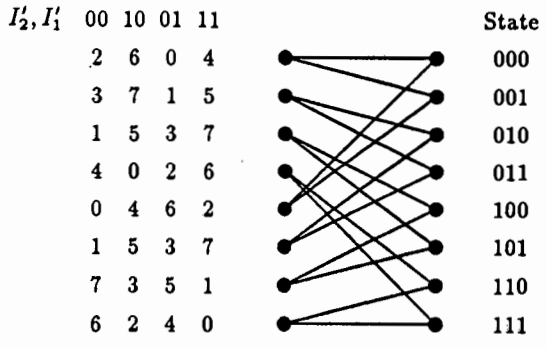


Figure 2.6: *State transition diagram with signal mapping assignments for 8-PSK constellation*

So far we have been able to use Ungerboeck's rules strictly. Now consider the transition 000 → 000. From table(2.2) the $\pi/2$ rotated equivalent transition is 111 → 111. The input for the 000 → 000 transition is 00_2 or 10_2. The input required for the rotated 111 → 111 transition is 01_2 or 11_2. Thus the inputs for the 111 → 111 transition are rotated by $\pi/2$. (01_2 is 00_2 advanced by $\pi/2$, and 11_2 is 10_2 advanced by $\pi/2$.) Thus the signal elements associated with the transition 111 → 111 must be the set (2,6) advanced by $\pi/2$, i.e. the set (4,0). See figure(2.5). This process must be repeated for all the transitions that have been assigned by using the Ungerboeck rules. This is easiest achieved by tabulating the results. This has been done in table(2.3) for the rest of the transitions of the currently assigned state transitions.

Assigned Transition	Input	Rotated Transition	Required Input	Phase rot of input	Assigned Mapping	Rotated Mapping
000 → 000	00	111 → 111	01	+90	2	4
	10		11	+90	6	0
000 → 001	01	111 → 110	00	-90	0	6
	11		10	-90	4	2
001 → 010	00	110 → 101	01	+90	3	5
	10		11	+90	7	1
001 → 011	01	110 → 100	00	-90	1	7
	11		10	-90	5	3

Table 2.3: *Table of phase rotated state vectors for the encoder example*

Further assignments are done by repeating the process described above of applying the Ungerboeck Rules for unassigned transitions and then applying the rules for RIC's to the resulting phase rotated transitions. The signal set (4,0) and (6,2) are assigned to state 100_2 to obey the Ungerboeck rule that transitions entering the same state must be assigned from the next higher partition. (The previous assignment set the requirement that all transitions *entering* state 000_2 should be from $B0$). After applying the rules for RIC's it is found that the state transition from state 011_2 violate the Ungerboeck rules. This results in a sub-optimal trellis code. It is however possible to find another 8 state code that is optimal[1].

The completed table of assignments is figure(2.6). The circuit required to generate the convolutional encoder can be derived from a truth table representation of the state transition diagram. For more information on the implementation of the convolutional encoder refer to [1, Chap 8.2].

Chapter 3

V.32 Specifications for a Trellis Code

This chapter summarises the relevant information about the trellis code as defined by the V.32 recommendation. The information presented here is a synthesis of the information contained in the V.32 recommendation and parts of Wei's paper[12], from which the V.32 recommendation is derived.

The CCITT[1] recommendation V.32[2] describes the requirements for the implementation of a 2–wire, duplex modem operating at data signal rates of up to 9600 bits/s for use on the general switched telephone network and on leased telephone type circuits.

The general characteristics of the V.32 specifications may be summarised as follows[2]:

- duplex mode operation on the GSTN[3].
- channel separation by echo cancellation techniques.
- QAM[4] for each channel, with synchronous line transmission at 2400 bauds.
- various signalling data rates

[1] CCITT - International Consultative Committee for Telegraphy and Telephony
[2] Taken from the V.32 recommendations
[3] General Switched Telephone Network
[4] Quadrature Amplitude modulation

  - 9600 bits/s synchronous
  - 4800 bits/s synchronous
  - 2400 bits/s synchronous

- two modes of operation for 9600 bits/s
  - One using 32 carrier states with trellis coding.
  - One using 16 carrier states (no coding implemented).
- exchange rate sequences during startup to establish the data rate, coding and any other special facilities.
- an asynchronous mode of operation.

As the above list indicates, the V.32 recommendation covers more than defining a trellis code. This document covers only the trellis coding aspect of the V.32 recommendation. Thus only the information pertaining to the modulation using 32 carrier states with trellis coding and 16 carrier states without trellis coding will be described. The latter uncoded modulation will be described to provide a comparative reference for the trellis coded modulation system in terms of error performance.

The general format of the modulator for 9600 bits/s operation is shown in figure(3.1). The scrambler circuit modifies the input data stream to help "randomize" the data to the modulator. This helps prevent long data sequences being generated that do not use the full range of signal elements. This is required to help ensure the average value of the signal transmitted is maintained to the true average of the signal constellation. This is required to implement gain control (or equivalently *scaling*) in the receiver. Simulations generally use randomly generated data, thus removing the need for the data scrambler. In addition, it may be desirable to

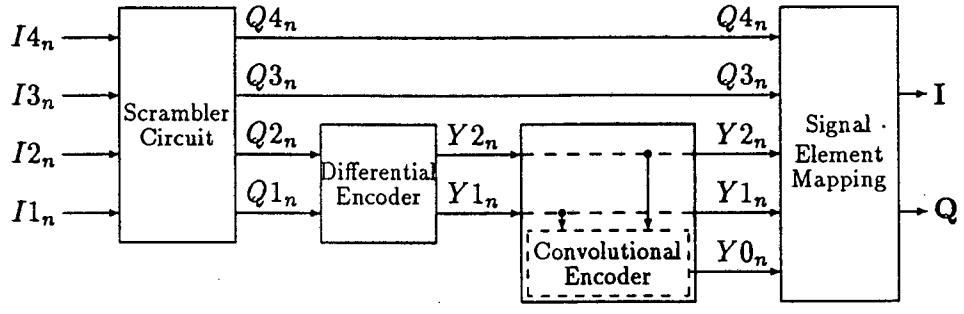


Figure 3.1: *Block diagram of V.32 9600 bits/s encoder*

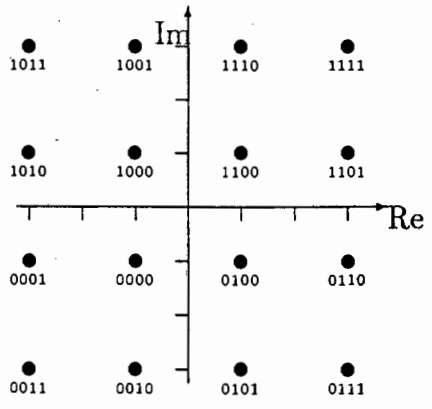
investigate how a modem circuit behaves when constant data is transmitted. The scrambler will also have some effect on the average length of error events. This will modify the performance of the overall system. For these reasons the scrambler (and corresponding de-scrambler) circuits are neither considered nor implemented in the simulation.

3.1 9600 BITS/S NON-REDUNDANT CODING

The differential encoder is defined in column A of table(3.1). There is no convolutional encoder, and the extra bit $Y0_n$ does not exist. The values of $Y2_n$ and $Y1_n$ are passed directly from the differential encoder to the signal mapper. The QAM signal mapping is defined by figure(3.2) and in column A of table(3.2).

3.2 9600 BITS/S WITH TRELLIS CODING

The trellis coded option of the differential encoder uses a different differential encoder from the previous one. The differential encoder is defined by column B of table(3.1). The convolutional encoder is an 8 state encoder that generates one redundant bit $Y0_n$. The values of $Y2_n$ and $Y1_n$ are not modified by the encoder. The details of the convolutional encoder are shown in figure(3.3). The QAM mapping is given in figure(3.4) and in column B of table(3.2). Note the use of a 32-Cross (32-CR)



The binary numbers denote $Y1_n$ $Y2_n$ $Q3_n$ $Q4_n$

Figure 3.2: *Signal mapping for nonredundant coding for 9600 bits/s*

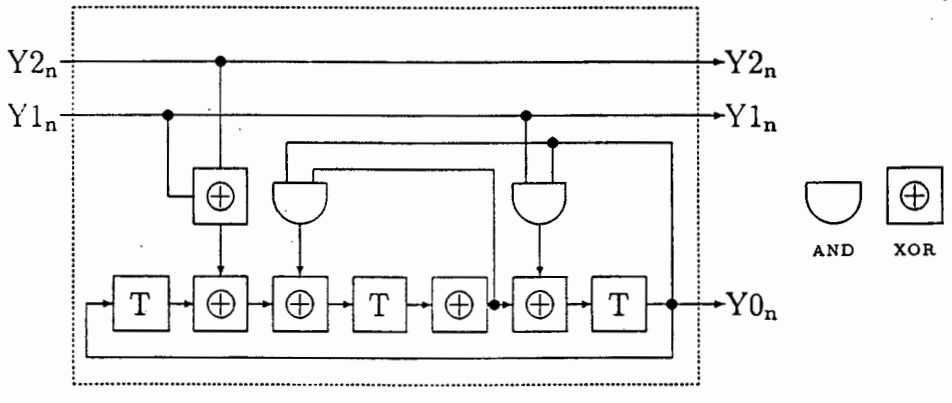
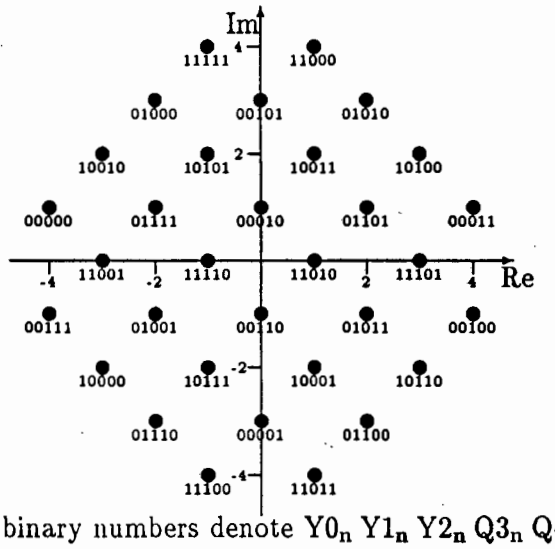


Figure 3.3: *Convolutional Encoder for V.32 9600 bits/s trellis coded modulation*

signal constellation. This mapping has desirable properties over a 32-Square (32-SQ) mapping :

1. Given the same average signal power, the minimum Euclidian distance among the signal elements of the 32-CR is 1.02 times that of that of the 32-SQ signal elements.

2. The ratio of peak to average power is 1.7 for the 32-CR signal space and 2.33 for the 32-SQ signal space.



The binary numbers denote $Y0_n$ $Y1_n$ $Y2_n$ $Q3_n$ $Q4_n$

Figure 3.4: *Signal mapping for 9600 bits/s with Trellis Coding*

3. The ratio of the average to minimum power of the 32-CR is 10 while this ratio is 21 for the 32-SQ signal space.

The state transition diagram is not provided under the V.32 recommendation. It can be derived from the logic circuit that implements the convolutional encoder. This exercise was done by the author before he discovered the state transition diagram in Wei's 1984 paper[12]. The state transition diagram is given in figure(3.5).

The rest of the V.32 recommendation deals with the interchange circuits and with startup connection and training procedures for the modems. This is all beyond the scope of this document and will not be discussed further.

The details of the V.32 recommendation for trellis coded modulation follows the basic structure for trellis coded modulation presented in the Ungerboeck representation of Chapter 1. Note however, the V.32 recommendation is *not* an Ungerboeck code since the convolutional encoder described in the V.32 recommendation is not

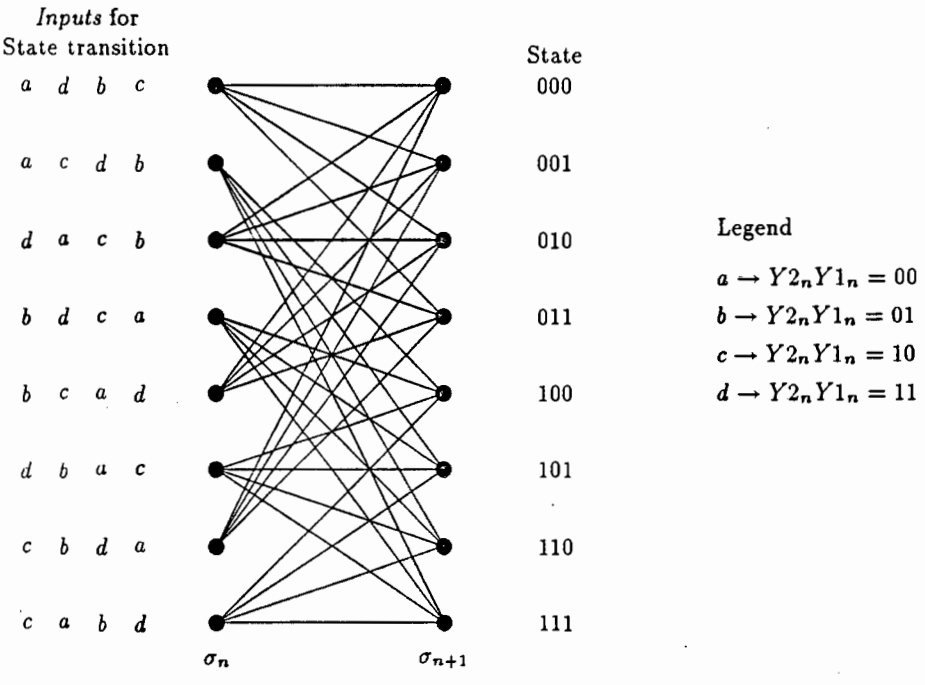


Figure 3.5: *State Transition Diagram for the V.32 Convolutional encoder*

linear[12].

				Nonredundant Coding		Trellis Coding	
Inputs		Previous Inputs		Column A		Column B	
$Q2_n$	$Q1_n$	$Y2_{n-1}$	$Y1_{n-1}$	$Y2_n$	$Y1_n$	$Y2_n$	$Y1_n$
0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	1	1	1	0
0	0	1	1	0	1	1	1
0	1	0	0	1	1	0	1
0	1	0	1	1	0	1	0
0	1	1	0	0	1	1	1
0	1	1	1	0	0	0	0
1	0	0	0	0	0	1	0
1	0	0	1	0	1	1	1
1	0	1	0	1	0	0	0
1	0	1	1	1	1	0	1
1	1	0	0	0	1	1	1
1	1	0	1	1	1	0	0
1	1	1	0	0	0	0	1
1	1	1	1	1	0	1	0

Table 3.1: *Differential Encoder tables for V.32*

Coded inputs					Nonredundant Coding		Trellis Coding	
					Column A		Column B	
$Y0_n$	$Y1_n$	$Y2_n$	$Q3_n$	$Q4_n$	Re	Im	Re	Im
0	0	0	0	0	-1	-1	-4	1
0	0	0	0	1	-3	-1	0	-3
0	0	0	1	0	-1	-3	0	1
0	0	0	1	1	-3	-3	4	1
0	0	1	0	0	1	-1	4	-1
0	0	1	0	1	1	-3	0	3
0	0	1	1	0	3	-1	0	-1
0	0	1	1	1	3	-3	-4	-1
0	1	0	0	0	-1	1	-2	3
0	1	0	0	1	-1	3	-2	-1
0	1	0	1	0	-3	1	2	3
0	1	0	1	1	-3	3	2	-1
0	1	1	0	0	1	1	2	-3
0	1	1	0	1	3	1	2	1
0	1	1	1	0	1	3	-2	-3
0	1	1	1	1	3	3	-2	1
1	0	0	0	0			-3	-2
1	0	0	0	1			1	-2
1	0	0	1	0			-3	2
1	0	0	1	1			1	2
1	0	1	0	0			3	2
1	0	1	0	1			-1	2
1	0	1	1	0			3	-2
1	0	1	1	1			-1	-2
1	1	0	0	0			1	4
1	1	0	0	1			-3	0
1	1	0	1	0			1	0
1	1	0	1	1			1	-4
1	1	1	0	0			-1	-4
1	1	1	0	1			3	0
1	1	1	1	0			-1	0
1	1	1	1	1			-1	4

Table 3.2: *Signal-state mappings for 9600 bits/s modulation (V.32)*

CHAPTER 4

IMPLEMENTATION OF SIMULATION

A simulation program is a tool used to investigate the behaviour of a system without physically implementing the system. This allows the user to characterise and evaluate the performance of a system before the system is practically realized. An important feature of a simulation is that the environment in which the system works can be precisely controlled by the user, under repeatable conditions.

A thorough understanding of TCM was required for the implementation of the simulation. Without it, there was the risk of developing software that could not be applied to a general implementation of TCM.

The following points were considered goals that must affect how the simulation is implemented :

1) The format chosen must allow the simulation to be considered a reusable tool.

2) The code developed must focus on the trellis coded modulation aspects of the V.32 recommendation.

3) The programs must be structured to enable easy porting to more powerful processing facilities.

The 1st point implies that the simulation must not be tied to the specific trellis code implementation of the V.32 recommendation. As a tool, the user should not need

to access the source code. Thus some form of user definable configuration would be required. The trellis coded modulation modules of the simulation are intended to provide a measure of the performance of the code under simulation. The modules are *not* intended as debugging tools for codes that do not work.

The 2nd point implies that the program(s) developed should limit the scope of processing to what is unique to trellis coded modulation systems and to processing that is required to produce the desired results. Large programs designed for providing a sophisticated user interface while actually doing very little in terms of Trellis Coded Modulation simulation should be avoided.

The 3rd point necessitates the code to be written in a language that is available on other processing platforms, and using data structures that are practical on most processing platforms.

4.1 Structuring of Software

The simulation specified by this document implements a specific function. As such it could be implemented as a large program that simulates the TCM system of the V.32 recommendation, and presents the results to the user. This program could not be considered a general tool for TCM systems. It would however have the potential for being an educational tool for demonstrating the workings of an existing trellis coded modulation system.

The requirement for a general tool makes it is necessary to avoid a single monolithic program. The penalty paid for this is performance. In general, a greater performance in terms of execution speed is achievable by dedicating a program to a specific task.

The alternative is to develop a series of program modules that exchange data via some protocol. This is a *modular structure.* Each module has a specific function and accepts data in a pre-determined format, generally in a file format. By cascading modules together, one can simulate the overall functionality required. Thus the overall simulation is comprised of a series of modules that each perform some processing function. The simulation results are achieved by the accumulated processing of each module. For example, consider a digital communication system. There is a data source, an encoder/modulator, the channel, the demodulator/decoder and finally a data sink. This is illustrated in figure(4.1). This approach has distinct

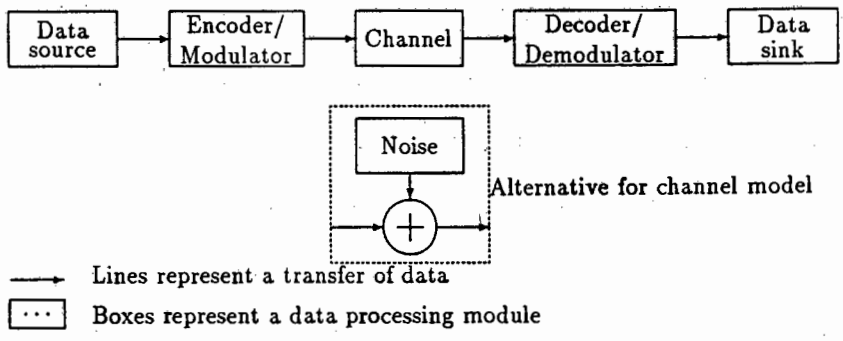


Figure 4.1: *Example batch processing to implement a communication system*

advantages :

- The structure is flexible. The function of the overall simulation can be modified by changing the functions of some of the modules. For example, a simple resistor-capacitor filter module can be replaced by a band pass filter.

- The system is expandable. Added functionality can be added to the system by creating a new module that implements the new function. For example, implement a non-linear median filter to remove shot noise.

- The system is reusable. Any module can be re-used for a different processing task by applying different processing to the data that is applied to the module.

The main requirements of the software implementation will dictate if the above restrictions are acceptable. More on this later.

4.1.1 Data Redirection and Piping

A facility provided by many operating systems is the ability to *pipe* or *redirect* files.

Redirection is the ability for the operating system to substitute a file for the normal input device of an application program, and to substitute some other file for the normal output device.

Piping is the routing output data from the standard output device of one application program to the standard input device of another application program without the user needing to generate intermediate files. Piping facilities are provided by the UNIX and MSDOS operating systems amongst others.

Piping allows the processing from one program to be passed directly to the next program without having to worry about generating and deleting files during and after the process. (The operating system automatically generates a data store for the data transfer and will recover the space upon completion of the command. This is normally implemented by the operating system generating temporary files.)

The initial input can be supplied by redirecting an existing input file to the application program using the appropriate redirection command. Similarly, the final results can be assigned to a permanent file by redirecting the output of the final application to a file.

To illustrate this consider the following examples for MSDOS based computers :

```
dir | sort | more
```

Alternatively, to create a sorted directory file `sdirectory.txt`, use

```
dir > directory.txt
sort < directory.txt > sdirectory.txt
```

The first example uses piping to route the output of the `dir` command to the `sort` program. The output from the `sort` program is in turn piped to the input of the `more` program to be displayed on the display device of the computer. The `|` character is the piping command for MSDOS. Note that there are no files remaining after the commands have been completed.

The second example generates an unsorted file `directory.txt` and a sorted file `sdirectory.txt` each containing a list of the files on the computers disk at the time. The first line generates the directory listing and *redirects* the resulting output to the file `directory.txt`. The second line sorts the file be taking the redirected input from the file `directory.txt` and places the results in the file `sdirectory.txt` by redirecting the output to that file. The `<` is the 'redirect input' command while the `>` is the 'redirect output' command.

These examples are provided only to illustrate the principle of program data redirection and piping.

4.1.2 Data Master - a signal processing tool

PC Data Master[3] is a software package designed for batch orientated digital signal processing(DSP). The functions provided by the package include waveform generation, noise generation, mathematical operations (addition, multiplication and others), Fast Fourier Transforms and an application for graphically displaying data files.

[3]Software ©Durham Technical Images, site licensed to UCT

A feature of the PC Data Master package is a graphical user shell from which to run the various applications. This *Data Master shell* is required for the applications that produce graphical output. The main feature of this shell is that it provides its own form of piping. A key advantage of the Data Master shell over MSDOS is the Data Master shell generates temporary store in main memory for piped data and will only spill to disk if memory runs low. This significantly speeds up the piping of data from one application to another.

As part of the package, the PC Data Master system describes in detail the file format used by the applications for data interchange. The characteristics of this data interchange format are :

- ASCII file format. This is important if the applications are to be ported to other operating systems.
- Variable length files — up to 2^{32} entries.
- Multi-channel data representation. Complex numbers are represented as two channel signals.
- Fields for sample interval, sample units, channel units and variable length comment fields for any other application.

The data in the interchange files is conceptually stored in channels. This is in line with the PC Data Master being intended as a DSP package. The data length is considered to be the length of each channel. For example a two channel signal with ten entries would have $2 \times 10 = 20$ numerical entries.

Refer to Appendix B for detailed information on the file interchange format.

4.2 Implementation Environment Decisions

The easy access and availability of MSDOS based personal computers encouraged the author to plan the development of all initial software on MSDOS based personal computers.

Based on the requirements of the simulation and the various considerations described above, the following implementation decisions were taken :

1. The signal processing shell provided by Data Master will be the basis of the operating environment.

2. The developed programs will consist of standalone modules that implement some well defined processing function. The programs will be usable in the Data Master shell environment and under the operating system environment.

3. All modules developed would support data piping in a structure compatible with the both the MSDOS and UNIX operating systems.

4. The code would be developed in ANSI[4] compatible C code as far as is possible.

These decisions were felt to best support the goals of the overall simulation implementation.

- The Data Master provides sufficient processing ability to allow the author to concentrate his effort on the trellis coded modulation aspects of the overall simulation process. The facilities for basic manipulating of data are already provided by Data Master.

- No shortcomings were seen in the Data Master data interchange format for the batch processing environment that is to be used.

[4]American National Standards Institute

- Development of code in ANSI compatible C would allow the completed programs to be ported to UNIX workstations with minimal rewriting of code.

- The piping used in the operating environment is easily applied to the UNIX workstations. Thus the fundamental structure of the simulations would not need to change if the applications were ported to UNIX workstations.

- The C programming language is one of the more universal languages available on various computer systems. C is *always* supplied with a UNIX based operating system.

4.3 Signal Representation for Simulation

A signal processing simulation must represent the signals of the system in a format suitable for processing. Generally, an electrical signal is represented as a sequence of samples spaced equally in time. In order to represent a signal without a loss of information, *at least* two samples are required of the signal per period of the highest frequency component of that signal. This is a fundamental limit set by the Nyquist criterium[5]. In practice this figure must be increased by 50% to 200% to allow for the non-ideal processing algorithms that are practically realizable. This has the effect of requiring large data stores for the transfer of data from one module to another. The sampled channel representation needs processing achieved a representation on which the decoder can make a decision as to what the received symbol is. The processing required is specific to the modulation method applied to the system. But is all this necessary?

In discussions with Dr Robin Braun[6] it was decided that a sampled representation of the channel signal was not necessary. Instead, the signal can be represented as a

[5]See any textbook on digital signal processing for a description of the Nyquist criterium.
[6]Supervisor for the thesis project.

single complex number for each symbol on the channel that represents the sampled output of a suitable matched filter. The basis for this decision is discussed below :

The overall structure of the required simulation is shown in figure(4.2). The trellis

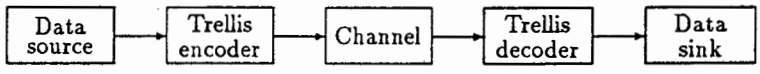


Figure 4.2: *Processing structure of V.32 TCM simulation*

encoder produces a QAM baseband signal which is placed on the channel. The trellis decoder converts the baseband signal to a data stream by applying the necessary decoding. Included in the trellis decoder must be a Viterbi decoder or some other maximum likelihood decoder. For the maximum likelihood decoder to operate, the in-phase (I) and quadrature (Q) signal components are required in order to establish the Euclidian distance between the measured signals and the proposed signal. (See chapter 1).

The optimum way of detecting a signal is to use a *matched filter*. Refer to a textbook on digital communications such as [9, 5] for further information. It is possible to implement a matched filter as an 'integrate and dump' type of circuit, otherwise known as a *correlation receiver*. Figure(4.3) shows a correlation receiver that is designed to optimally detect the pulse waveform $p(t)$. The period of $p(t)$ is T_b. The received signal $r(t)$ is the original transmitted signal that has been modified by the channel characteristic $h(t)$, and with added noise. The correlation receiver multiplies $r(t)$ with a locally generated version of the signal $p(t)$, and integrates the product over the symbol interval $0 \ldots T_b$[7]. The output of the integrator is sampled at time T_b to give a *single* value of interest. The output of the correlation receiver before and after the sampling instant is unimportant for the receiver. Whatever the

[7]In this notation, integrator value is reset at the start of each new symbol that is received. This resets the starting time of the integrator to zero for each new symbol that is received.

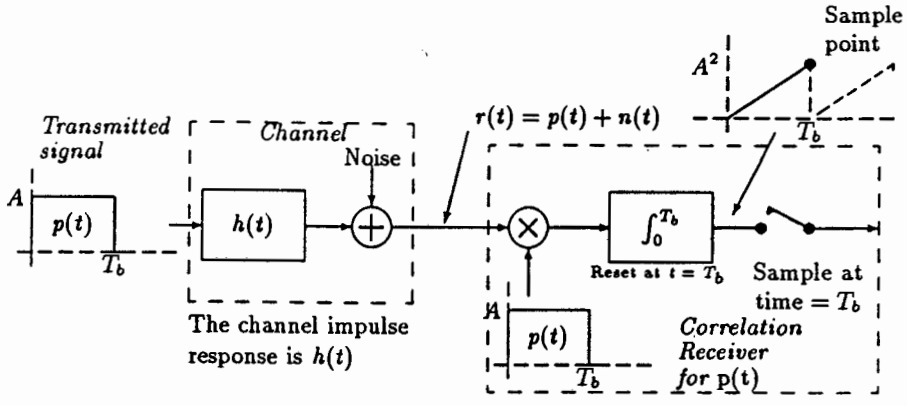


Figure 4.3: *Correlation Receiver for Pulse Waveform $p(t)$*

effect of the channel and the additive noise is, it will manifest itself on the output of the matched filter at the sampling instant. Thus if it is assumed that the receiver uses a matched filter, it is possible to represent the received signal completely as the output of the matched filter at the sampling instant of each channel symbol.

A QAM signal can be represented as the sum of two orthogonal signals

$$\begin{aligned} s_1(t) &= A_1(t)\cos(\omega_c t) \\ s_2(t) &= A_2(t)\sin(\omega_c t) \end{aligned}$$

A correlation receiver for a QAM signal requires a correlator for each of the orthogonal signals $s_1(t)$ and $s_2(t)$. The output of the two correlators form the I and Q numbers that represent the location of the received signal on a constellation diagram. This may be viewed as a complex number (as is done on the previous constellation diagrams).

The important point made by Dr Braun is that for a signal with additive Gaussian white noise, the signal to noise ratio of the output signal of the matched filter is the same as the signal to noise ratio of the input signal to the matched filter. This is a result of the fact that for Gaussian white noise with a uniform power spectral density $S(\omega) = N_0/2$, the mean square noise output of the matched filter is given

by equation(4.1).[5]

$$\overline{n_0}^2(t) = E\frac{N_0}{2} \tag{4.1}$$

E is the energy of the waveform that the matched filter is designed to detect. Note that equation(4.1) is dependent only on the energy of the waveform and the power spectral density of the noise. If $f(t)$ is the waveform the matched filter is designed to detect, and $F(\omega)$ is the fourier transform of $f(t)$ then the output signal of the matched filter at the sampling instant T_b is given by equation(4.2)[9].

$$\begin{aligned} f_o(t_m) &= \frac{1}{2\pi}\int_{-\infty}^{\infty}|F(\omega)|^2 d\omega \\ &= E \end{aligned} \tag{4.2}$$

The output of the matched filter due to a matched signal being applied to it depends only upon the energy of that waveform. Hence the signal to noise ratio of the output signal of the matched filter (SNR_o) is given by

$$\begin{aligned} SNR_o &= \frac{f_o^2(t_m)}{\overline{n_0}^2(t_m)} \\ &= \frac{E^2}{E\frac{N_0}{2}} \\ &= \frac{2E}{N_0} \end{aligned} \tag{4.3}$$

Under the assumption that the signal bandwidth of the channel is limited to the bandwidth required for the signal, the signal to noise ratio of the input signal can be derived as follows :

Let P be the RMS[8] power of the modulated signal. T is the period of each symbol on the channel. The average energy of each symbol is given by $E = PT$. Thus the RMS power P can be written as

$$P = \frac{E}{T}$$

[8] *R*oot *M*ean *S*quare

The RMS noise power $\overline{n_0}^2(t)$ is given by equation(4.4).

$$\begin{aligned} \overline{n_0}^2(t) &= \frac{N_0}{2} \cdot \text{Bandwidth} \\ &= \frac{N_0}{2T} \end{aligned} \tag{4.4}$$

Hence the signal to noise ratio can be expressed as

$$SNR_i = \frac{2E}{N_0} \tag{4.5}$$

This is the same expression as the for the signal to noise power of the matched filter output (equation(4.3)).

For effects other than white noise, it is possible to model the effect at the output of the matched filter because it is a linear filter. For example, phase jitter in the synchronization circuit could be modelled by multiplying the complex representation of the I and Q values by a complex number with magnitude of one and a phase distribution of the phase jitter being modelled. Other effects could be modelled by analytically applying the function of the matched filter to the analytic representation of the baseband effect. In the event of this being too difficult or unsuitable, it would still be possible to implement a matched filter module and apply that module to the baseband signal to yield the required I and Q number representation. (A module would also be required to convert the I-Q representation of the encoder/modulator circuit to a time signal.)

Thus it was decided to use the I-Q number representation for the signal on the channel. This approach has other benefits too :

- The assumed use of a matched filter implies that the decoder is optimum. Thus the simulation deals with the performance of the code rather than the

demodulation techniques of the system. This is considered to be an advantage for this implementation.

- Without tying the decoder/demodulator to a specific modulation type, it becomes possible to design the decoder/demodulator to be a more general tool than would otherwise be possible.

- The quantity of data transferred between the modulation program and the demodulation program modules is substantially reduced. (The reduction would be less if the signal space had more than 2 dimensions).

4.4 The V.32 Simulation

With the structure of the simulation sorted out is was possible to start implementing the code required for the simulation. The modules and the logical structure of the modules are shown in figure(4.4). The functionality of the modules is defined as :

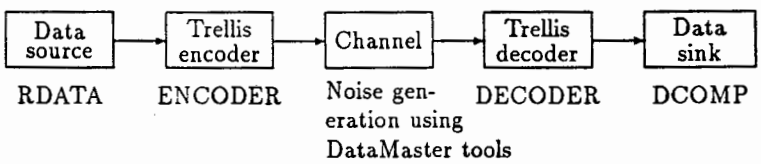


Figure 4.4: *Processing modules of V.32 TCM simulation*

RDATA This module is used to generate a sequence of random data for the subsequent ENCODER. The data is represented as a sequence of numbers in the range $0 \ldots N-1$ where N is the number of symbols transmitted by the modulation circuit. m bits of information are transmitted per symbol where $m = \log_2 N$. N can be specified by using a parameter to the module, allowing RDATA to be used for any M-ary modulation system. For the purposes of debugging and to be able to repeat simulations with the same data, the random number generator of RDATA must be a known one, with the ability to set the starting point of the random number sequence by supplying a parameter.

ENCODER The encoder module implements the modulator part of the V.32 specification. Data from the data source is processed into a representation suitable for the channel representation. As discussed earlier, for the case of the V.32 simulation, the data on the channel is represented as one complex number per source symbol. The encoder implements the encoder described in figure(3.1). The final version of the program uses a configuration file to define the modulator processing. The configuration file allows the modulator to be changed without requiring any changes to the source code.

Channel The performance of the code is measured by assuming the presence of additive white Gaussian noise in the data channel. This is achieved in the simulation by adding a Gaussian distributed waveform with zero mean and a variance required to give the required signal to noise ratio for the channel. The generation of the Gaussian distributed waveform and the addition of that waveform to the channel signal was initially done using existing Data Master tools. Later, a separate program 'ADDRAND' was developed to overcome a 'bug' in the Data Master tools. ADDRAND reads a Datamaster format input file and adds noise to the data and writes the resultant data to the output file. In addition, ADDRAND can vary the phase of a complex (2 channel) waveform with a Gaussian distribution.

DECODER This module implements all the functions necessary to decode the channel signal, *assuming the signal comes from suitable matched filters.* This in turn implies the matched filter has the suitable carrier recovery or symbol synchronization required for the optimum detection of the channel signal. The truncated form of the *Viterbi algorithm* is used to implement maximum likelihood decoding. A suitable differential decoder is applied to the Viterbi decoder. The output format of the data is the best estimate of the original

data symbols that the decoder can make. The final version of the program uses a configuration file to define the encoder used to generate the input data. Thus the decoder module automatically adapts itself to the signal generated by the encoder module (provided of course the *same* configuration file is used).

DCOMP [9] This is the data sink of the simulation. This module is used to compare the results of the DECODER module with the original data and display a summary of the data and the differences between the original data and the decoded data (including the bit error rate). The results from this program are used in the table of results.

4.4.1 Trial Simulation

A trial simulation of a simple modulator/demodulator circuit was implemented to evaluate the entire environment before the main development of code was attempted. The modulation scheme used for the simple implementation was binary phase-shift keying (BPSK). The structure of figure(4.4) was used, with the substitution of the encoder and decoder modules with modules that implement BPSK modulation and demodulation.

The trial simulation used the sampled matched filter signal representation for the channel as was proposed for the V.32 simulation. The trial simulation was intended to identify problems with the sampled matched filter signal representation before implementing the more complicated V.32 simulation.

The trial simulation consisted of four modules :

RDATA Generation of random data symbols for the modulator. The range of the data symbols and the starting value for the random number generator are optional user parameters for the module.

[9] DCOMP, by J.Schoonees, is an enhanced version of the author's original RCOMP.

BPSKMOD A simple module that maps the input symbols to an output sequence of symbols that represent the position on an I-Q diagram of the signal symbol.

BPKDEMOD This module converts the input sequence of I-Q complex numbers to a sequence of data symbols. The output should correspond to the input of the BPSKMOD module.

RCOMP A module to compare two sets of data symbols and present a summary of the statistics of the two files, including number of errors (differences).

Noise was added to the 'channel' by using the existing modules of Data Master. The simulation was run for signal to noise ratios ranging from 2dB to 10dB. The results show very little deviation from the expected (theoretical) results. Refer to Appendix A for details on the theoretical results, a table of the simulation results and the noise calculations.

The successful results of the trial simulation indicates the feasibility of using the matched filter signal representation for the channel. The Data Master environment proved easy to use, and the application modules were easy to interface to existing Data Master modules.

4.4.2 V.32 Specific Simulation

Initially, the implementation of the modules to implement the encoder and decoder of figure(4.4) were written to work specifically the TCM system of the V.32 recommendation. The resulting program modules are *not* re-configurable without changing the source code. This was done intentionally in order to ascertain the software requirements of the algorithms necessary to implement generalised encoder and decoder modules.

V.32 encoder

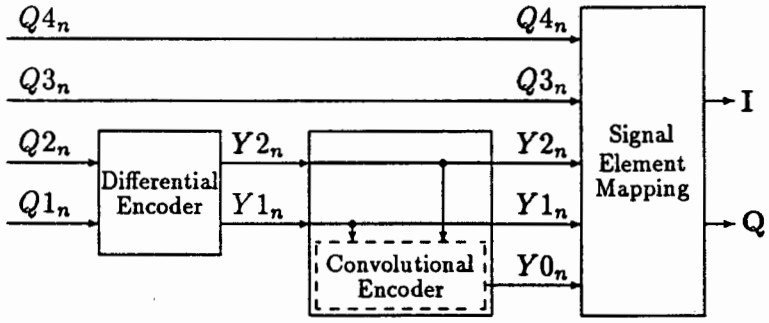


Figure 4.5: *Block Diagram of V.32 9600 bits/s encoder Structure as implemented in the Simulation*

Figure(4.5) is a block diagram of the functionality of the encoder module V32MOD. The internal processing of the module may be broken down into three sections as illustrated in the block diagram.

Differential Encoder The differential encoder is implemented by using a lookup table that contains the information of column B of table(3.1). The previous output is maintained in a static variable.[10] The lookup table was coded directly in the program source code. A lookup table allows the *method* of implementing the differential encoder to be applied to any mapping without changing the algorithm code. The differential encoder assumes an initial state of 00_2 for the previous output of the differential encoder.

Convolutional encoder This is also implemented by a lookup table. As before, the use of a lookup table allows the method of implementing the convolutional encoder to be applied to other convolutional encoders. The current state of the encoder combined with the inputs from the differential encoder are used to index into the table. The table contains the next state of the encoder. No direct output was generated from the encoder. The program used the knowledge

[10] A *static variable* in the C programming language maintains its value between calls to the function that defines the static variable.

that the output signals from the convolutional encoder are the output bits of the differential encoder and the least significant bit of the current state of the convolutional encoder. In this implementation, the table defining the convolutional encoder was *generated* by applying a program that simulated the logic function of figure(3.3).

Signal Mapping The output of the convolutional encoder and the remaining uncoded signal lines were used index into a table to generate a complex number representing the signal point on the constellation diagram of figure(3.4). The complex number is the output of the program.

The convolutional encoder by default assumes an initial state of all zeros. The user can change this default initial state by providing a suitable parameter to the module when it is executed.

All the data pertaining to the V.32 recommendation for trellis coded modulation is placed in tables in program memory. This is intentional for two reasons :

1. The use of lookup tables in a computing algorithm is often the fastest way of obtaining a computational goal. This is especially true of state machines such as a convolutional encoder.

2. When a lookup table is applied to a state machine such as a convolutional encoder, the contents of the table completely defines the convolutional encoder. The use of tables allows the encoder to be changed without changing the structure of the software.

The intention of developing a generalized encoder/decoder pair made the use of data tables a suitable means of implementation.

V.32 Decoder

The decoder module V32DEMOD is required to demodulate the matched filter signal representation of the channel data. In order to do so, some form of maximum likelihood decoding is required. The truncated Viterbi algorithm provides a practical method of implementing such a decoder. (See Chapter 1). The truncated Viterbi algorithm is a sub-optimal decoder, but can be made nearly optimum by choosing a sufficiently long truncation depth. [8, page 495] states that simulation has shown that truncation depths of $\delta > 5L$ have a negligible coding gain over a truncation depth of $\delta = 5L$. L is the constraint length of the applicable code.

Other algorithms were investigated for implementing the maximum likelihood decoding. *Sequential decoding* algorithms exist for sub-optimum maximum likelihood decoding[5]. These algorithms are less computational intensive than the truncated Viterbi algorithm but do not perform as well. They do have the advantage of being practically implementable on codes that have long constraint lengths (i.e. many states). The truncated Viterbi algorithm is practically limited to codes that have constraint lengths in the range 7 to 11 [5] because of the exponential dependence of the computational requirements with the constraint length of the code.

Since the simulation is intended to evaluate the performance of convolutional trellis codes, and not the decoding algorithms, the alternatives to the Viterbi algorithm were not considered further.

To facilitate the easy upgrade to a generalized decoder, all the information about the convolutional code required for the maximum likelihood decoder is derived during program execution from the tables used to implement the encoder module.

COMPUTING THE METRIC

This section deals with implementing the Viterbi algorithm described in Chapter 1. Some practical implementation considerations are expanded upon as a supplement to the information in Chapter 1.

In the V.32 implementation for a trellis code, there are two bits of information that have no effect on the state of the encoder. These result in four parallel transitions for each of the state transitions of the code. There are four valid transitions entering each of the valid states. This means that when a survivor transition is selected for a state, each of the four parallel transitions for each of the four valid state transitions must be checked against the received symbol to determine which has a minimum metric. This means for each of the eight states there are $4 \times 4 = 16$ possibilities that must be checked. This means the Euclidian distance between the received signal point must be checked against $16 \times 8 = 128$ valid constellation points for each symbol interval. Since there are only 32 valid constellation points, the distance between each valid constellation point and the received signal is computed four times (four being the number of parallel transitions). This point was noted as a potential point for optimization after the program module was working[11].

In addition, associated with each of the survivor paths, it is necessary to store information to say which of the four possible parallel transitions was responsible for the transition with the minimum metric.

The V.32 decoder assumes an initial encoder state of 0, which corresponds to the default starting state of the encoder. The assumed starting state of the encoder can be changed by providing the decoder module a suitable parameter when it is

[11] The function `Euclidian()` in the program modules V32DEMOD and DECODER *was* susequently optimized for the C compiler used, and the resulting program executed almost twice as fast.

executed.

Testing the Simulation

All initial testing of the software was done using the debugging environment provided by the C language compiler. The debugging aids allowed the operation of the software to be scrutinized in order to verify the intended operation.

The V32MOD program module was verified to operate correctly by comparing the program results for known input data steams against results generated manually. Three cases of randomly generated test data were used. The manually computed outputs were limited to about 32 values because of the difficulty in generating the code without human error. In all three cases, the program results were identical to the manually computed results. The author was satisfied that the test results showed the coding to be correct.

The V32DEMOD program module was confirmed to operate as expected by applying the data generated by the completed V32MOD program and to verify that the output results were as expected. Additional confidence in the maximum likelihood decoding was obtained by perturbing some of the input data to the program and observing the results. The program managed to correct several perturbations placed in the input file.

The consistency of the interaction between the encoder and decoder program modules provided the final confidence to the author to proceed with the implementation of the generalized encoder and decoder modules.

4.4.3 Generalized Encoder/Decoder

The software modules written to implement the trellis code of the V.32 recommendation gave insight to the requirements of a general decoder. Figure(4.6) shows

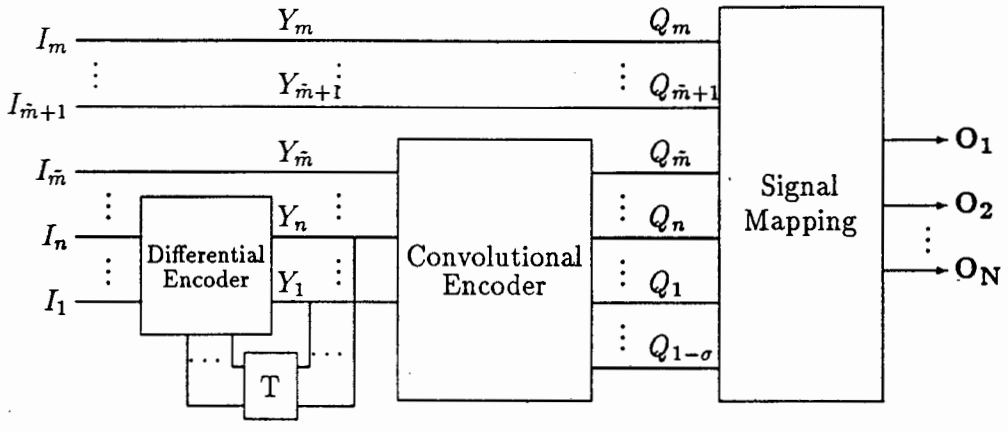


Figure 4.6: *Generalized Structure of a Trellis Coded Modulation System*

the basic structure of such a generalized encoder. There are m parallel input bits $I_m \ldots I_1$, which represent an input range of 2^m unique values. The n bits $I_n \ldots I_1$ are differentially encoded to give $I'_n \ldots I'_1$. The differentially encoded bits and the remaining bits $I_m \ldots I_{n+1}$ are combined to form the m bits $Y_m \ldots Y_1$. $\tilde{m}$ bits $Y_{\tilde{m}} \ldots Y_1$ are applied to the convolutional encoder. The convolutional encoder in turn has a number of memory elements, and produces $\tilde{m}+\sigma$ output bits $Q_{\tilde{m}} \ldots Q_{1-\sigma}$. The σ bits are the redundant bits used to map to the expanded signal space. The uncoded bits $Y_m \ldots Y_{\tilde{m}+1}$ are combined with the bits $Q_{\tilde{m}} \ldots Q_{1-\sigma}$ from the convolutional encoder to select a signal element from the expanded signal space. This signal mapping can be to one or more signal dimensions (indicated by $\mathbf{O_1} \ldots \mathbf{O_N}$ in figure(4.6).

The convolutional encoder is a 'black box' with $\tilde{m}$ binary inputs and $\tilde{m} + \sigma$ outputs. Within the black box is the memory of the encoder which defines the current state. The function is entirely defined by the mapping of the inputs to the outputs with the current state of the encoder. The next state of the encoder is defined by the current state of the encoder and the input bits. The function of the convolutional encoder can be completely defined by implementing a table that is indexed by the current input combined with the current state, and whose contents are the output bits and the next state of the encoder. This view of the convolutional

encoder corresponds with the model developed in Chapter 1.

The model used for the generalized convolutional encoder adds a further justification to the decision to implement the channel as a matched filter signal representation. The model of figure(4.6) specifies a variable number of output signal dimensions $O_1 \ldots O_N$. On the receiver side this corresponds to a bank of N matched filters (or equivalently, correlation receivers) to detect each signal dimension. The implementation of the modulation of the N-dimensional signal and the N matched filters is specific to the type of signals used. By using the N-dimensional number representing the output of the matched filter, the encoder/decoder modules are transparent to the exact implementation details of the channel and detection. This is considered an advantage so that the user of the encoder/decoder modules can simulate the performance of the *convolutional code* and/or the signal space of the overall system.

Note : A requirement for the simulation results to be correct is that the measure of distance between valid signal elements in the N-dimensional space be the Euclidian distance.

The generalized encoder and decoder program modules were written to implement the structure of figure(4.6). All the information about the encoder is read from a configuration file. For readability, and ease of changing, the configuration file format was chosen to be text. A comment delimiter is defined to allow user comments in the file.

In brief, the configuration file defines the values for m, n, $\tilde{m}$, σ and N of figure(4.6), as well as the tables to implement the differential encoder, the convolutional encoder and the signal element mapping. Refer to Appendix D for detailed information on the file format. Included in the Appendix is the configuration file

required for the V.32 recommendation for TCM, and a configuration file for uncoded 16-QAM as used in the simulation.

Testing the Generalized Encoder/Decoder Modules

The test procedure used for the generalized encoder and decoder program modules was to interchange the input and output results of the V32 specific program modules with the generalized encoder/decoder modules. The results confirmed that the programs operated identically for the same input information.

Application to Uncoded Modulation

The generalized encoder/decoder module removed the need to create program modules to implement the uncoded version of the V32 recommendation for 9600 bits per second modulation. The uncoded simulation could be simply done by defining a suitable configuration file. The convolutional encoder table of the configuration file becomes a single entry – there are no inputs, and there are no outputs. The single entry is required because the indexing numbers into the table is based on binary number bit positions. This effectively gives 2^i where i is the bit position. Since $2^0 = 1$ there must be one entry for configurations that do not have a convolutional encoder. The configuration file for the uncoded 16-QAM option of the V.32 recommendation is shown in Appendix D.

The correct operation of the generalized encoder and decoder modules gave the author a large degree of confidence in the correctness of the algorithms used in the program modules. The configuration file for the uncoded option contains entries of zero for $\tilde{m}$ (the number of bits into the convolutional encoder), and for σ (the number of memory bits) as defined in figure(4.6). In the generalized encoder and

decoder modules, zero is a boundary condition. In practice boundary conditions are often the source of program errors.

CHAPTER 5

APPLICATION OF SIMULATION AND RESULTS

This chapter discusses the expected results of the V32 simulation for a channel with additive Gaussian noise, and details the simulation results obtained for that model. Some predictions for the results are explained. For a comparative reference, the uncoded option of the V.32 recommendation is also simulated.

A section illustrates how to use the simulation tools to test the rotational invariance of a encoder/decoder implementation. The commands are shown in the text to give an idea of *how* the tools are actually used.

A channel with phase jitter is simulated as an example of the applying the simulation to a set of conditions.

5.1 EXPECTED RESULTS

Wei shows in his 1984 paper[12] that the convolutional code and signal mapping used in the V32 recommendation has a coding gain of 4.0 dB over that of 16-SQ QAM used for the uncoded modulation at the same data rate.

As discussed in Chapter 1, the symbol error rate is not only dependent on the coding gain of the convolutional code. The symbol error rate is proportional to the average length of *error events*. For the TCM specified by the V.32 recommendation, the coding gain of the code used is set by the minimum free distance d_{free} of two code sequences [12] (rather than the distance between signal elements of parallel transitions). This implies that any decoding error is likely to result in an error event

whose average length that is greater than one. Thus the simulation results should show a real coding gain a little less than 4.0 dB below the *knee* of the symbol error rate diagram of figure(5.2). The knee of the symbol error rate diagram is where the symbol error rate versus signal to noise ratio curve shows an accelerated decline in probability of error for an increase in signal to noise ratio.

There is another effect that should be expected as a consequence of the minimum free distance being set by the distance between two code sequences. In the region where the probability of a decoding error becomes high, the coded modulation can be expected to perform *worse* than the uncoded modulation. This is a direct result of a decoding error resulting in an error event with an average length of more than one. In practice, this is not a problem because at the knee region of the symbol error rate diagram the bit error rate is generally considered unacceptable for the modem to be used at all.

5.1.1 Theoretical Symbol Error Rates for 16-QAM

The probability of a error event for M-ary QAM can be shown to be : [5, Pages 318-321]

$$P_e' \simeq 2\left(1 - \frac{1}{\sqrt{M}}\right) \operatorname{erfc}\left(\sqrt{\frac{3E_{av}}{2\left(M-1\right)N_0}}\right) \tag{5.1}$$

N_0 is the power spectral density of the noise in the system. Equation(5.1) assumes matched filtering, and is therefore applicable to these simulations. E_{av} is the average energy of the signal assuming that all the signal elements are equiprobable. Equation(5.1) is an approximation that is accurate for small values of P_e (small meaning $P_e \ll 1$, e.g 0.01).

For the 16-QAM specified by the V.32 recommendation $M = 16$ and $E_{av} = 10$. (Both the encoder and decoder program modules compute the RMS value of the

signal mapping. The average power of each signal element of the constellation is obtained by squaring the RMS value.) By applying the known values of M and E_{av}, equation(5.1) can be simplified to :

$$P_e' = \frac{3}{2}\text{erfc}\left(\sqrt{\frac{1}{N_0}}\right) \tag{5.2}$$

The 16-QAM system specified by the V.32 recommendation uses differential encoding to make the system insensitive to phase ambiguities of multiples of 90°. This results in a single decoding error event of the demodulator producing an error event after the differential encoder. The length of the average error event can be computed as follows :

Consider the four level signal constellation seen by the I or Q channel correlation

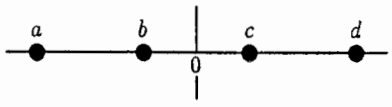


Figure 5.1: *Line representation of the I (or Q) channel*

receiver, as indicated in figure(5.1). The measured received signal can lie anywhere on the line. The four level signal is represented by two binary digits. One of the binary digits is mapped to the differential decoder, and the other bit is not. The constellation points are mapped so that a transition from one side of the zero line to the other will change one the bit mapped to the differential decoder. A single error that passes through the differential decoder causes *two* consecutive errors at the output of the differential decoder. A single error in the other bit does not affect the differential decoder, and hence only causes a single error to occur in the overall system. The average length of an error event is calculated as

$$l_{ave} = P_{e1} \cdot 1 + P_{e2} \cdot 2 \tag{5.3}$$

where P_{e1} and P_{e2} are the probability of an error events of length 1 and 2 respectively. For each side of the zero line of figure(5.1), there are two ways for a decoding error

that does not cross the zero line, namely transmitted symbol a could be received as symbol b and transmitted symbol b could be received as symbol a. Similarly, the decoding error that crosses the zero line is if transmitted symbol b is received as symbol c. Note that the probabilities of a transmitted symbol being received in error as a symbol that is not adjacent to the transmitted symbol are ignored. This is valid since at low error probabilities, the chance of doubling the error resulting from the Gaussian noise is very small. Thus we can conclude that $P1_e = 2/3$ and $P2_e = 1/3$. Applying these figures to equation(5.3), we have

$$l_{ave} = 1\frac{1}{3}$$

Every error event predicted by equation (5.2) results in an error event with an average length of l_{ave}. Thus is theoretical probability of error for the V.32 implementation of 16-QAM is given by

$$P_e = 2 \cdot \text{erfc}\left(\sqrt{\frac{1}{N_0}}\right) \tag{5.4}$$

5.1.2 Computing Noise Variance from SNR

The require noise on the matched filter output can be adding a Gaussian distributed variable with zero mean an a variance $\sigma_n^2 = N_0/2$. For a given SNR, σ_n may be calculated as follows :

Repeating equation(4.3),

$$SNR = \frac{2E}{N_0}$$

In dB's, this may be re-written as

$$SNR = 10\log\left(\frac{E}{\sigma_n^2}\right) \tag{5.5}$$

Thus σ_n^2 may be written in terms of the input SNR ratio (in dB's) as :

$$\sigma_n^2 = E_{av} \cdot 10^{\left(\frac{-SNR}{10}\right)} \tag{5.6}$$

E_{av} is used in equation(5.6) so that correct average noise power is maintained for equation(5.4).

Equation(5.6) also applies to the V.32 TCM, provided that E_{av} used is correctly obtained for the 32-CR signal constellation used for the TCM code. It turns out that the average energy is the same as the 16-QAM constellation, (i.e. $E_{av} = 10$).

5.2 Simulation Procedure

Using the above equations, it was established that the range of SNR = 20dB ... 34dB theoretically covered symbol error rates in the range $0.63 \ldots 10^{-6}$. The simulation procedure followed was :

1. A batch file was generated to process the simulation automatically. The simulation parameters are set up using environment variables[1]. This allows the simulation parameters used in the batch file to be modified without editing the batch file.

2. For each SNR used, the applicable value of σ_n was computed along with the theoretical symbol error rate.

3. The simulation was run with a sample count of 10000 to estimate the value of P_e. The simulation was then re-run using a higher value sample count if necessary. The criteria for establishing whether an extra run is necessary is :

$$\text{Minimum}\left(N = \frac{100}{P_e}, 2 \cdot 10^6\right)$$

$2 \cdot 10^6$ was set as the upper limit in order to keep the simulation times down. The limit applied allows P_e to be measured accurately down to 1.3×10^{-5}. See Appendix E for statistical details of this statement.

[1]Environment variables (in MSDOS context) are named variables that can be read and written to using standard MSDOS commands.

4. The simulation results were tabulated, and the confidence interval calculated for the results.

The command file implemented the instructions to execute the necessary program modules and applied the necessary piping to the data. The process of the simulation commands is summarized below :

- Generate a data file containing the required number of random symbols.
- Modulate the random data symbols using

  A) The configuration file for 16-QAM.

  B) The configuration file for V.32 TCM.

- Add Gaussian distributed noise sample with the suitable standard deviation to the I and Q components of the matched filter representation of the signal.
- Decode the data files of the 'noisy' data and place results into a file.
- For both the 16-QAM and the V.32 TCM compare the decoded data file with the original data file and report the results.
- Delete temporary files on the disk.

5.3 Simulation Results

The results of the simulations are shown in table(5.1) and in the graph of figure(5.2).

5.3.1 Discussion of Results

The results indicated in figure(5.2) are generally as expected. Several points may be noted about the results.

Simulation Results for Symbol Error Rates					
		16-QAM		V.32 TCM Code	
SNR	Theory P_e	Error Prob P_e	90% CI	Error Prob P_e	90% CI
16		0.7268	0.0070	0.9002	0.0078
20	0.6346	0.5087	0.0083	0.8228	0.0106
24	0.2260	0.2110	0.0053	0.4929	0.00817
26	0.0920	0.0867	0.0034	0.1214	0.00405
27	0.05035	0.04968	0.00106	0.02235	0.00071
28	0.02402	0.02421	0.00057	0.00263	0.00019
29	0.00965	0.00972	0.00036	0.00035	6.8E-5
30	0.00313	0.00328	0.00021	1.2E-5	3.8E-6
31	7.76E-4	0.00074	8.2E-5	< 2E-7	
32	1.37E-4	1.05E-4	1.7E-5		
33	1.59E-5	1.2E-5	4.0E-6		
34	1.08E-6				

Table 5.1: *Simulation Results for V.32 TCM and V.32 16-QAM codes for 9600 bits/s in the Presence of Additive Noise*

1. The number of samples used in the simulation limits the range of significant answers to error rates of $P_e > 1 \cdot 10^{-5}$. The number of samples required is inversely proportional to the symbol error probability.

2. For SNR of more than 26.5 dB the TCM system performs better than the uncoded 16-QAM system. This crossover occurs at a symbol error rate of approximately 0.06.

3. The expected poorer performance of the TCM code at 'high' symbol error rates does occur.

4. At a symbol error rate of $P_e = 10^{-5}$, the TCM system operates at about 3.1dB lower signal to noise ratio than the 16-QAM system. The gap appears to widen slightly at lower signal symbol error rates. This figure is less than the 4.0 dB coding gain of the convolutional code used. This difference was expected, as discussed earlier.

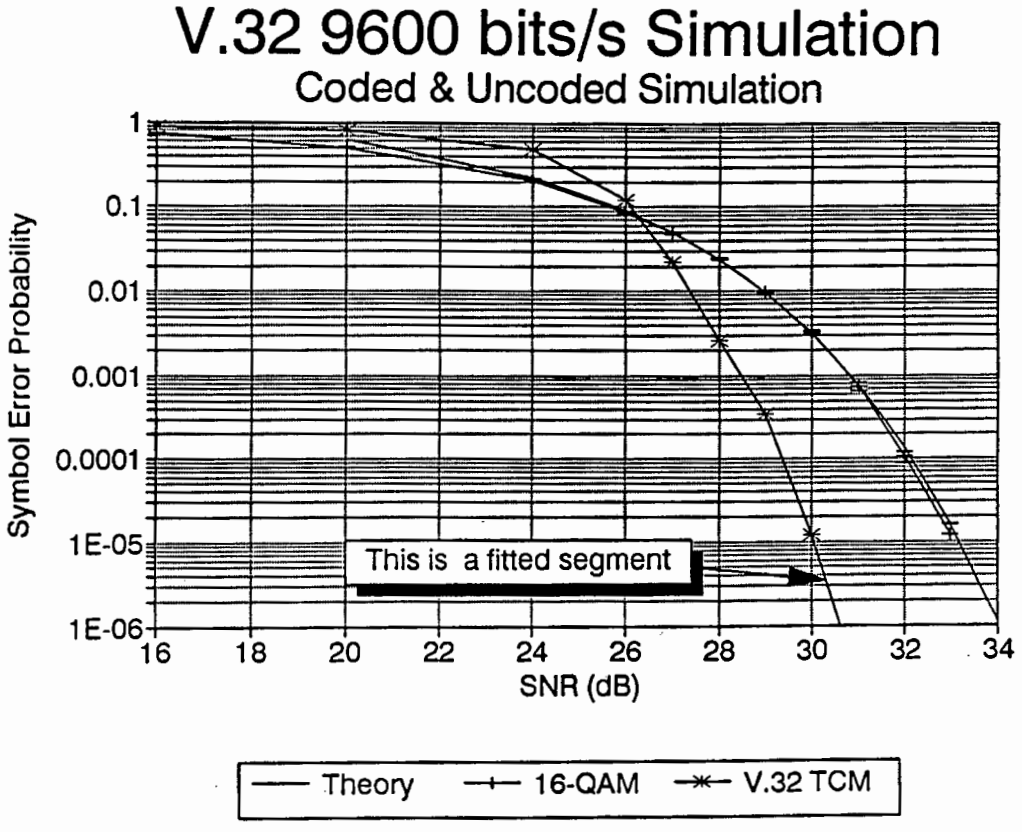


Figure 5.2: *Graphical Results for V.32 TCM and V.32 16-QAM codes for 9600 bits/s in the Presence of Additive Noise*

5. For symbol error rates of less than 0.05, the simulation results for the 16-QAM system compare excellently with the theoretical symbol error rates. This is a strong evidence that the simulation implementation is correct. Deviations of the simulated results from the theoretical results is expected at high symbol error rates. This is because several simplifying approximations are made in the derivation the theoretical results that assume a small probability of error. The results justify these simplifying approximations.

5.4 Testing Rotational Invariance

Using the simulation tools developed thus far, it is possible to test the rotational invariance of QAM type modulation schemes. Consider a modulation scheme that is rotationally invariant to α radians. There will be $N = 2\pi/\alpha$ possible phase rotations that should produce the correct demodulation. On the I-Q representation of the baseband signal, a phase rotation may be produced by multiplying each complex number representation of the I-Q number by the complex constant $e^{i\alpha}$. Since $|e^{i\alpha}| = 1$ for all α, this operation does not change the magnitude of the signal. The Data Master module 'MATH' can do such a scaling. It is unnecessary to apply noise to the channel signal representation when testing rotational invariance.

After demodulating the phase rotated channel data, the results should produce only a single error event at the point where the phase rotation occurred. In the case of the uncoded 16-QAM, the error event is always a single error[2]. For the V.32 TCM, the error event may be several symbols long. If the implementation of the modulation is not rotationally invariant, the phase change will cause a catastrophic failure of the system where all subsequent data after the phase rotation will be incorrect (except for the data the is *by chance* the same as was transmitted).

The rotational invariance was tested for both the coded and uncoded versions of the V.32 modulation using the following commands within the Data Master shell :

```
rdata 100 -s1 -n16 !  encoder -cxxxx.cfg !  atob !  ...
math -scl0;1 !  btoa !  decoder -cxxxx.cfg !  dcomp -n16 -s1
```

The command `math -sclx;y` scales the input data by the complex number (x,y). For the above example, $(0,1) = e^{i\pi/2}$. The results showed a single error for the

[2]A phase change is like a step function - there is only one transition. Therefore it never causes a error event of length more than one, as is possible for a symbol error. A symbol error that affects the differential decoder will cause an error event of length 2.

Phase Rotation	No. of Errors	
	16-QAM	V.32 TCM
0	0	0
$\pi/2$	1	3
π	1	2
$3\pi/2$	1	4

Table 5.2: *Results of Rotational Invariance Tests*

uncoded 16-QAM and between two and four incorrect symbols for the TCM coded option. The modulation scheme is selected by using the appropriate configuration file name for *xxxx.cfg*. The results are summarized in table(5.2).

The multiple symbol errors found using the TCM configuration gives a rough estimate of the length of an average error event. From table(5.2), the average length of the error event due the applied phase rotations is 3.

5.5 Applying the Simulation to Phase Jitter

This section deals with the practical application of the TCM simulation program modules to investigate the effects of phase jitter on the performance of the V.32 TCM system.

Phase jitter is the random variation of the phase angle of a signal. Often this is due to the phase characteristics of the channel carrying the signal. Phase jitter is a general characteristic of the general switched telephone network. Another source of phase jitter is the carrier recovery circuit used in the demodulation circuits. Like all electronic circuits, there are thermal noise effects. The thermal noise eventually manifests itself as phase jitter. (In a well designed carrier recovery circuit this phase jitter may be negligible).

For the purposes of this evaluation of the effects of phase jitter, the phase jitter model shall be a Gaussian distributed phase error with an RMS phase error equal to the standard deviation of the Gaussian distribution used. Phase jitter is normally a

$1/f$ type of noise. A $1/f$ distribution can be modelled by integrating (or accumulating) samples from a Gaussian distribution. This resulting $1/f$ distribution is a noisy waveform that has large low frequency component. The carrier lock circuits, if operating correctly should be able to track the low frequency components of the $1/f$ noise. This effectively subtracts the low frequency components from the noise, resulting in a new distribution that is almost Gaussian in nature. A more accurate model was not deemed necessary in this application here, since the simulation is intended to illustrate the application of the tools rather obtain accurate results for phase jitter effects.

The simulation of phase jitter was applied to data at a constant signal to noise ratio of 30dB. From table(5.1), the probability of a symbol error with no phase jitter is $P_e = 1.2 \cdot 10^{-5}$. Phase jitter was added to the channel signal by using the developed tools, operating under the Data Master shell. The Data Master shell command line used to implement the simulation is :

```
RDATA nnnn -n16 -s1 !  ENCODER ! ADDRAND ...
-g -pxxxx -d0.3162278 !  DECODER ! DCOMP -s1 -n16
```

nnnn is the number of samples used for the simulation run, and *xxxx* is the RMS value of the phase jitter for the simulation. See Appendix C for information on the other program options. This command line implements a simulation for the structure of figure(5.3).

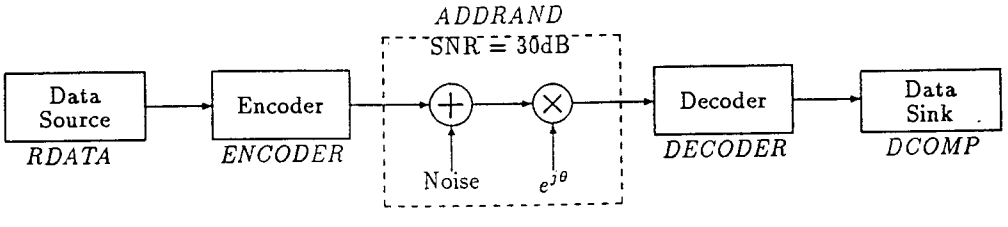


Figure 5.3: *Simulation Structure for Phase Jitter effects*

5.5.1 Phase Jitter Simulation Results

The simulation was initially applied for small values of RMS phase jitter, and increased until the resultant probability of error reached approximately $1 \cdot 10^{-3}$. The results are summarised in table(5.3).

The last entry in the table is taken from the original simulation without any phase

RMS Phase Deviation (rad /°)	SNR	No. Samples	Error Events	Prob. Error P_e	90% C.I.
0.1 / 5.73	30	$1 \cdot 10^5$	449	$4.49 \cdot 10^{-3}$	$3.49 \cdot 10^{-4}$
0.071 / 4.05	30	$1 \cdot 10^6$	362	$3.62 \cdot 10^{-4}$	$3.13 \cdot 10^{-5}$
0.05 / 2.86	30	$1 \cdot 10^6$	61	$6.1 \cdot 10^{-5}$	$1.28 \cdot 10^{-5}$
0.022 / 1.28	30	$1 \cdot 10^6$	14	$1.4 \cdot 10^{-5}$	$6.2 \cdot 10^{-6}$
0.01 / 0.573	30	$1 \cdot 10^6$	9	$9 \cdot 10^{-6}$	$4.9 \cdot 10^{-6}$
0	30	$2 \cdot 10^6$	24	$1.2 \cdot 10^{-5}$	$4.0 \cdot 10^{-6}$

Table 5.3: *Results of Phase Jitter applied to the V.32 TCM system*

jitter.

The results show that for an RMS phase jitter exceeding 1.28° there is a significant reduction in the performance of the V.32 TCM system. The actual degradation of the performance would best be assessed by generating a family of data such as in table(5.3) using other values of SNR. If the results were plotted as a family of curves for each SNR, it would be possible to read the actual code performance degradation from the curves. Such an exercise is beyond the scope of this thesis – the results presented here are intended only to illustrate the application of the program modules.

Chapter 6

Discussion and Conclusions

6.1 Discussion

General

This thesis was intended to to provide a platform for researching the topic of trellis coded modulation, and to provide a simulation for trellis coded modulation. The V.32 recommendation was used as an example of a trellis coded modulation system on which the simulation programs could be tested.

The development of the simulation presented in this document provided a framework wherein the author developed an understanding of the previously unknown topic of TCM. The research started with the studying of the early papers on the topic such as [10][11][12]. After achieving a moderate understanding of the principles of TCM the author proceeded to implement the simulation. It was found that an in-depth understanding of the principle of operation of TCM was required to make judgement on suitable structures for the software model of the simulation, and for the algorithms to implement the simulation.

This thesis provides detailed descriptions the relevant information required to understand the operation of TCM as implemented in the simulation. The purpose of this is twofold.

1. The description of a concept *requires* an in depth understanding of the topic in question.

2. The author wished to supplement the shortcomings encountered in the descriptions used as references for this thesis.

DATA MASTER

The application of the Data Master file compatible tools provided a base for the development work to start with. The Data Master file data interchange format proved to provide all the facilities necessary for the simulation. There were, however, problems with using Data Master - mainly with the implementation of the original Data Master tools. Two problems were identified when the Data Master tools were applied to the simulations.

1. The 'RANDOM', 'MATH' and 'ATOB'[1] modules do not handle files containing more than 32767 (or $2^{16} - 1$) data points. This is considered an error by the author since the file format used by the Data Master tools provides for files of up to 2^{32} records. The same limit may apply to the other Data Master tools, but were not identified since no other tools were used in the simulation of this document.

2. The 'RANDOM' Data Master tool does not generate an accurate Gaussian distribution. This was revealed by the poor results initially obtained when the simulation was processed. The results documented here were all obtained after developing another random number generator 'ADDRAND'. The random number generator used in 'ADDRAND' was obtained from the software diskette accompanying the book [7].

Overall, the Data Master tools provided a useful means of testing ideas and provided the necessary processing for simulations early in development. However, for

[1] See *PC Data Master* distribution manual for detailed information.

large data runs, such as those required for probability of error simulations, custom processing modules had to be developed to overcome the Data Master shortcomings.

Simulation Structure

The generalized encoder/decoder processing modules do provide a general set of tools for working with TCM codes. Additionally, three other support processing modules were developed for generating source data, adding noise and phase jitter, and for testing the results for decoding errors. All of the simulations used to generate the results in this document (with the exception of the *trial* BPSK system) were implemented using these five tools.

The modular structure of the tools allows additional tools be integrated into the overall system. This allows the functionality of the system to be increased as necessary.

The transfer of data via piping requires that there be sufficient space on the file storage device. This device would normally be a hard disk drive, where sizes exceeding 200MB are common now. An estimate of the space requirements of a typical simulation consisting of one million complex signal elements is 30MB. This is based on an average of 7 characters to represent a real number. A complex number requires 2 real numbers and a separator, requiring 15 bytes per complex signal element. At least one input file and one output file must exist on the disk at the same time.

Longer data runs require proportionally more space. At some finite length of data, the whole philosophy of passing processed data via files would become impractical. To overcome such a problem it would be necessary to combine all of the functions of the various processing tools into a single program module that processes the data as it is generated. This eliminates the need for storing intermediate

results for the entire data run. This approach would make the resultant program a single solution to a single problem. Any change in the structure of the simulation process would require the rewriting of the source code. Thus such a solution could not be considered a general tool.

Equivalence to Programming

In principle, the application of tools such as those provided by Data Master is a form of high level programming. The additional modules developed for the simulation in this thesis provide extensions to the available instructions. The resulting 'language' has a limited scope of application, viz. digital signal processing. As such, the language cannot be considered a general purpose tool, but rather a specialized language to solve a specialized type of problem.

A single processing module does not have enough functionality to be considered useful. However, when the modules are combined, the processing unit formed has the potential to do some meaningful processing (depending of course on *how* they are 'programmed' as a unit).

This view of the structure highlights the benefit of the processing modules or tools type of approach used in the simulation implementation. The tools can be reused in new 'programs' to implement some other useful processing. Additionally, the scope of the language can be increased by adding new commands or processing modules. This is in contrast to a single processing module being developed to implement all the processing required for the simulation of this document.

Possible Applications for the Simulation

The Data Master ASCII file format used by the program modules for the simulation allow simple inspection and modification of the data transferred between modules.

This allows an experimenter to try some 'what if ...' type of experiments and see what the results are. For example, for the V.32 TCM code, it is possible to take the output of the encoder module and modify a signal element to find out how the decoder deals with the perturbation. A single error event can be forced by making the perturbation large enough. (The decoder module does not limit the magnitude of the input signal representation). This will allow a reasonable estimate of the average length of an error event.

The use of a configuration file to define the TCM system for both the encoder and decoder modules allows the user to design their own TCM system, and to verify the operation of the design by simulation. It may be instructive for a senior university course on digital communications to set as an exercise the design of a relatively simple trellis code. The design could then be tested using the simulation tools provided by this thesis.

The testing of a rotationally invariant code is easily achieved by using the simulation, as discussed in chapter 5. The verification of a rotationally invariant TCM is a tedious task to do by inspection. This in itself provides ample justification for the development of the simulation environment.

The author feels that a code of the sort of complexity of the example used in chapter 2, section 2.3 is ideal for illustrating the application of rotational invariance with TCM. Simpler examples would be more suitable if rotational invariance was not included with the problem.

6.2 Conclusions

- The simulation results confirm the improved performance of TCM systems over systems not using redundant coding, but with similar bandwidth and power requirements.

- The simulated results for the uncoded 16-QAM modulation used in the V.32 recommendation compared excellently against the theoretical results.

- The implementation of the simulation required the author to develop an in depth understanding of *how* TCM systems operate. Based on such understanding the author was able to make effective decisions about the implementation of the software so that the resulting programs could be considered a re-usable tool.

- The use of soft decision viterbi decoder in the demodulator/decoder program module tends to be optimal decoding, provided the truncation depth is suitably large. This is necessary if the demodulator/decoder is to evaluate the true performance of the system under simulation.

- The use of the sampled matched filter representation for the channel signal has allowed the TCM modules to be generalized to handle channel signal formats other than the QAM format used for the V.32 recommendation. The exact demodulation process need not be known in order to investigate the behaviour and performance of the code. In order for the Viterbi algorithm to work optimally, the Euclidian distance between different signal elements must reflect the true measure of distance between the signal elements.

- The simulation can be applied to uncoded modulation formats, which is useful for comparative performance simulations.

- The simulation tools developed with this dissertation operate correctly in the intended scope of the simulation.

- The software tools are extendable. If additional processing is desired for any reason, (such as a implementation of a matched filter), new modules can be easily integrated into the overall simulation.

- The use of an ASCII configuration file for the TCM encoder and decoder program modules allows the user to define different TCM systems, without having to modify any source code.

- The modular structure of processing tools allows the simulation tools to be re-used for other simulations.

- The Data Master tools used in the simulation provided a suitable data interchange format. The shortcomings found in the implementation of the original Data Master tools precluded using the Data Master tools as part of the simulations used in this dissertation.

APPENDIX A

BPSK TRIAL SIMULATION

Binary phase shift keying (BPSK) is a simple form of modulation that uses two *antipodal* sine wave signals to represent one of two channel states. Algerbraicly, the pair of signals, $s_1(t)$ and $s_2(t)$ are used to represent the binary symbols 1 and 0 respectively. $s_1(t)$ and $s_2(t)$ are defined as

$$s_1(t) = A\cos(2\pi f_c t) \tag{A.1}$$

$$\begin{aligned} s_2(t) &= A\cos(2\pi f_c t + \pi) \\ &= -A\cos(2\pi f_c t) \end{aligned} \tag{A.2}$$

The carrier frequency f_c is chosen such that $f_c = n_c/T_b$ for some integer n_c. $1/T_b$ is the bit rate of the system. This definition of the carrier frequency results in an integral number of cycles for each bit interval.

Assuming a matched filter receiver and additive Gaussian noise with a power spectral density of $S(\omega) = N_0/2$, it can be shown [5, pp275-279] that the probability of error for BPSK is

$$P_e = \frac{1}{2}\text{erfc}\left(\sqrt{\frac{E_b}{N_0}}\right) \tag{A.3}$$

E_b is the energy of the signal per bit of information, and is related to the amplitude A by

$$E_b = \frac{A^2 T_b}{2}$$

The matched filter channel representation for the BPSK system only requires a single channel (or dimension) since the output of the matched filter is a real number.

Only one correlator is required in the matched filter since the two signal elements $s_1(t)$ and $s_2(t)$ are antipodal. Notice that the magnitude of matched filter signal representation of the channel signal is equal to E_b and ***not*** the amplitude A of the time domain representation of the signal.

The derivation of a standard deviation σ of a Gaussian distributed random variable for a given signal to noise ratio (SNR) on the channel follows is the same as shown in section 5.1. Repeating equations (5.5) and (5.6),

$$SNR = 10 \log \left(\frac{E_b}{\sigma_n^2} \right)$$

and

$$\sigma_n^2 = E_b \cdot 10^{\left(\frac{-SNR}{10}\right)}$$

The above information was applied to calculate theoretical symbol error rates for SNR's in the rang of 2.0 dB's to 10.0 dB's. Simulations were run for 30 000 sample points to establish the simulated symbol error rates. The simulation used a signal amplitude that had a signal energy $E_b = 1.0$. The results obtained are tabulated in table(A.1).

SNR (dB)	σ_n	Theoretical P_e	Measured P_e	90% C.I
2	0.7943	0.1038	0.1049	0.0037
3	0.7079	0.0793	0.0792	0.0032
4	0.6310	0.0571	0.0561	0.0027
5	0.5623	0.0375	0.0399	0.0023
6	0.5102	0.0228	0.0210	0.0016
7	0.4467	0.0125	0.0114	0.0012
8	0.3981	0.0060	0.0056	0.0008
9	0.3548	0.0024	0.0022	0.0005
10	0.3162	0.0008	0.0007	0.0003

Table A.1: *Theoretical and Simulated results for BPSK modulation*

The resulted show an excellent correlation between the expected (theoretical) results and the results from the simulation. The values of the 90% confidence interval (90% C.I.) here are derived by applying the results of Appendix E.

Appendix B

Data Master file Interchange Format

The information in this appendix deals with the file format used in the simulation programs developed. The file format is defined in [3, p.35]. The information is duplicated here as a reference.

ASCII Data File Format

In order to be compatible with the signal processing application programs (of PC Data Master), ASCII data files should have the following format.

A data header containing as individual text lines :

the letter 'A'
the number of samples per channel
the number of data channels
the sample interval multiplier
the sample interval exponent
the data title
the label for the sample interval (e.g. "time")
the units for the sample interval
the label for channel 1
the label for channel 1
the label for each additional data channel

comment text lines

a blank text line indicating the end of the header

Data lines containing as individual integer of floating format text strings :

sample 1 for channel 1

sample 1 for channel 2

⋮

sample 1 for channel m

sample 2 for channel 1

sample 2 for channel 2

⋮

sample n for channel 1

sample n for channel 2

⋮

sample n for channel m.

where: physical sample interval = multiplier $\times$ 10^{exponent}

The ASCII data records could also be in the following format :

s1c1 s1c2 s1c3 ...s1cm

s2c1 s2c2 s2c3 ...s2cm

⋮

snc1 snc2 snc3 ...sncm

where sncm denotes sample n on channel m.

Problem using Data Master with Turbo-C

During initial program development, an incompatibility was identified when using Turbo-C program modules running under the Data Master shell. Using the ANSI

defined function `fopen(const char *fname)` of turbo-C, the Data Master shell does not honour the file open request. This causes the program to fail with a runtime error.

This problem can be overcome by using a non-ANSI defined function calls `creat(const char *path, int amode)` and `fdopen(int handle, char *type)`. These functions are generally available on UNIX implementations of C. In the program modules implemented, a conditional define was used to try and develop portable code.

Using the following code extract, it was possible to make the programs to compile unmodified under both Turbo-C and Microsoft C.

```
#ifdef __TURBOC__
#define fopen_rd(f) fdopen(open(f,O_RDONLY|O_TEXT),"rt")
#define fopen_wr(f) (_fmode=O_TEXT,fdopen(creat(f,S_IFREG|S_IWRITE),"wt"))
#else
#define fopen_rd(f) fdopen(open(f,O_RDONLY|O_TEXT),"rt")
#define fopen_wr(f) fdopen(open(f,O_WRONLY|O_CREAT|O_TRUNC|O_TEXT,S_IWRITE),"wt")
#endif
```

During program execution, only the `fopen_rd(f)` and `fopen_wr(f)` are used to open files for reading and writing respectively.

Data Master Piping

Under the Data Master command shell, the pipe command "!" actually generates a temporary file name and passes that name to the program module being executed as a command line option. This is in contrast to the operating system pipe which actually redirects the standard output device to a file. Under the Data Master shell the command

```
data.asc !  atob !  data.bin
```

will take the the ASCII file format input file **data.asc** and convert it using the program module **atob** to a binary format file **data.bin**. The command line generated by Data Master to execute the instruction would be

```
atob data.asc data.bin
```

As the above example illustrates, an input file name is added as a command line option. The output file name is then also added as a command line option. Any other options specified for the program module would be given on the command line *before* the file names are given.

The author felt it desirable to support both Data Master piping and direct piping via the operating system. This was achieved by writing the programs to accept all input from the standard input device, and write all output to the standard output file device by default. Support was added for Data Master style piping by parsing the command line for file names at the end, and re-mapping the standard input and output devices the the appropriate files given on the command line. For examples of this, study the source code of any of the program modules (such as `encoder.c`).

APPENDIX C

SUMMARY OF PROGRAM MODULES

GENERAL

All the program modules implemented for the simulation have common features. Every program accepts and/or generates data in the ASCII Data Master file format described in Appendix B. As described in Appendix B, the program modules can either operate on files specified on the command line (as required for Data Master piping), or use the standard input and output devices of the operating system for data files (as required for piping using the operating system).

The command line for the programs have the following general format :

`prog_name` [<count>] [-<option>] [-<option>] ...[<*input_file*>] [<*output_file*>]

In the above notation square braces [...] denote optional items, and are not entered on the command line. Angle braces <...> denote an option name, and is not literal text. If the option is specified, it replaces the name *including the angle braces.* The ordering of the command line options is significant. You cannot specify a <count> after entering options -<options>. Similarly, you cannot specify the data files before the -<options>.

<count> is used to specify the number of data elements to be generated by the program. This option is only used when the program module used is a data source (such as RDATA). Normally the file size is set by one or more of the input files.

-<option> are zero of more occurrences of the command line options for the program. Using the command line options, the user can adjust the action of the program module. The options are specific to each module. Options include such actions as setting the random number distribution type, starting states etc. See the program option listings later in this appendix for more details.

<input_file> specifies the name of the input file to be used by the program. If no name is specified, the program defaults to the standard input device. In the case of a module that only generates output data (such as RDATA), there is no input file.

<output_file> specifies the name of the output file to be used by the program. If no name is specified, the program defaults to the standard output device. In the case of a module that only processes input data (such as DCOMP), there is no output file.

For program modules that process both input and output files (such as ENCODER), leaving off on of the file names will result in the single file name specified on the command line being used for the input file. The output file will be the standard output device.

Program Modules and Options

RDATA

This module generates a sequence of numbers used to represent input symbols to the modulator. The distribution of the numbers is uniform.

Syntax : RDATA [#n] [-nx] [-rxx] [<output file name>]

#n is the number of samples to generate. If not specified, the default is 1024.

-nx specifies the range of the samples. Numbers will have the values 0..(x − 1). Default is -n2 (ie. binary).

-rxx specifies the data rate of the symbols. Used to set the time interval between symbols only. The default -r2400.

-sxxx specifies the starting 'seed' for the random number generator. If not specified, then the C function `randomize()` is used.

<output file name> is where the name of the DataMaster file where the data will be placed. If not specified the output will be written to STDOUT.

ENCODER

ENCODER is the module that implements trellis coded modulation of an input data stream. The output data is a sequence of COMPLEX numbers representing the modulated data as the M-dimensional position of the signal.

Syntax : ENCODER [-c<config file>] [-sn] [-?] [-!] [<infile> [<outfile>]]

-c<config file> specifies the configuration file for encoder implementation. If not specified, it defaults to DECODER.CFG.

-sn sets the starting state for the convolutional encoder of the TCM system. Defaults to 0. Valid range is defined by the configuration file.

-! This option is used to check if the configuration file is OK. Parses the configuration file and reports any errors. No data is processed regardless of the other arguments.

-? Display the help screen.

<infile> is the name of the input file containing DataMaster format of integer numbers of range $0 \ldots N-1$ where N is the number of unique values the input data can take on. If not specified, it defaults to STDIN.

<outfile> is the name of the DataMaster ASCII format output file containing the M numbers. If not specified, defaults to STDOUT.

The details of the configuration file are discussed in Appendix D.

DECODER

DECODER is the module to implement trellis coded demodulation of an input data stream. The output data is a sequence of symbols (numbers) representing the data as decoded by the decoder.

Syntax : DECODER [-c<config file>] [-sn] [-?] [-!] [<infile> [<outfile>]]

-c<config file> specifies the configuration file for the encoder implementation. If not specified, it defaults to DECODER.CFG.

-s*n* sets the starting state for the convolutional encoder of the TCM system. Defaults to 0. Valid range is defined by the configuration file.

-! This option is used to check if the configuration file is OK. Parses the configuration file and reports any errors. No data is processed regardless of the other arguments.

-? Display the help screen.

<infile> is the name of the input file containing DataMaster format of M channel numbers where M is number of signal dimensions of the matched filter signal representation. If not specified, it defaults to STDIN.

<outfile> is the name of the DataMaster ASCII format output file containing the output (decoded) symbols. If not specified, defaults to STDOUT.

The configuration file used is *exactly* the same as used for the ENCODER module.

ADDRAND

The ADDRAND module adds noise to a input file sequence of numbers, using either a uniform distribution or Gaussian distribution. Both the input and output files are ASCII Data Master format files. Additionally, the module can add a random phase variation to the output resultant data. The phase variation is always Gaussian distributed.

Syntax : ADDRAND [-n<xx>] [-g] [-m<xx>] [-d<xx> [<input file name>] [<output file name>]

-n<xx> is the number of samples to generate. If this option is included, there is no input file, and the output file of <xx> points consists of the random generated data only.

-r<xx> specifies the data rate of the symbols. Used to set the time interval between symbols only. Default is 2400. This is only used if the -n<xx> option is specified, otherwise the rate is set by the input data file.

-g specifies the random numbers must have a Gaussian distribution. Otherwise uniform distribution is used.

-m<xx> specifies the mean value of the random numbers. Defaults to 0.0

-d<xx> Set the deviation of the random numbers. For the uniform distribution this is the difference between the peak and the mean. For Gaussian distribution, this specifies the standard deviation of the numbers.

-p<xx> Set the deviation of the random phase variation (Gaussian distribution). If not specified, the phase deviation is 0.0 (i.e. no phase variation).

-s<xx> Specify the starting seed of random number generator. Default is random. 0 also gives random.

-? Display a help message

<input file name> is the input file from which the numbers will be added to. If not specified, STDIN is used. This is not present if the -n<xx> option is used.

<output file name> is where the name of the DataMaster file where the data will be placed. If not specified the output will be written to STDOUT.

DCOMP

This program compares two Data Master ASCII data files for differences to establish en error rate. The program displays a summary of the difference statistics to the standard error device of the operating system. (This prevents the output from being re-directed to a file by the operating system. This was done since the difference statistics are textual information, and does not comply with the Data Master ASCII file format). If the original data was generated by the program RCOMP using a -s*xxx* option, the processed data can be compared to the original data by using the same -s*xxx* option for DCOMP rather than specifying the original input file. The printed output is a summary of the statistical comparison.

Syntax : DCOMP [<options>] [<input file name> [<2nd file name>]]

>options> are each preceded by a hyphen (-) and separated by spaces:

-nx specifies the number of symbols in the data stream. A binary stream is the default ($x = 2$).

-sxxx specifies the starting 'seed' for the random number generator. This must be the same as the original seed used to generate the original data using RDATA. The defaults is 0.

-ix specifies the number of initial input samples to skip at the start of the input file(s).

-ox specifies the positive or negative sample offset (shift) of the second data stream relative to the first.

-p specifies that output should also be sent to the printer.

<input file name> is the name of the input file to be compared with. This file sets the number of symbols. If not specified, the input defaults to 'stdin'.

<2nd file name> is the name of the second input file to be compared with the first.

The **-s**xxx option was implemented to remove the need to store the original data file used for the simulation run. This is achieved by using the same pseudo-random number sequence to generate the data internally. In this implementation, the built in random number function of Turbo-C was used.

Appendix D

Details of Configuration File

D.1 Description of Configuration File Entries

The ENCODER and DECODER program modules use a configuration file to define the encoder design completely. The only assumption made about the encoder is that is has the general structure of figure(4.6). The configuration file is a text file that contains information about the number of bits used for each section of the circuit, and defines the tables used to implement the logic functions. To facilitate human readability of the configuration file, the ENCODER and DECODER programs allow one or more whitespace characters to be placed between the components of the file. In addition, a comment delimiter is defined. All text on a line occurring after a semi-colon (;) is ignored by the programs. This allows meaningful comments to be placed in the configuration file for future reference.

The contents of the configuration file is an *ordered* set of numbers used to define the various parts of the circuit. The total number of elements in the configuration file is dependant on the actual code that is defined. The table does not contain more information than necessary. This means that the size of the tables in the configuration file is determined by the previous definitions.

The details of the various fields is defined in the ordered description list below :

Total number of input bits This field defines the size of the input symbol to the encoder. The values is the number of binary digits (bits) used to represent the

input symbols. If this filed value is k, then there are 2^k possible values for the input symbol.

Number of Differentially encoded bits This field defines how many bits are applied to the differential encoder. The value can legally be defined as 0 if there is no differential encoder.

Table for Differential Encoder This is a table whose contents define the differential encoder. The size of the table is defined by the previous field as follows: If q bits are differentially encoded then there are 2^{2q} elements in the table. The table is indexed by combining the current input bits with the previous output bits of the differential encoder. The previous output forms the most significant part of the index. The contents of the table is the output of the differential encoder. Note that if no bits are differentially encoded there must still be a single (zero) entry in the table. This is consistent with the definition of the table size since $2^{2 \cdot 0} = 1$.

Number of Memory bits in Convolutional Encoder This field defines the constraint length of the convolutional encoder. If the value of this field is k, then there are 2^k states in the convolutional encoder. This field can legally be defined as 0.

Number of input bits to Convolutional Encoder This defines the number of bits that go into the convolutional encoder that affect the next state of the encoder. This value can legally be defined as 0 if there is no convolutional encoder implemented. This field will be referred to as l shortly. This field combined with the previous field define the length of the table used to define the convolutional encoder. The length of the table is calculated as length $= 2^{k+l}$.

Number of output bits from the Convolutional Encoder The value in this field is generally one more the the value in the previous field (since there is normally only one redundant bit).

Table for Convolutional Encoder This is a table of 2^{k+l} records. Each record contains two fields, namely the *next state* of the convolutional encoder and the *output data* from the convolutional encoder. The table is indexed by combining the current state of the convolutional encoder with the current input to the convolutional encoder. The current input to the convolutional encoder is the most significant part of the index.

Number of Signal Elements This defines the number of entries for the signal mapping table. This value is in fact a check only, since the value can be derived from the previous entries of the configuration file.

Number of dimensions of the Signal This specifies the number of dimensions of the channel signal. This figure is based on the matched filter signal representation of the channel signal. This value is the number of correlation receivers are required to determine the channel signal. For a QAM type of signal, this value will always be 2.

Table of Channel Signal Mapping This table defines the signal space of the channel as represented by the sampled matched filter signal representation. For a N-dimensional signal there are N entries for each of the signal elements. For a QAM type of signal, the entries are the **I** and **Q** numbers of each of the signal mappings from the constellation diagram.

Truncation Depth This entry specifies the truncation depth to be applied by the Viterbi algorithm in the DECODER module. The minimum value is 2. The

maximum value is determined by the version of the DECODER module. For the implementation presented in this dissertation, the maximum value is 50.

The text of the configuration files used in the simulations follows.

D.2 Configuration file for V.32 TCM

```
;   File = "DECODER.CFG" :
;  Configuration file for V.32 recommendation TCM
;  using a 32-CR signal constellation.

4       ; Total number of input bits (0-15)

2       ; Number of input (& output) bits of differential encoder
        ; If n is the number of input bits,
        ; the differential encoder will
        ; have (2^n)^2 entries.
        ; Assuming n=2, format of table must be
        ;           D3      D2      D1      D0
        ; Index :   Y2n-1   Y1n-1   Q2      Q1 (binary)
        ; Value :                   Y2n     Y1n
0  1  2  3
1  2  3  0
2  3  0  1
3  0  1  2

3       ; Number of bits in state machine (2^3 = 8 states)
2       ; Number of bits that control the state machine (Jn-1 ..  J0)
3       ; Number of bits output from state machine (Qn-1 ..  Q0)
        ; The state machine table follows.  The index is generated
        ; from the current state and the input bits.  The output is
        ; a 2 valued array that contains the next state, and
        ; the output bit(s) of the encoder.
        ;           D4      D3      D2      D1      D0
        ; Index :   J1      J0      S2n-1   S1n-1   S0n-1
        ; INDEX
0  0    ; [00 000]
4  1    ; [00 001]
1  0    ; [00 010]
7  1    ; [00 011]
2  0    ; [00 100]
6  1    ; [00 101]
3  0    ; [00 110]
5  1    ; [00 111]
2  2    ; [01 000]
7  3    ; [01 001]
3  2    ; [01 010]
4  3    ; [01 011]
0  2    ; [01 100]
5  3    ; [01 101]
```

```
1  2    ; [01 110]
6  3    ; [01 111]
3  4    ; [10 000]
5  5    ; [10 001]
2  4    ; [10 010]
6  5    ; [10 011]
1  4    ; [10 100]
7  5    ; [10 101]
0  4    ; [10 110]
4  5    ; [10 111]
1  6    ; [11 000]
6  7    ; [11 001]
0  6    ; [11 010]
5  7    ; [11 011]
3  6    ; [11 100]
4  7    ; [11 101]
2  6    ; [11 110]
7  7    ; [11 111]

; End of convolutional state table

32       ; Number of mapping elements (This is a check - can
         ; be derived from previous info).
2        ; Number of dimensions of the signal
; The index into the array is the uncoded bits (Most significant
; part) plus the differentially encoded bits plus the convolutional
; encoder bits(s)

-4.0  1.0   -3.0 -2.0   -2.0  3.0    1.0  4.0
 4.0 -1.0    3.0  2.0    2.0 -3.0   -1.0 -4.0
 0.0  1.0   -3.0  2.0    2.0  3.0    1.0  0.0
 0.0 -1.0    3.0 -2.0   -2.0 -3.0   -1.0  0.0
 0.0 -3.0    1.0 -2.0   -2.0 -1.0   -3.0  0.0
 0.0  3.0   -1.0  2.0    2.0  1.0    3.0  0.0
 4.0  1.0    1.0  2.0    2.0 -1.0    1.0 -4.0
-4.0 -1.0   -1.0 -2.0   -2.0  1.0   -1.0  4.0
;

40       ; Define the path length used in the implementation of
         ; the Viterbi algorithm.  (Maximum = 50)

; End of definition
```

D.3 Configuration file for uncoded 16-QAM

```
;   File = "16QAM.CFG" :
;  Configuration file for the 16-QAM code of the V.32
;  recommendation that does not use redundant coding.
;
4       ; Total number of input bits (0-15)

2       ; Number of input (& output) bits of differential encoder
        ; If n is the number of input bits, the differential encoder
        ; will have (2^n)^2 entries.
        ; Assuming n=2, format of table must be
        ;           D3     D2     D1     D0
        ; Index :   Y2n-1  Y1n-1  Q2     Q1 (binary)
        ; Value :                 Y2n    Y1n
2  3  0  1
0  2  1  3
3  1  2  0
1  0  3  2

0       ; Number of bits in state machine (2^0 = 1 state)
0       ; Number of bits that control the state machine (0)
0       ; Number of bits output from state machine (0)
        ; The state machine table follows.  The index is
        ; generated from the current state and the input
        ; bits.  The output is a 2 valued array that contains
        ; the next state, and the output bit(s) of the encoder.
        ;                  ...    D1     D0
        ; Index :   0      ...    0      0
        ; INDEX
0  0    ; [00 000]  There is no convolutional encoder
; End of convolutional state table

16      ; Number of mapping elements (This is a check -
        ; can be derived from previous info.
2       ; Number of dimensions of the signal
; The index into the array is the uncoded bits (Most
; significant part) plus the differentially encoded bits
; plus the convolutional encoder bits(s)

-1.0 -1.0   -1.0  1.0    1.0 -1.0    1.0  1.0
-1.0 -3.0   -3.0  1.0    3.0 -1.0    1.0  3.0
-3.0 -1.0   -1.0  3.0    1.0 -3.0    3.0  1.0
-3.0 -3.0   -3.0  3.0    3.0 -3.0    3.0  3.0
;
```

```
2       ; Define the path length used in the implementation
        ; of the Viterbi algorithm.  (Maximum = 50)
        ; 2 is the minimum

; End of definition
```

APPENDIX E

STATISTICS OF SIMULATION RESULTS

The information provided in this appendix is derived from [6].

In the context of this appendix, and *error event* means a decoding error of a symbol, as measured by the simulation. An error event here will always be a single error when applied to uncoded modulation such as 16-QAM modulation used in the V.32 recommendation.

The simulation is a test of a series of trials that can have only one of two possible outcomes :

- There is no decoding error. (No *error event*).
- there is a decoding error. (An *error event* occurred).

For a given value of SNR, the probability of an error event should remain constant.

In a simulation of N trials there will be e decoding errors. The remaining $N - e$ trials will not contain decoding errors. In the case of uncoded modulation, each error event is independent of any other error event. Trials which satisfy these conditions are known as *Bernoulli trials.* The probability distribution function for such trials is the **Binomial Distribution.** The probability of an error event is given by

$$P_e = \frac{e}{N}$$

and the probability that no error will occur is

$$(1 - P_e)$$

It can be shown [6, p70] that the mean value of the binomial distribution is given by

$$\mu = N \times P_e \tag{E.1}$$

The variance of the binomial distribution is given by

$$\sigma^2 = N \times P_e \times (1 - P_e) \tag{E.2}$$

If $P_e \ll 1$, equation(E.2) can be accurately approximated by

$$\sigma^2 = N \times P_e \tag{E.3}$$

Equations E.1, E.2 and E.3 refer to mean and variance of the number of events, and not P_e. In other words μ is the average number of error events that can be expected.

From the simulations, μ is estimated as the value of e. An estimate of P_e is in turn derived from e by the equation

$$P_e' = \frac{e}{N} \tag{E.4}$$

Likewise, the variance σ^2 of μ is estimated as

$$s^2 = e \tag{E.5}$$

The estimate of the variance of P_e is then

$$s_{P_e}^2 = \frac{e}{N} \tag{E.6}$$

N is simply a scaling factor that is set by the length of the simulation run. For the standard deviation s to be within 10% of the mean estimate e simplifies as

$$\begin{aligned} \sqrt{s^2} &= 0.1e \\ \Rightarrow \quad 10\sqrt{e} &= e \\ \Rightarrow \quad e &= 100 \end{aligned} \tag{E.7}$$

Thus the criteria for the standard deviation to be within 10% of the magnitude of e is to simply continue the simulation until 100 error events have occurred.

The binomial distribution can be approximated by the normal distribution if $P_e \times N > 5$ and $(1 - P_e) \times N > 5$. This is the case for the simulations. Thus inferences concerning confidence intervals may be established using standard normal distribution tables.

For a 90% confidence interval, the standard deviation s must be scaled by α where $\mathrm{Erfc}(\alpha) = 0.05$. From tables of $\mathrm{Erfc}(\alpha)$ it is found $\alpha = 1.645$. Hence the 90% confidence level for the answer to be within 10% of the true result requires that

$$1.645\sqrt{e} = 0.1e$$

In other words, for the answers to be within 10% of the true answer, with a 90% confidence level, requires that the simulation be continued until

$$\begin{aligned} \sqrt{e} &= 1.645 \cdot 10 \\ \text{or } e &= 271 \end{aligned} \tag{E.8}$$

The band of P_e for a 90% confidence level is given by

$$\Delta P_e = \frac{1.645\sqrt{e}}{N} \tag{E.9}$$

BIBLIOGRAPHY

[1] Ezio Biglieri, Darriush Divsalar, Peter J. McLane, and Marvin K. Simon. *Introduction to Trellis-Coded Modulation with Applications.* Macmillan, New York, 1991.

[2] CCITT. *A familiy of 2-wire, duplex modems operating at data signalling rates of up to 9600 bits/s for use on the general switched telephone network and on leased telephone-type circuits.* Recommendation V.32, Spain, 1984. Málaga-Torremolinos.

[3] Durham Technical Images. *PC Data Master Basic Distribution Manual*, 1988. Version 2.0.

[4] Forney G. D., Jr. The Viterbi algorithm. *IEEE Proc*, 61:268–278, 1973.

[5] Simon Haykin. *Digital Communications.* John Wiley & Sons, Inc, New York, 1988.

[6] Irwin Miller and John E. Freund. *Probability and Statistics for Engineers.* Prentice-Hall, New Jersey, third edition, 1985.

[7] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, Cambridge, 1988.

[8] John G. Proakis. *Digital Communications.* McGraw-Hill, January 1989.

[9] Ferrel G. Stremler. *Introduction to Communcation Systems.* Addison-Wesley, 1982.

[10] Gottfried Ungerboeck. Channel coding with multilevel/phase signals. *IEEE Transactions on Information Theory*, IT-28, January 1982.

[11] Lee-Fang Wei. Rotationally invariant convolutional channel coding with expanded signal space–Part I: 180 degrees. *IEEE Journal on Selected Areas in Communications*, SAC-2, September 1984.

[12] Lee-Fang Wei. Rotationally invariant convolutional channel coding with expanded signal space–Part II: Nonlinear codes. *IEEE Journal on Selected Areas in Communications*, SAC-2, September 1984.

[13] Z. C. Zhu and A. P. Clark. Rotationally invariant coded PSK signals. *IEEE Proc*, 134:43–52, February 1987. Part F.