ON THE COMPUTATION OF THE PROBABILITY OF UNDETECTED ERROR FOR LINEAR BLOCK CODES ON THE GILBERT CHANNEL

by

BRENDEN WONG

B. Sc. University of British Columbia

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
DEPARTMENT OF ELECTRICAL ENGINEERING

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
September, 1991

Department of Electrical Engineering

The University of British Columbia
Vancouver, Canada

Date Oct. 15, 1991

Abstract

An important measure of the performance of error detecting codes is the probability of undetected error. Extensive study on the subject has yielded results which allow for the computation of the probability of undetected error for many codes on the binary symmetric channel (BSC). However, little is known about code performance in more complicated channel models. The Gilbert channel is a two-state, three-parameter model with memory which simulates the effects of burst noise. In this thesis, we investigate methods to compute the probability of undetected error of binary linear block codes on this channel.

We examine an approach to approximate code performance based on the P(m,n) distribution which is the probability of m errors in a block of n bits and the weight distribution of the code. For the Gilbert channel, P(m,n) can in principle be calculated from the channel parameters. In practice however, existing methodologies suffer from rather excessive computational requirements, particularly when n is larger than one thousand or so. We have developed an efficient method to calculate P(m,n) for reasonable channel parameters. This allows the probability of undetected error for many codes to be readily estimated.

For certain channel and code parameters, the approximation method described above may not be sufficiently accurate. Exact analytical results are difficult to obtain, however; because unlike the BSC, the probability of a particular error pattern on the Gilbert channel depends not just on the number of *1*'s in the pattern. Nevertheless, by appropriately exploiting certain symmetries present on the Gilbert channel, we can acquire some useful results. We have derived the probability of undetected error for the single parity check code. We have also obtained a formula for summing over a cyclic class of vectors and shown that reciprocal generator polynomials generate cyclic codes which have the same probability of undetected error on the Gilbert channel.

The Monte Carlo simulation technique is often used when exact analysis is difficult. In a simulation study of CRC codes, we are able to observe several interesting qualitative results with just a reasonable amount of computational effort. We find that as on the BSC, on the Gilbert channel the probability of undetected error does not always increase with worsening channel conditions. Also, the CRC-CCITT code appears to maintain its superiority in terms of error detection performance over the CRC-ANSI code on the Gilbert channel, and perhaps most significantly, for some ranges of channel parameters, the probability of undetected error estimated using BSC results with the effective bit error rate can be quite inaccurate.

Contents

List of Tables

List of Figures

Acknowledgement

First, I must thank Prof. C. Leung for his very excellent supervision throughout the course of this work and also for his help in obtaining financial support.

I must also thank my peers from whom I have benefitted greatly by association. Space restrictions prevent me from naming them all so I should just mention that several discussions with C. Lai and V. Wong on the subject on this thesis have been most helpful.

Finally, I gratefully acknowledge the financial support of NSERC through Grant #OGP0001731 and The Department of Electrical Engineering which were in the form of partial research assistantship and partial scholarship respectively.

1. Introduction

The ability to exchange or retrieve information accurately and efficiently is of paramount importance in everyday life. The speed and accessibility requirements of today have led to complex communications architectures, enough so that it is customary to consider a layered structure where each layer serves as a virtual communications link. At the most basic level (the "physical layer"), the link is physically real and consists of a data source, an encoder, a channel over which the data is transferred, a decoder, and the final output as the receiver's interpretation of the original message. Clearly, correct and efficient operation at this level is essential and ultimately related to the performance of a complex hierarchical system. The physical communications channel may be thought of as being described by the medium over which data is transmitted or stored and the method by which such data is represented (modulation). The problem is that in general, due to various imperfections, the channel distorts the original message. It is the purpose of coding to add redundancy in a well defined manner to the source data in order that discrepancies from the original message may be detected and/or corrected by the receiver.

Coding is a mathematical scheme which maps each possible message consisting of a sequence of symbols into a distinct sequence of symbols known as a codeword [1-3]. In practice, the sets of message and codeword symbols, that is, the message and codeword alphabets, are often identical. Naturally, the most common alphabets used for digital communications consist of powers of two symbols so the binary alphabet $\{0,1\}$ is one of particular significance. In a block code, the number of symbols in a message or codeword is fixed. If the messages are of length k and the alphabet is of size s, there are s^k k-tuples representing all possible messages. The k-tuples spanning the space V_k are mapped one to one into the space of n-tuples V_n. A linear mapping (with respect to addition and multiplication in the finite field GF(s)) is particularly simple to visualize and implement as it can be accomplished by matrix multiplication. A linear block code C(n,k) is then defined to be a k-dimensional linear subspace of V_n. For a message m (k-dimensional row vector), the corresponding codeword c (n-dimensional row vector) is given by c=mG where G ($k \times n$ matrix with rank k) is known as the generator matrix. The extra n-k symbols are called parity symbols. It is often convenient to represent a codeword by a polynomial. That is, we identify the coordinate representation $(c_0,c_1,\ldots,c_{n-1})$ of a codeword c with the polynomial $c(x)=c_0+c_1x+\ldots+c_{n-1}x^{n-1}$. One particular class of linear block code which has been much studied and often used in practice is the class of cyclic codes. A cyclic code has the property that if a given vector is a codeword, then so are all vectors which arise from cyclic permutations of this vector. A cyclic code can be characterized by a generating polynomial g(x) of degree n-k. The codewords of a cyclic code are then given by the set of all polynomials {g(x)d(x)} where d(x) is a polynomial with degree at most k-1.

Let $c \in C(n,k)$ be a transmitted codeword. The received vector $v \in V_n$ may be written as v=c+e where $e \in V_n$ is the error vector representing the distortions or noise introduced by the channel. Hence, the ith component of e is a *0* or *1* depending on whether the ith bit transmission is error free or in error respectively. In general, $e \neq 0$ and $v \notin C(n,k)$; this being the case the receiver has several recourses. If the code is used for error detection, v is discarded and a retransmission is requested (the procedure by which this occurs is the function of the second layer in the communications hierarchy, the "data link layer", with which we will not be concerned here). It is also possible that the received vector $v \in C(n,k)$ but $v \neq c$ in which case the code has failed to detect the error. The probability of undetected error is then a measure of reliability of an error-detecting code. If the code is used for error correction, an estimate $c' \in C(n,k)$ is found based on v and the maximum likelihood principle. An error in decoding results if the estimate is incorrect, that is if $c' \neq c$. Thus the probability of decoding error is a measure of the ability of an error-correcting code to identify correctly a codeword that is corrupted by noise. A code can also be used for simultaneous error detection and correction, that is to decode received vectors which are reasonably similar to corresponding codewords and reject those which are too dissimilar. In general, error correction is much more difficult to analyze and implement than error detection. In the following we consider only binary linear block codes used for error detection.

One of the simplest noise models is the binary symmetric channel (BSC). In this channel, two possible symbols, *0* and *1*, or bits in this case, are transmitted and received. There is a probability ε that a transmitted bit will be received incorrectly. Successive errors are independent. The bit error rate ε is assumed to be constant and representative of the overall channel conditions. For a linear code, the probability of undetected error is independent of the particular codeword transmitted and is just the probability that the error vector is itself a codeword. The Hamming weight (or simply weight) of a vector v, denoted wt(v), is the number of *1*'s that it contains. If the number of codewords with Hamming weight i is A_i (i=0,...,n), the Hamming weight enumerator is the polynomial defined by

$$A(z) = \sum_{i=0}^{n} A_i z^i \ . \tag{1.1}$$

For the BSC, all error vectors of weight i have probability $\varepsilon^i(1-\varepsilon)^{n-i}$ so that knowledge of the weight distribution of a code is sufficient for computing its probability of undectected error since

$$\begin{aligned} P_u &= \text{prob}(e \neq 0,\ e \in C(n,k)) \\ &= \sum_{i=1}^{n} A_i \varepsilon^i (1-\varepsilon)^{n-i} \\ &= (1-\varepsilon)^n \left[A(\varepsilon/(1-\varepsilon)) - 1\right] \ . \end{aligned} \tag{1.2}$$

There is a large body of literature describing many results and techniques used in the study of weight enumerators [1-3]. Regarding P_u itself, an important characteristic is whether it is monotonic with respect to ε. A code where P_u is monotonically increasing with ε is called proper. Properness and other aspects of P_u are discussed for various codes on the BSC in [4-11].

While the BSC is useful for describing channels afflicted by random or shot noise errors, its memoryless nature renders it inappropriate for channels where errors tend to occur in clusters or bursts. Burst noise effects are typically associated with multipath fading in radio channels, switching transients and crosstalk on wired channels, and surface defects and foreign particles on magnetic storage media. The Gilbert channel, illustrated in Fig. 1.1, is a model devised to simulate such burst effects [12]. The channel consists of two states, a "good" state G where transmission is completely error-free and a "bad" state B where the probability of correct transmission is h; a state transition from G to B occurs with probability P and from B to G with probability p. It is convenient to define Q=1-P, q=1-p, and h'=1-h so Q is the probability of remaining in state G,

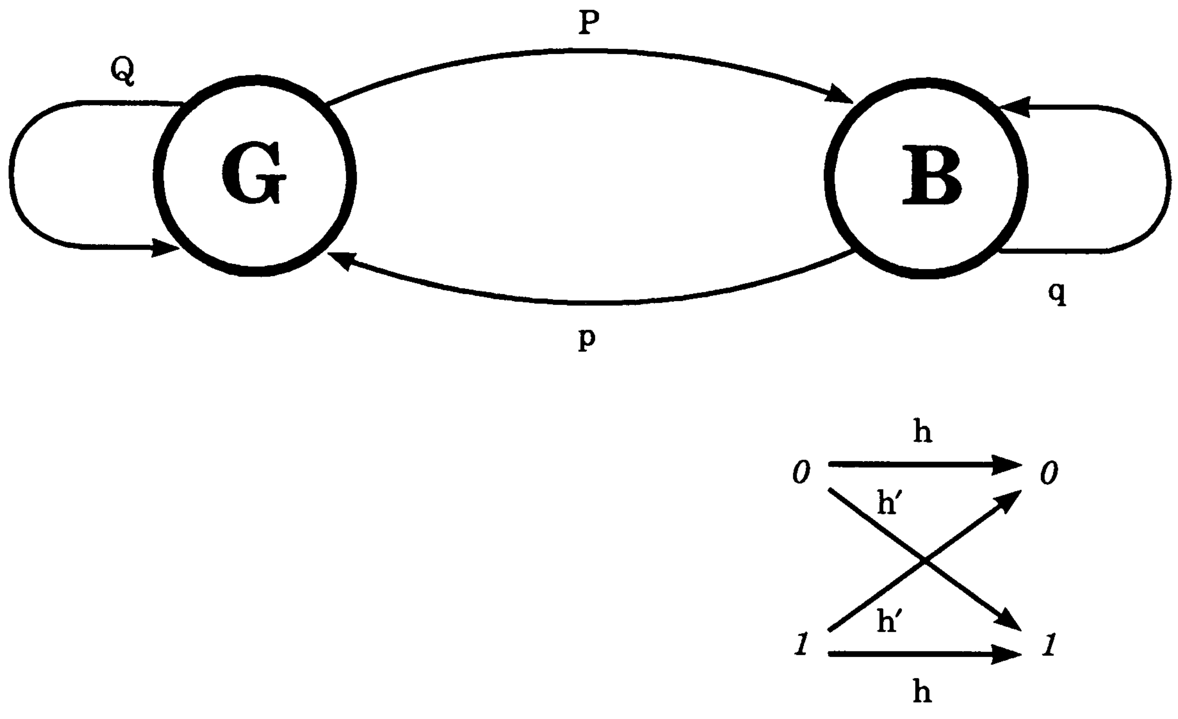


Figure 1.1 The Gilbert channel

q is the probability of remaining in state B, and h' is the bit error rate while in state B. As well, we denote the average or unconditional probability of the system being in state B by p_B=P/(P+p), the corresponding probability for state G by p_G=p/(P+p), and the effective bit error rate of the channel by $\varepsilon_{eff}=p_B h'$. For suitable (small) values of P and p, the model produces the type of burst error statistics that are of interest. The idea is that most of the time the channel is in the good state but occasionally, due to a change in the transmission characteristics, the channel lapses into the bad state which tends to persist. As an example, Gilbert found an adequate fit using h=0.84, P=0.003, p=0.034 for Call #1296 which was over a 500-mile radio path with loaded cables at the ends from the telephone circuit measurements of Alexander, Gryb, and Nast [13]. An extension to the Gilbert channel where G is not completely error-free but has a probability of correct transmission g (g>h) was considered by Elliott [14] and hence is known as the Gilbert-Elliott channel.

As the Gilbert channel is a system with memory, the probability of the next error vector bit being *0* or *1* depends on the current state of the channel and therefore on the past history of bits. However, this dependence can extend back only as far as when the last *1* was received since that pins down the system in the B state at that instant. Alternatively, the conditional independence can be expressed by $\text{prob}(e \mid \ldots 10^j) = \text{prob}(e \mid 10^j)$ where $e=0,1$ and the superscript j=0,1,... denotes the number of repetitions. If $w(j)=\text{prob}(0^j 1)$, $v(j)=\text{prob}(0^j 1 \mid 1)$, and $u(j)=\text{prob}(0^j \mid 1)$, then the probability of a particular error vector e can be decomposed [14] as

$$\text{prob}(e) = w(a) \left\{ \prod_{i=1}^{|e|-1} v(b_i) \right\} u(c) \tag{1.3}$$

where a is the number of *0*'s before the first *1*, b_i is the number of *0*'s between the ith and (i+1)th *1*, and c is the number of *0*'s following the last *1* in e. The above "building block" probabilities in the Gilbert channel can be drawn directly from the study of recurrent events in probability theory [15]. The results are

$$u(j) = \frac{1}{J-L}[(J+p-Q)J^j - (L+p-Q)L^j]$$
$$v(j) = \frac{h'}{J-L}[(qJ+p-Q)J^j - (qL+p-Q)L^j]$$
$$w(j) = p_B u(j) \tag{1.4}$$

where

$$J = \frac{1}{2}\{Q+hq+\sqrt{(Q+hq)^2+4h(p-Q)}\}$$
$$L = \frac{1}{2}\{Q+hq-\sqrt{(Q+hq)^2+4h(p-Q)}\} \tag{1.5}$$

are the roots of the quadratic $D(t)=t^2-(Q+hq)t-h(p-Q)$.

Generally, in order to find the probability of undetected error of a C(n,k) code in the Gilbert channel, we need to sum over 2^k-1 expressions of the form given in (1.3). Consider the computational effort required for such an exhaustive evaluation technique. To generate a codeword e (using matrix multiplication) requires 2nk additions and multiplications. Assuming that the $u(\cdot)$, $v(\cdot)$, and $w(\cdot)$ are precomputed, to compute prob(e) requires wt(e)+1 multiplications. Then overall, at least approximately $(1+2nk+n/2)2^k \approx nk2^{k+1}$ arithmetic operations are required to obtain P_u for each set of parameters P, p, h. We see that the number of such operations and therefore the computational complexity grows rapidly with k. For even a modest size code, say the Hamming (31,26) code, the actual computation using a Sun SPARC® station 1 workstation takes approximately 4 days. As a matter of comparison, the MATLAB® benchmark tests rate the SPARC® station 1 at 1.4 Mflops (millions floating point instructions per second) and the Cray XMP® supercomputer at 33 Mflops. The codewords themselves can be precomputed and stored to effect a small savings of factor 4k, but storing all codewords requires $2^k n$ bits of media space which for the above example of the Hamming (31,26) code is approximately 260 Mbyte. Clearly, determining P_u by exhaustive computation for all but the shortest codes is impossible even with the most powerful computing facilities that are currently available. The main difficulty is the exponential increase in the number of codewords with k. For the BSC, the situation is alleviated by the natural division of the codewords into n weight classes since all codewords with a given weight have the same probability and that is why the weight distributions of block codes have been studied extensively. For the Gilbert channel however, knowledge of a code's weight distribution is of no direct value; the general problem of determining P_u is much more difficult. Nevertheless, we have already alluded to the importance of coding and the fact that channel conditions are often not representable by the BSC alone so this problem deserves investigation. Our objective here will be to find viable means of computing P_u.

Two binary linear block codes are said to be equivalent to each other if the generator matrix of one can be obtained by a column (coordinate label) permutation of any generator matrix of the other [3]. Evidently, this definition is motivated by the fact that, on a BSC, equivalent codes have identical error detecting (and error correcting) properties since they have exactly the same weight enumerators. In general, the average probability of undetected error $E[P_u(C)]$ over all equivalent codes of a given code C(n,k) depends only on the weights A_i and the probability P(m,n) of m errors in a block of n bits. $E[P_u]$ may be considered an approximation to $P_u(C)$ [14]. This approach appears feasible for the Gilbert channel as P(m,n) may be obtained from recursion relations given by Elliott [14,16] or a closed form expression given by Cuperman [17]. In practice, however, the computation is problematical, especially if n is larger than one thousand or so. We address these problems in Chapter 2 by developing a technique that computes P(m,n) efficiently for reasonable channel parameters. As well, we make some observations regarding the validity of the $E[P_u]$ approximation and the performance of some short cyclic codes of the BCH (Bose-Chaudhuri-Hocquenghem) type relative to their equivalents. There has also been an approach to approximating P_u using the burst length distribution Q(l,n) where l is the burst length defined as the number of bit positions from the first to last error inclusive although it does not appear to be applicable for many realistic channel conditions [18].

In general, it is difficult to find exact analytical expressions for P_u. In fact, no nontrivial result of this type is known. A single parity check code is one of the simplest linear block codes; an extra bit which is the modulo 2 sum of the message bits is appended to the original message to form a codeword. Chapter 3 consists primarily of the derivation of P_u for the single parity check code. We also give there an expression for the probability of a given vector and its cyclic permutations which may be useful in determining P_u for cyclic codes as well as an identity regarding the P_u of cyclic codes which have generator polynomials satisfying a particular relationship.

A commonly used technique for determining error rates in communications systems is the Monte Carlo Simulation [19]. Random noise effects are computer generated according to some fixed distribution corresponding to system parameters and error events are tabulated. The average number of such events over a large number of samples can then be taken as an estimate of the actual physical error rate. As an application of the Monte Carlo technique, we study the performance of two CRC (Cyclic Redundancy Check) codes in the Gilbert Channel. A CRC codeword consists of the message bits plus a number of parity bits which corresponds to the remainder of a modulo 2 division of the message expressed as a polynomial and a predetermined polynomial known as the CRC generator polynomial. A judiciously chosen generator polynomial endows the CRC code with some desirable error detection characteristics. In particular, CRC codes are known for their strength in detecting burst errors. Hence, it is reasonable and useful to examine CRC codes under burst channel conditions in order to compare with and augment the conventional BSC results.

In Chapter 5, we give a summary of our findings and discuss topics which may deserve further study. Two appendices give some necessary details for the computations in Chapter 2. A third appendix contains the source code listings for the programs used in the numerical computations and simulations.

® Maple is a registered trademark of Waterloo Software Systems.
MATLAB is a registered trademark of The MathWorks, Inc.
Sun SPARC is a registered trademark of Sun Microsystems.
Cray XMP is a registered trademark of Cray Research, Inc.

2. Approximation By Averaging

2.1 Using P(m,n) to Approximate P_u

If P(m,n) is the probability of m *1*'s in an error vector of length n bits, then the average probability of a weight m error vector is $P(m,n)/c^n_m$ where $c^n_m = n!/[m!(n-m)!]$ is the binomial coefficient. Hence an approximation to the probability of undetected error of a C(n,k) code is given by

$$P_u \cong \sum_{m=1}^{n} \frac{A_m}{c^n_m} P(m,n) \ . \tag{2.1}$$

The idea is that even though codewords within the same weight class may in general have rather different probabilities, collectively these differences tend to cancel so if A_m is quite large, we can expect the total probability of all weight m codewords to be reasonably approximated by A_m times the average probability of a weight m vector.

Let π be a permutation of n indices. Denote the equivalent code resulting from the application of π to the vectors of C(n,k) by πC(n,k). Then as noted by Elliott [14],

$$\begin{aligned}
E[P_u] &= \frac{1}{n!} \sum_{\pi} P_u(\pi C) \\
&= \frac{1}{n!} \sum_{\pi} \sum_{x \neq 0\, x \in \pi C} \mathrm{prob}(x) \\
&= \frac{1}{n!} \sum_{x \neq 0\, x \in C} \sum_{\pi} \mathrm{prob}(\pi x) \\
&= \frac{1}{n!} \sum_{x \neq 0\, x \in C} [\mathrm{wt}(x)]![n-\mathrm{wt}(x)]!P(\mathrm{wt}(x),n) \\
&= \sum_{i=1}^{n} (A_i/c_i^n)P(i,n) \qquad (2.2)
\end{aligned}$$

so the approximation for P_u in (2.1) is in fact also the exact average over equivalent codes.

Unfortunately, there is no a priori way of determining the effectiveness of the approximation (2.1). We can obtain some empirical results only by experimentation. However, because of the connection set forth by (2.2), at least we also get information regarding the relative performance of codes as compared to their equivalents. That is, if the exact P_u for a code were known, comparison with (2.1) or (2.2) in the first context gives a measure of the accuracy of the approximation and in the second context a measure of how well the code performs in error detection relative to its equivalent codes. From a practical viewpoint though, we must first be able to compute P(m,n) efficiently. The P(m,n) distribution, also known as the counting distribution, has been studied for a variety of channel models [20] and is often parametrized by experimentally measured statistical quantities. For the Gilbert channel however, it can be expressed directly in terms of the fundamental channel parameters.

2.2 A Series Expansion for P(m,n)

A renewal process is a process where the occurrence of an event resets the state of the system, or in our present terminology, one where an error determines the state of the channel. The P(m,n) distribution for a renewal process is given quite generally by the recursion relations [16]

$$P(0,n) = 1 - \sum_{i=1}^{n-1} w(i)$$
$$P(m,n) = \sum_{i=1}^{n-m+1} p_1 u(i-1) R(m,n-i+1) \; ; \; 1 \leq m \leq n \tag{2.3}$$

where

$$R(1,n) = u(n-1)$$
$$R(m,n) = \sum_{i=1}^{n-m+1} v(i-1) R(m-1,n-i) \; ; \; 2 \leq m \leq n \; , \tag{2.4}$$

$u(\cdot)$, $v(\cdot)$, and $w(\cdot)$ have the same meaning as the definitions given before (1.4), and p_1 is the unconditional bit error rate (ε_{eff} for the Gilbert channel). Computing P(m,n) this way requires a number of summations increasing as n^4. The main difficulty however is the amount of memory required to store R(m,n) in a typical computational algorithm. For instance, a 1000×1000 array of numbers of type double in the C programming language (providing approximately 15 decimal place accuracy) requires about 8 Mbyte. Hence, in practice it would be awkward to handle n much larger than 1000 or so. Moreover, it is difficult to infer directly any relationship between P(m,n) and the channel parameters using this recursive technique. This would be a disadvantage, for instance, in the reverse problem where channel parameters must be extracted from experimental distributions for then an analytic form is almost certainly preferred.

A set of recursion relations for computing P(m,n) specific to the Gilbert channel requiring order n^2 summations is given in [14]. See (3.3), (3.4). However, the memory requirement is double that of the general recursive technique. Also, for n small, the computational time is not much different and is in fact inferior to that of the method which we will propose. Hence we do not discuss the latter recursion method further in this chapter.

A nonrecursive approach to computing P(m,n) for the Gilbert channel was given by Cuperman [17]. If $P_0(m,n)$ is the probability that the system is in the B state m times out of n, then P(m,n) is just

$$P(m,n) = (1-h)^m \sum_{i=m}^{n} c_m^i h^{i-m} P_0(i,n) \tag{2.5}$$

It can be shown that

$$P_0(m,n) = \begin{cases} \dfrac{p}{P+p} Q^{n-1} & ;\ m=0 \\ \dfrac{P}{P+p} \sum\limits_{j=0}^{m-1} \sum\limits_{t=0}^{2} c_j^{m-1} c_{n-m-j-t}^{n-j-t} c_t^2\, q^{m-j-1} Q^{n-m-j-t} (P-q)^{j+t} & ;\ 1 \le m \le n-1 \\ \dfrac{P}{P+p} q^{n-1} & ;\ m=n\ . \end{cases} \tag{2.6}$$

See Appendix A for details concerning the derivation of (2.6). The second equation has been written in a slightly more compact form than the corresponding equation in [17]. However, note in particular that its range of validity is actually $1 \le m \le n-1$ rather than $1 \le m \le n/2$ as stated in [17]. On the surface, (2.5) and (2.6) appear to provide quite an efficient means of computing

P(m,n) with order n^3 summations. This is not so however as the evaluation of $P_0(m,n)$ is actually quite problematical. The difficulty is in the amount of numerical accuracy required. The data in Table 2.1 illustrates the point. We have tabulated there each of the m terms, s(j) for j=0,...,m-1, in the summation for $P_0(m,n)$, calculated using double precision C language routines. The state transition probabilities are fixed at P=.001, p=0.1 and m=20 is taken. Note that for ease of reading, only 7 significant digits are displayed; the actual computations in C are performed to the full 15 digits accuracy. The sums of s(j) over j are compared with the results obtained using the infinite precision computation of Maple®. For n=30, agreement is quite precise. For n=40, the deviation is slight. For n=50, however, the C program generated result is a nonsensical one. The problem is that in general, each term s(j) in the summation for $P_0(m,n)$ is large (absolute value much greater than one) but such large numbers must add together to give a small (positive) number (much less than one). In the above case n=50, the largest s(j) are of the order 10^{13} while the actual $P_0(m,n)$ is of the order 10^{-4} so a posteriori, it is evident that at least 17 decimal digits accuracy is required for proper computation. Similarly, for the case n=40, we see that only 2 or so digits of accuracy can be expected with C routines. With n=500 say, hundreds of digits of accuracy are typically required. This order of numerical accuracy is supported by specialized computing packages such as Maple® though the drawback is extremely long execution time. Using a SPARC® station 1, the computation for a single $P_0(m,n)$ when $n \approx 500$ can take up to one hour.

Here we develop a method to circumvent the problems that are encountered by the previous methods. For most channel conditions which the Gilbert

P=0.001, p=0.1			
j	s(j)		
	n=30	n=40	n=50
0	1.924937e+04	5.373783e+07	1.270440e+10
1	-1.293079e+05	-5.322397e+08	-1.494887e+11
2	3.840047e+05	2.436174e+09	8.225323e+11
3	-6.627478e+05	-6.835843e+09	-2.811022e+12
4	7.347007e+05	1.315219e+10	6.684053e+12
5	-5.453716e+05	-1.838674e+10	-1.173890e+13
6	2.737843e+05	1.930997e+10	1.577618e+13
7	-9.148801e+04	-1.553482e+10	-1.658501e+13
8	1.940020e+04	9.675176e+09	1.382414e+13
9	-2.346285e+03	-4.683027e+09	-9.204888e+12
10	1.221875e+02	1.758559e+09	4.909311e+12
11	-0.000000e+00	-5.086999e+08	-2.093842e+12
12	0.000000e+00	1.119024e+08	7.099229e+11
13	-0.000000e+00	-1.835178e+07	-1.892875e+11
14	0.000000e+00	2.179734e+06	3.902036e+10
15	-0.000000e+00	-1.797807e+05	-6.061153e+09
16	0.000000e+00	9.669907e+03	6.820174e+08
17	-0.000000e+00	-3.070471e+02	-5.213714e+07
18	0.000000e+00	4.822956e+00	2.405282e+06
19	-0.000000e+00	-2.511794e-02	-5.013373e+04
$\sum$ s(j)	3.934083e-04	5.258999e-04	-2.910662e-03
Actual P(20,n)	3.934082e-04	5.302741e-04	6.672299e-04

Table 2.1 Computation of $P_0(20,n)$ using (2.6) and double precision C language routines.

channel is designed to reasonably simulate, $P \lesssim 0.01$ and $P/p \lesssim 0.1$. That is, P is nominally a small parameter so we can express (2.6) as a power series in P. Then P(m,n) can be computed from (2.5) using an approximation for $P_0(m,n)$ that is its truncated power series, the number of terms taken corresponding to the accuracy desired. The point is that successive terms in the expansion get progressively smaller so that severe numerical precision is not required.

In order to derive a compact expression for the order r contribution to $P_0(m,n)$, it is simpler to start from an alternate form of (2.6). Since

$$\begin{aligned} P_0(m,n)(P,p) &= \text{prob(system in state B m times out of n)} \\ &= \text{prob(system in state G n-m times out of n)} \\ &= \text{prob(system in state B n-m times out of n with} \\ &\qquad \text{state transition probabilities reversed)} \\ &= P_0(n-m,n)(p,P) \ , \end{aligned} \tag{2.7}$$

the second line in (2.6) can be written

$$\begin{aligned} P_0(m,n) &= p_G \sum_{j=0}^{n-m-1} \sum_{t=0}^{2} c^{n-m-1}_{j} \, c^{n-t-j}_{m-t-j} \, c^{2}_{t} \, Q^{n-m-j-1} q^{m-t-j} (P-q)^{j+t} \\ &= p_G \, q^m \sum_{t=0}^{2} c^{2}_{t} (-1)^t \sum_{j=0}^{n-m-1} c^{n-m-1}_{j} \, c^{n-t-j}_{m-t-j} \, (-1)^j \\ &\quad \cdot \sum_{\alpha=0}^{j+t} \sum_{\beta=0}^{n-m-j-1} c^{j+t}_{\alpha} \, c^{n-m-j-1}_{\beta} \, (-P)^{\alpha+\beta} q^{-\alpha} \ . \end{aligned} \tag{2.8}$$

Denote the coefficient of the P^r term in (2.8) by $P_0(m,n;r)$, that is, write

$$P_0(m,n) = p_G \sum_{r=0}^{n-m+1} P_0(m,n;r) \, P^r \ , \tag{2.9}$$

then

$$P_0(m,n;r) = (-1)^r q^{m-r} \sum_{t=0}^{2} c_t^2 (-1)^t \sum_{j=0}^{n-m-1} c_j^{n-m-1} c_{m-t-j}^{n-t-j} (-1)^j \sum_{\beta=0}^{r} c_{r-\beta}^{j+t} c_\beta^{n-m-j-1} q^\beta$$

$$= (-1)^r q^{m-r} \sum_{\beta=0}^{r} q^\beta \sum_{t=0}^{2} c_t^2 (-1)^t \left\{ \sum_{j=0}^{n-m-1} (-1)^j c_j^{n-m-1} c_{m-l-j}^{n-t-j} c_{r-\beta}^{j+t} c_\beta^{n-m-j-1} \right\} \tag{2.10}$$

The last expression can be simplified using the results in Appendix B. For t=2, the factor in braces is

$$\sum_{j=0}^{n-m-1} (-1)^j c_j^{n-m-1} c_{m-2-j}^{n-2-j} c_{r-\beta}^{j+2} c_\beta^{n-m-j-1}$$

$$= \sum_{j=0}^{n-m-1} (-1)^j \frac{(n-2-j)!}{(m-2-j)!(n-m)} \frac{(j+2)(j+1)}{(r-\beta)![j+2-(r-\beta)]!} \frac{1}{\beta!(n-m-j-1-\beta)!}$$

$$= \frac{1}{(n-m)\beta!(r-\beta)!} \sum_{j=0}^{n-m-1} (-1)^j \frac{(n-j-2)!}{[n-(n-m)-2-j]![j-(r-\beta-2)]![(n-m)-\beta-j-1]!}$$

$$\bullet \{[j-(r-\beta-2)][j-(r-\beta-1)] + 2(r-\beta)[j-(r-\beta-2)] + (r-\beta)(r-\beta-1)\}$$

$$= \frac{1}{(n-m)\beta!(r-\beta)!} \{U(n,n-m,2,r-\beta,\beta) + 2(r-\beta)U(n,n-m,2,r-\beta-1,\beta) + (r-\beta)(r-\beta-1)U(n,n-m,2,r-\beta-2,\beta)\} \ . \tag{2.11}$$

where the function $U(\cdot,\cdot,\cdot,\cdot,\cdot)$ is as defined in (B.1). Similarly, for t=1 and t=0, we have

$$\sum_{j=0}^{n-m-1} (-1)^j c_j^{n-m-1} c_{m-1-j}^{n-1-j} c_{r-\beta}^{j+1} c_\beta^{n-m-j-1}$$

$$= \frac{1}{(n-m)\beta!(r-\beta)!} \{U(n+1,n-m,2,r-\beta,\beta) + (r-\beta)U(n+1,n-m,2,r-\beta-1,\beta)\} \ . \tag{2.12}$$

and

$$\sum_{j=0}^{n-m-1} (-1)^j c^{n-m-1}_{j} c^{n-j}_{m-j} c^{j}_{r-\beta} c^{n-m-j-1}_{\beta}$$
$$= \frac{1}{(n-m)\beta!(r-\beta)!} \{U(n+2,n-m,2,r-\beta,\beta)\} \tag{2.13}$$

respectively. Hence, summing over t in (2.10) gives

$$\begin{aligned} P_0(m,n;r) = (-1)^r q^{m-r} \sum_{\beta=0}^{r} q^\beta \frac{1}{(n-m)\beta!(r-\beta)!} & \\ \bullet \{[U(n+2,n-m,2,r-\beta,\beta) - 2U(n+1,n-m,2,r-\beta,\beta) & \\ + U(n,n-m,2,r-\beta,\beta)] & \\ + 2(r-\beta)[-U(n+1,n-m,2,r-\beta-1,\beta) + U(n,n-m,2,r-\beta-1,\beta)] & \\ + (r-\beta)(r-\beta-1)U(n,n-m,2,r-\beta-2,\beta)\} \ . & \end{aligned} \tag{2.14}$$

The fourth argument of the U(·,·,·,·,·) function must be nonnegative in order to use the results of Appendix B so the summand in the last equation has to be treated in three parts. Using (B.5), (B.10), (B.11), (2.14) becomes after some algebra

$$\begin{aligned} P_0(m,n;r) = (-1)^r q^{m-r} \frac{1}{(n-m)} & \\ \bullet \left\{ \sum_{\beta=0}^{r-2} (-q)^\beta \frac{(n-m-\beta+1)(n-m-\beta)}{(r-\beta)!\beta!} V(n+\beta,n-m,2,r-2) \right. & \\ + (-q)^{r-1} \frac{-r(n-m-r+2)}{(m+r-1)(r-1)!} V(n+r,n-m,2,r-1) & \\ \left. + (-q)^r \frac{r(r+1)}{m(m+r)r!} V(n+r+1,n-m,2,r) \right\} & \end{aligned} \tag{2.15}$$

for $r \geq 2$ where the function V(·,·,·,·) is defined in (B.2). For r=1, the same expression holds except that the sum should be ignored. For r=0, only the last term inside the braces is present. Thus

$$
\begin{aligned}
P_0(m,n;0) &= 0 \\
P_0(m,n;1) &= q^{m-1}\frac{1}{(n-m)m}\left\{(n-m+1)V(n+1,n-m,2,0) + \frac{2q}{m+1}V(n+2,n-m,2,1)\right\} \\
P_0(m,n;r) &= (-1)^r q^{m-r}\frac{1}{(n-m)} \\
&\quad \bullet \left\{\sum_{\beta=0}^{r} (-q)^\beta \frac{(n-m-\beta+1)(n-m-\beta)}{(r-\beta)!\beta!} V(n+\beta,n-m,2,r-2)\right\} \; ; \; r\geq 2 \; .
\end{aligned}
\tag{2.16}
$$

Evaluating $V(\cdot,\cdot,\cdot,\cdot)$ by means of (B.9), the three equations can in fact be rewritten as one. For consistency, we also write the first and third equations of (2.6) as series expansions. The final result is

$$
\begin{aligned}
P_0(0,n;r) &= (-1)^r\, c^{n-1}_{r} \\
P_0(m,n;r) &= q^{m-r}\frac{1}{n-m}\, c^{n-m}_{r-1} \sum_{\beta=0}^{r} \frac{(n-m-\beta+1)(n-m-\beta)}{m+\beta}\, c^{r}_{\beta} c^{m+\beta}_{r} (-q)^\beta \quad ; \; 1\leq m\leq n-1 \\
P_0(n,n;r) &= q^{n-1}\, p^{-1}\, \delta_{r,0} \; .
\end{aligned}
\tag{2.17}
$$

where $\delta_{r,0}$ equals 1 if r=0 and 0 otherwise.

To summarize, what we have accomplished thus far is to give a means of calculating $P_0(m,n)$ through the use of (2.9) and (2.17). However, the summation in r can be extended only as far as necessary depending on the precision required (and the available computational resources). That is, we can truncate (2.9) and take the partial sums as successively better approximations:

$$
P_0(m,n) \cong p_G \sum_{r=0}^{R} P_0(m,n;r)P^r \equiv P_0^{(R)}(m,n) \; . \tag{2.18}
$$

The subsequent computation of P(m,n), the quantity which we are ultimately interested in, is straightforward as (2.5) involves a relatively small (average n/2) number of positive terms so that great numerical accuracy is not required. The number of summations required when determining P(m,n) this way is of order n^2r^2. Clearly, if the $P_0(m,n)$ are known to within a certain percentage accuracy, P(m,n) computed from (2.5) can be no worse.

The efficiency of our technique is summarized in Table 2.2. The entries in the table represent the number of terms in the expansion required in order to compute $P_0(m,n)$ to within 0.1% of the exact value. As expected, the method works best for small P since P is the expansion parameter. Also, the number of terms required increases with p and n.· This dependence can be explained as follows. Keeping only terms up to order R in P corresponds approximately to discarding processes with more than R $G \rightarrow B$ state transitions. A flip-flopping of states involves the factor pP so when p is large, processes with relatively large numbers of $G \rightarrow B$ transitions can less likely be ignored. Similary, if n is large, there is more opportunity for the system to make a $G \rightarrow B$ transition.

Some remarks concerning how the table is compiled are in order. The ratios $|P_0(m,n;r+1)P^{r+1}|/|P_0^{(r)}(m,n)|$ are computed for all m for successive values of r. When for a particular value of r=R, the ratio becomes less than 0.1% for all m, the computation is stopped and R is entered into the table. We have checked extensively using the recursion techique (2.3) for $n \leq 511$ and selectively using the exact Maple® calculation (2.6) for $n > 511$ that indeed $\{|P_0(m,n)-P_0^{(R)}(m,n)|/P_0(m,n)\} \lesssim 0.1\%$. This justifies our choice of 0.1% both as a reasonable level of accuracy and as a "convergence" criterion. However, one caveat on the use of our method is pointed to by parenthesized entries in Table 2.2. For those entries, greater than 15 digits precision is required to compute the individual terms $P_0(m,n;r)$ for r close to R so that Maple® must be used.

n	P=.00001				
	p=.0001	p=.001	p=.01	p=.1	p=.3
7	1	1	1	1	1
15	1	1	1	1	1
31	1	1	1	1	1
63	1	1	1	1	2
127	2	2	2	2	2
255	2	2	2	2	2
511	2	2	2	2	3
1023	2	2	2	3	4
2047	2	2	2	4	6
4095	2	2	3	6	10

n	P=.0001			
	p=.001	p=.01	p=.1	p=.3
7	1	1	1	1
15	2	2	2	2
31	2	2	2	2
63	2	2	2	2
127	2	2	2	3
255	2	2	3	4
511	3	3	4	5
1023	3	3	5	8
2047	4	4	8	13
4095	5	6	(13)	(21)

n	P=.001			P=.01	
	p=.01	p=.1	p=.3	p=.1	p=.3
7	2	2	2	2	2
15	2	2	2	3	3
31	2	2	2	4	4
63	3	3	3	6	6
127	3	3	4	8	8
255	4	4	7	13	13
511	5	6	10	(22)	(22)
1023	7	10	17	(40)	(41)
2047	(11)	(17)	(30)	>50	>50
4095	(19)	(29)	>50	>50	>50

Table 2.2 Efficiency of series expansion technique for computing $P_0(m,n)$.

From (2.17), we can see that once again the problem is the appearance of large numbers with alternating signs in the sum. The dominant factor in the summand is $c^{m+\beta}_{r}$ which for large m will give rise to r+1 large numbers unless r is quite small. For example, with n=2047, m=1023, p=.01, and r=11, the summand ranges from 3.2×10^{28} to 1.4×10^{31} in absolute value. Note however that the problem may not arise for certain combinations of parameters. In particular, if P is small enough, $P^rP_0(m,n;r)$ may in fact be negligibly small for large r. Also, if the accuracy requirement is made somewhat less stringent, then fewer terms will be needed in the expansion so that the numerical problem can be delayed from occurring. Hence, the resummation of (2.6) leading to our present formalism has accomplished what we expected in that the terms in the expansion (2.9) do indeed become small rapidly for reasonable channel parameters although the numerical accuracy problem has not entirely disappeared.

We should also mention that since $p_G=(1+P/p)^{-1}$, one might think that a more consistent technique would also take into account the expansion of this factor in (2.8) and (2.9). In fact, we find that doing this results in no improvement, neither in accuracy nor efficiency. Rather, it just serves to complicate matters as the sum in (2.18) becomes a double sum, and the expansion corresponding to (2.9) becomes infinite.

2.3 Comparison with Other Methods of Computing P(m,n)

Cuperman gives the following approximations based on (2.6) together with their claimed ranges of validity [17],

$$P_0(m,n) \cong Pq^{m-1}Q^{n-2m}[(n-m-1)p+2] \;;\; mP \ll 1, P \ll p,q$$

$$P(m,n) \cong P(1-h)^m \sum_{i=m}^{n} h^{i-m}q^{i-1}[(n-i-1)p+2] \;;\; nP \ll 1, P \ll p,q \;. \qquad (2.19)$$

Both expressions are meant to be used for $1 \leq m \leq n/2$. The ad hoc manner in which (2.19) is derived does not allow for consistent interpretation or improvement. Table 2.3 gives a comparison of approximations for $P_0(m,n)$ using the first equation in (2.19) and our series expansion method with (2.9) and (2.17). For n=256, P=.0001, and p=.3, we take R=4 to give us the approximation $P_0^{(4)}(m,n)$ which is within 0.1% of the exact $P_0(m,n)$. As can be seen from the table, Cuperman's approximation deviates by as much as 28%. Even for m=50 so that mP=.005, the disagreement is still 19%. Generally, the second equation in (2.19) is even more inaccurate. With channel parameters P=.0001, p=.1, h=.7, even with a block length as small as n=16 (so nP=.0016), Cuperman's approximation is off by as much as 55% as Table 2.4 shows. There, $P^{(2)}(m,n)$ is the series expansion approximation calculated from (2.5) using $P_0^{(2)}(m,n)$ and is again within 0.1% of the exact value of P(m,n). Note that only two terms in the expansion for $P_0(m,n)$ are required to reach that level of accuracy.

The computation of P(m,n) by the recursion relations of (2.3) and (2.4) is in general reasonably efficient for $n \lesssim 1000$. We have already mentioned the

n=256, P=.0001, p=.3			
m	$P_0(m,n)$ (Cuperman)	$P_0^{(4)}(m,n)$	% difference
1	7.62e-03	7.62e-03	0.0
5	1.80e-03	1.84e-03	2.1
10	2.98e-04	3.12e-04	4.6
15	4.91e-05	5.27e-05	6.9
20	8.09e-06	8.89e-06	9.0
25	1.33e-06	1.50e-06	11.0
30	2.19e-07	2.52e-07	12.8
35	3.61e-08	4.22e-08	14.5
40	5.94e-09	7.08e-09	16.0
45	9.77e-10	1.18e-09	17.5
50	1.61e-10	1.98e-10	18.8
55	2.64e-11	3.30e-11	20.0
60	4.33e-12	5.49e-12	21.1
65	7.11e-13	9.13e-13	22.1
70	1.17e-13	1.51e-13	23.1
75	1.91e-14	2.51e-14	23.9
80	3.13e-15	4.15e-15	24.6
85	5.11e-16	6.85e-16	25.3
90	8.36e-17	1.13e-16	25.9
95	1.37e-17	1.86e-17	26.4
100	2.23e-18	3.05e-18	26.8
105	3.63e-19	4.99e-19	27.2
110	5.92e-20	8.16e-20	27.5
115	9.63e-21	1.33e-20	27.7
120	1.56e-21	2.17e-21	27.9
125	2.54e-22	3.52e-22	27.9
128	8.52e-23	1.18e-22	27.9

Table 2.3 Comparison with Cuperman's expression for $P_0(m,n)$.

n=16,	P=.0001,	p=.1	h=.7
m	P(m,n) (Cuperman)	$P^{(2)}(m,n)$	% difference
1	6.69e-04	6.72e-04	0.4
2	4.40e-04	4.52e-04	2.6
3	2.80e-04	3.05e-04	7.9
4	1.68e-04	2.02e-04	16.8
5	9.20e-05	1.27e-04	27.5
6	4.46e-05	7.21e-05	38.1
7	1.88e-05	3.56e-05	47.3
8	6.68e-06	1.48e-05	54.8

Table 2.4 Comparison with Cuperman's expression for P(m,n).

excessive memory requirement, however. Note that reducing the precision in order to conserve memory in this method is risky because of the large number of terms involved. Consider Table 2.5 which is a fairly representative example comparing the times required to compute all P(m,n) for n=1000 with P=0.0001, p=0.1, h=0.5 using the recursion and series expansion methods on a SPARC® station 2. When the series expansion computations can be handled by C language programs, the improvement in computational time is substantial. Even when Maple programming must be used, times to compute the P_0(m,n) are comparable to that of the recursion technique in computing P(m,n) (see remark below), especially when we take only r=5, which will nonetheless produce results to within 0.1% accuracy. An additional practical advantage in favor of the series expansion that is not reflected in the results in Table 2.5 is that once P_0(m,n) has been computed, only a little further computational effort is required to obtain P(m,n) using (2.5) for as many values of h as desired. With the recursion method, the entire computation must be repeated for each set of parmeters P, p, h.

In summary, by using the series expansion in P, we can approximate P(m,n) in a consistent and efficient manner where accuracy may be traded off for computational effort if necessary. For most channel parameters, only a few terms in the expansion are required in order to obtain good accuracy; typically such computations are at least an order of magnitude faster than when using the recursion method to obtain the exact result. For large values of n and P, efficiency suffers due to the requirement for greater numerical accuracy but at least excessive amounts of memory are not required.

n=1000	P=.0001, p=.1, h=.5
Method	Time to compute all $P(m,n)$
recursion	8 min.
series expansion (C program)	
r=5	10 sec.
r=10	15 sec.
r=20	33 sec.
series expansion (Maple program)	Time to compute all $P_0(m,n)$
r=5	3 min.
r=10	5 min.
r=20	45 min.

Table 2.5 Computational times of P(m,n) for the recursion and series expansion methods.

2.4 Some Observations on Short BCH Codes

For codes with $k \lesssim 20$, it is feasible to compute their P_u exhaustively by using the code generator matrices to generate all codewords and summing over their probabilities as given by (1.3). By comparing with the results from applying (2.1) or (2.2), we can study the validity of the approximation (2.1), and at the same time, the relative performance of a given code and its equivalent codes over the Gilbert channel.

We will consider in particular the (primitive, cyclic) BCH codes [1-3] with $n \leq 31$ and $k \leq 16$ as they are quite common in usage and have well known weight distributions. In addition, BCH codes are not unique in the sense described below; this property will also be interesting when considering P_u in the Gilbert channel. A BCH code of length 2^m-1 corresponds to a certain primitive polynomial of degree m. A primitive polynomial p(x) of degree m is an irreducible polynomial (one not divisible by any polynomial of degree less than m but greater than 0) such that the smallest integer n for which it divides x^n+1 is $n=2^m-1$. Since there may be more than one primitive polynomial of degree m, BCH codes with identical n and k are in general not identical. They do however have the same weight distributions and so on the BSC have the same error detection performance. Hence, it should be interesting to see whether a similar situation holds in the Gilbert channel. The problem is similar in nature to that of the relative performance of equivalent codes that we have already mentioned. However, in this case, there is one reduction that we can make immediately because of the time-reversal symmetry of the error bits produced in the Gilbert channel.

It turns out that primitive polynomials come in pairs. If p(x) is a primitive polynomial, then so too must be its reciprocal polynomial

$$p^*(x) = x^m \, p(1/x) \ . \tag{2.20}$$

(It is possible that $p^*(x)=p(x)$.) It follows that the generator polynomials defining the BCH codes of given (n,k) come in such reciprocal pairs [2]. Later in Chapter 3, we show that reciprocal generators give rise to cyclic codes with the same P_u on the Gilbert Channel. Hence, we need to be concerned with only half of the possible number of BCH codes.

We consider 7 BCH codes of different (n,k). The channel parameters taken are $p=\{0.01,0.1,0.3\}$, $h=\{0.5,0.7,0.9\}$ and $P=\{10^{-6},10^{-5.5},\ldots,10^{-2.5},10^{-2}\}$. Results are summarized below.

1. Hamming (7,4) code

A Hamming code is a special case of a BCH code with $k=2^m-m-1$. For m=3, there is only the primitive polynomial x^3+x+1 and its reciprocal. By the discussion above, it suffices to study just one of them in the Gilbert channel. The code appears well behaved with P_u monotonically increasing with P, decreasing with p and h. The actual P_u is always less than the approximate P_u, which we henceforth denote by $E[P_u]$, implying that the Hamming code is superior in error detection performance than the average of its equivalent codes. The percentage difference is only mildly sensitive to and decreases with P, but increases with p and decreases with h. For p=.3, h=.5, the difference is 18%. Generally, P_u rises linearly with P for $P \lesssim 10^{-3}$.

2. Hamming (15,11) code

For m=4, there is also just one primitive polynomial x^4+x+1 and its reciprocal. The qualitative behavior is the same as for the Hamming (7,4) code. The maximum difference between the exact and approximate P_u reaches a maximum of 51%.

3. BCH (15,7) code

Again, P_u is monotonic with increasing P and decreasing p and h. Generally, the exact P_u lies below $E[P_u]$ with a maximum difference of 140% when p=.3, h=.5. At h=.9, p=.01,.1 the difference is actually negative (i.e. the approximate P_u is larger than the exact one). However, the maximum difference is only -.5%.

4. BCH (15,5) code

The qualitative properties are very similar to those of the Hamming codes. The maximum difference between the exact and approximate P_u reaches 90% at p=.3, h=.5.

5. BCH (31,16) code

For m=5, there are three primitive polynomials: (i) x^5+x^2+1, (ii) $x^5+x^4+x^3+x^2+1$, and (iii) $x^5+x^4+x^2+x+1$ and their respective reciprocals. We will call the codes corresponding to these polynomials by the same numbers. Qualitatively, the P_u of the three different codes behave the same as that of the Hamming codes. The largest deviation between the exact P_u's and the approximate $E[P_u]$ occurs at p=.3, h=.5 and is about 950%. At h=.5, the differences between the three codes appear indistinguishable. The differences begin at emerge at low P_u, i.e., small P, large p, h, however; then code (i) has consistently lower P_u than code (ii) which in turn has consistently lower P_u than code (iii). At $P=10^{-6}$, p=.3, h=.9,

P_u(code (i))=4.0×10^{-15}, P_u(code(ii))=6.1×10^{-15}, and P_u(code(iii))=9.6×10^{-15}. The differences with the approximate P_u are 176%, 47%, and 18% espectively.

6. BCH (31,11) code

The P_u of these three codes behave similarly to that of the BCH (15,7) code. Maximum difference between exact and approximate P_u is 540% at p=.3, h=.5. At h=.9, p=.01,.1 the difference is negative with a maximum difference of only -1% for code (i), -.1% for code (ii), and -.2% for code (iii). There is a slight difference in P_u of 5% to 10% in regions of low P_u.

7. BCH (31,6) code

The three codes here have identical P_u. See Chapter 4, Part 3. Qualitatively, they are similar to the Hamming codes. The maximum difference between the exact P_u and E[P_u] is 400% at p=.3, h=.5.

We can make some general comments based on the observations above. All the BCH codes appear to have proper behaviour in that P_u increases monotonically with P and decreases monotonically with p and h. This is probably not surprising as the Hamming codes, double error correcting BCH codes like the BCH (15,7) code, the BCH (15,5) and triple error correcting codes with odd m such as the BCH (31,16) code are known to be proper [4,11]. The percentage difference between exact and approximate P_u is only mildly dependent on P and usually decreases with P. This is related to the linearity of P_u with respect to P as the exact and approximate P_u have almost parallel trajectories. The reason for this is that for small enough P, the pervasive p_B factor present in all probabilities (as given by (1.3) and (1.4)) contains the dominant linear contribution to P_u from P. The maximum difference between exact and approximate P_u always occurs at the largest p and the smallest h values we considered, namely at p=.3 and h=.5. The exact P_u is practically upper bounded by $E[P_u]$. When the exact P_u is observed to be greater, the difference is slight and always at low P_u. The differences between codes with the same (n,k) become most noticeable also at low P_u.

It should be mentioned that a similar study of some 35 codes with lengths less than 25 was originally reported by Elliott [14]. However, no names of codes or details were given other than the comment that in general P_u and $E[P_u]$ agreed within an order of magnitude except for extreme cases.

3. Some Exact Results

3.1 No Coding

For later use, we record here the probability of obtaining the zero error vector. Since the event not 0^n is the union of the mutually exclusive events $1, 01, \ldots, 0^{n-1}1$ [14],

$$\begin{aligned}
\text{prob}(0^n) &= 1 - \sum_{i=0}^{n-1} \text{prob}(0^i 1) \\
&= 1 - \sum_{i=0}^{n-1} w(i) \\
&= 1 - p_B(1-h) \sum_{i=0}^{n-1} \frac{(J+p-Q)J^i - (L+p-Q)L^i}{J-L} \\
&= 1 - \frac{p_B(1-h)}{J-L} \left\{ (J+p-Q)\frac{1-J^n}{1-J} - (L+p-Q)\frac{1-L^n}{1-L} \right\} . \qquad (3.1)
\end{aligned}$$

If no coding is used, any nonzero error vector results in an undetected error so the probability of undetected error is

$$P_u = \frac{p_B(1-h)}{J-L} \left\{ (J+p-Q)\frac{1-J^n}{1-J} - (L+p-Q)\frac{1-L^n}{1-L} \right\} . \qquad (3.2)$$

3.2 Single Parity Check Encoding

Let G(m,n) and B(m,n) be the probability of m errors in a block of length n given that the channel is initially in state G and state B respectively. The corresponding unconditioned probability is just the counting distribution of Chapter 2 and can be decomposed as

$$P(m,n) = p_G G(m,n) + p_B B(m,n) \ . \tag{3.3}$$

The conditioned probabilities may be found recursively from [14]

$$\begin{aligned}
&G(0,1) = 1\ , \quad G(1,1) = 0\ , \quad B(0,1) = h\ , \quad B(1,1) = h' \\
&G(m,n) = QG(m,n-1) + PB(m,n-1) \\
&B(m,n) = hqB(m,n-1) + h'qB(m-1,n-1) + hpG(m,n-1) + h'pG(m-1,n-1)
\end{aligned} \tag{3.4}$$

where of course G(m,n) and B(m,n) are zero if $m<0$ or $m>n$. Observe that the last equation gives a relationship between the probabilities of odd and even numbers of errors. By exploiting this fact, we will derive an analytic expression for the probability of undetected error for a single parity check code.

A codeword in a single parity check code of length n consists of n-1 message bits and a parity check bit which is *1* or *0* depending on whether the message contains an odd or even number of *1*'s. Hence a single parity check code consists of all possible even weight vectors and no odd weight vector. From the standpoint of probability of undetected error, it does not matter where the parity bit is inserted (or whether the original message bits are rearranged). Ultimately, what we are interested in is

$$P_u = \sum_{\substack{m\ \text{even} \\ m \neq 0}} P(m,n) \ . \tag{3.5}$$

Note that what we considered above is the even parity version of the single parity check code. For the odd parity version of the code, the parity check bit is instead the complement of that for the even parity case so the codewords are all the odd weight vectors. The odd parity version is not a linear block code since it does not contain a zero vector. However, since an odd/even weight vector added to an odd weight vector results in an even/odd weight vector, an undetected error results if and only if the error vector has even weight. That is, the odd parity and even parity codes have in fact the same performance as given by (3.5).

While we may think of using (2.6) in (3.5), it is immediately seen that the resulting summations are not easily tractable. The idea is to use (3.3) and (3.4) to effectively divide the summation into two parts and then solve for each by more elementary means. To this end, define

$$\begin{aligned} G_o(n) &= \sum_{m\ \text{odd}} G(m,n) \\ G_e(n) &= \sum_{m\ \text{even}} G(m,n) \\ B_o(n) &= \sum_{m\ \text{odd}} B(m,n) \\ B_e(n) &= \sum_{m\ \text{even}} B(m,n) \end{aligned} \tag{3.6}$$

where necessarily

$$\begin{aligned} G_o(n) + G_e(n) &= 1 \\ B_o(n) + B_e(n) &= 1\ . \end{aligned} \tag{3.7}$$

Summing over the recursion relations (3.4) gives

$$\begin{aligned} G_o(n) &= \sum_{m\ \text{odd}} QG(m,n-1)+PB(m,n-1) \\ &= QG_o(n-1) + PB_o(n-1) \end{aligned}$$

$$\begin{aligned}B_o(n) &= \sum_{m\ \text{odd}} hqB(m,n\text{-}1) + h'qB(m\text{-}1,n\text{-}1) + hpG(m,n\text{-}1) + h'pG(m\text{-}1,n\text{-}1)\\ &= hqB_o(n\text{-}1) + h'qB_e(n\text{-}1) + hpG_o(n\text{-}1) + h'pG_e(n\text{-}1)\\ &= (h\text{-}h')qB_o(n\text{-}1) + (h\text{-}h')pG_o(n\text{-}1) + h' \ . \qquad (3.8)\end{aligned}$$

The last equations do not quite allow us to solve for $B_o(n)$ and $G_o(n)$ because of the different arguments on the right hand side.

It is convenient at this point to introduce the formalism of generating functions. For a random variable over the nonnegative integers X(n), its generating function is given by the formal sum

$$x(t) = \sum_{n=0}^{\infty} X(n)t^n \ . \qquad (3.9)$$

For sufficiently well behaved X(n), x(t) will be well defined for some range of t. The individual probability X(n) is then the coefficient of t^n in a power series expansion of x(t).

Applying the definition (3.9), if $g_o(t)$ and $b_o(t)$ are the generating functions for $G_o(n)$ and $B_o(n)$ respectively, then

$$\begin{aligned}g_o(t) &= g_o(t) - G_o(1)t - G_o(0)\\ &= \sum_{n=2}^{\infty} G_o(n)t^n\\ &= Q \sum_{n=2}^{\infty} G_o(n\text{-}1)t^n + P \sum_{n=2}^{\infty} B_o(n\text{-}1)t^n\\ &= Qtg_o(t) + Ptb_o(t)\end{aligned}$$

$$\begin{aligned} \dot{b}_o(t) &= b_o(t) - B_o(1)t - B_o(0) + h't \\ &= \sum_{n=2}^{\infty} B_o(n)t^n + h't \\ &= (h-h')q \sum_{n=2}^{\infty} B_o(n-1)t^n + (h-h')p \sum_{n=2}^{\infty} G_o(n-1)t^n + h' \sum_{n=1}^{\infty} t^n \\ &= (h-h')qtb_o(t) + (h-h')ptg_o(t) + \frac{h't}{1-t} \ . \end{aligned} \tag{3.10}$$

Gathering similar terms gives

$$\begin{aligned} &(1-Qt)g_o(t) - Ptb_o(t) = 0 \\ &-(h-h')ptg_o(t) + [1-(h-h')qt]b_o(t) = \frac{h't}{1-t} \ . \end{aligned} \tag{3.11}$$

The solution of the simultaneous equations is

$$\begin{aligned} g_o(t) &= h'Pt^2 \frac{1}{1-t} \frac{1}{1-[(h-h')q+Q]t-(h-h')(1-q-Q)t^2} \\ &= \frac{h'Pt^2}{(1-t)(1-J_1t)(1-L_1t)} \end{aligned}$$

$$b_o(t) = \frac{1-Qt}{Pt} g_o(t) \tag{3.12}$$

where

$$\begin{aligned} J_1 &= J_1(Q,g,h) = J(Q,q,h-h') = J(Q,q,2h-1) \\ L_1 &= L_1(Q,g,h) = L(Q,q,h-h') = L(Q,q,2h-1) \ . \end{aligned} \tag{3.13}$$

Now if $P_o(n)$ is the unconditioned probability of obtaining an odd number of errors in a block of length n, its generating function is just

$$\begin{aligned} p_o(t) &= p_G g_o(t) + p_B b_o(t) \\ &= \frac{1}{P+p}(p-Q+\frac{1}{t})\, g_o(t) \ . \end{aligned} \tag{3.14}$$

Expanding $g_o(t)$ by partial fractions and the result by power series gives

$$p_o(t) = h'p_B t^2 \left(p-Q+\frac{1}{t}\right)$$
$$\cdot \left\{\frac{J_1^2}{(J_1-L_1)(J_1-1)}\frac{1}{1-J_1 t} + \frac{L_1^2}{(L_1-J_1)(L_1-1)}\frac{1}{1-L_1 t} + \frac{1}{(J_1-1)(L_1-1)}\frac{1}{1-t}\right\}$$
$$= h'p_B t + \sum_{n=2}^{\infty} \frac{1}{2}\left\{1+\frac{p_B}{P(J_1-L_1)}\left[(J_1+p-Q)J_1^n(L_1-1)-(L_1+p-Q)L_1^n(J_1-1)\right]\right\} t^n$$

(3.15)

The last expression allows us to recover $P_o(n)$ as the coefficient of the t^n term in the power series expansion, that is

$$P_o(1) = h'p_B$$
$$P_o(n) = \frac{1}{2}\left\{1+\frac{p_B}{P(J_1-L_1)}\left[(J_1+p-Q)J_1^n(L_1-1)-(L_1+p-Q)L_1^n(J_1-1)\right]\right\} \ ; \ n \geq 2 \ .$$

(3.16)

The probability of undetected error for a single parity check code is therefore

$$P_u = 1 - P_o(n) - \text{prob}(0^n)$$
$$= -\frac{1}{2}\left\{1+\frac{p_B}{P(J_1-L_1)}\left[(J_1+p-Q)J_1^n(L_1-1)-(L_1+p-Q)L_1^n(J_1-1)\right]\right\}$$
$$+\frac{p_B(1-h)}{J-L}\left\{(J+p-Q)\frac{1-J^n}{1-J} - (L+p-Q)\frac{1-L^n}{1-L}\right\} \ . \qquad (3.17)$$

As a check of this expression, we can consider for instance its large n limit. First, note that the quantity under the radical sign in the definition for J and L (1.5) can be written $[(Q-hp)-h]^2+4h(1-h)p>0$ and a straight forward application of Descarte's Rule of Signs shows that for $p \geq Q$, $1 \geq J > 0$, $0 \geq L > -1$ and for $p<Q$, $1>J,L>0$. That is, $|J|,|L|,|J_1|,|L_1|<1$. Hence all the exponentiated terms in (3.17) vanish for large n. The remaining terms combine to give $P_u=1/2$, expectedly the same result as for the BSC.

Figures 3.1 to 3.12 are plots showing P_u as a function of P and p for fixed values of h and n. We see graphically that P_u is monotonically increasing with P and decreasing with p and h. Hence for the single parity check code in the Gilbert channel, as channel conditions worsen, the probability of undetected error increases. This type of behaviour corresponds to that of a proper code in the BSC, that is a code for which P_u increases monotonically up to $(1/2)^{n-k}$ with the bit error rate [4]. The P_u also increases quite sharply with n for all channel parameter values, similar to the BSC case, as the single parity check code is quite limited in its error detection capabilities since it detects no even number of errors.

3.3 Probability of a Vector and its Cyclic Shifts

According to (1.3) and (1.4), the probability of an error pattern e is

$$\mathrm{prob}(e) = p_B u(a) \left\{ \prod_{i=1}^{wt(e)-1} v(b_i) \right\} u(c) \tag{3.18}$$

where a is the number of *0*'s before the first *1*, b_i is the number of *0*'s between the ith and (i+1)th *1*, and c is the number of *0*'s following the last *1* in e. Denote a right cyclic shift of s places by π_s. A negative s will correspond to a left cyclic shift. Then the probability of e and its next c right cyclic shifts and a left cyclic shifts is

$$\sum_{s=-a}^{c} \mathrm{prob}(\pi_s e) = p_B \left\{ \prod_{i=1}^{wt(e)-1} v(b_i) \right\} \sum_{s=0}^{b_0} u(s)u(b_0-s) \tag{3.19}$$

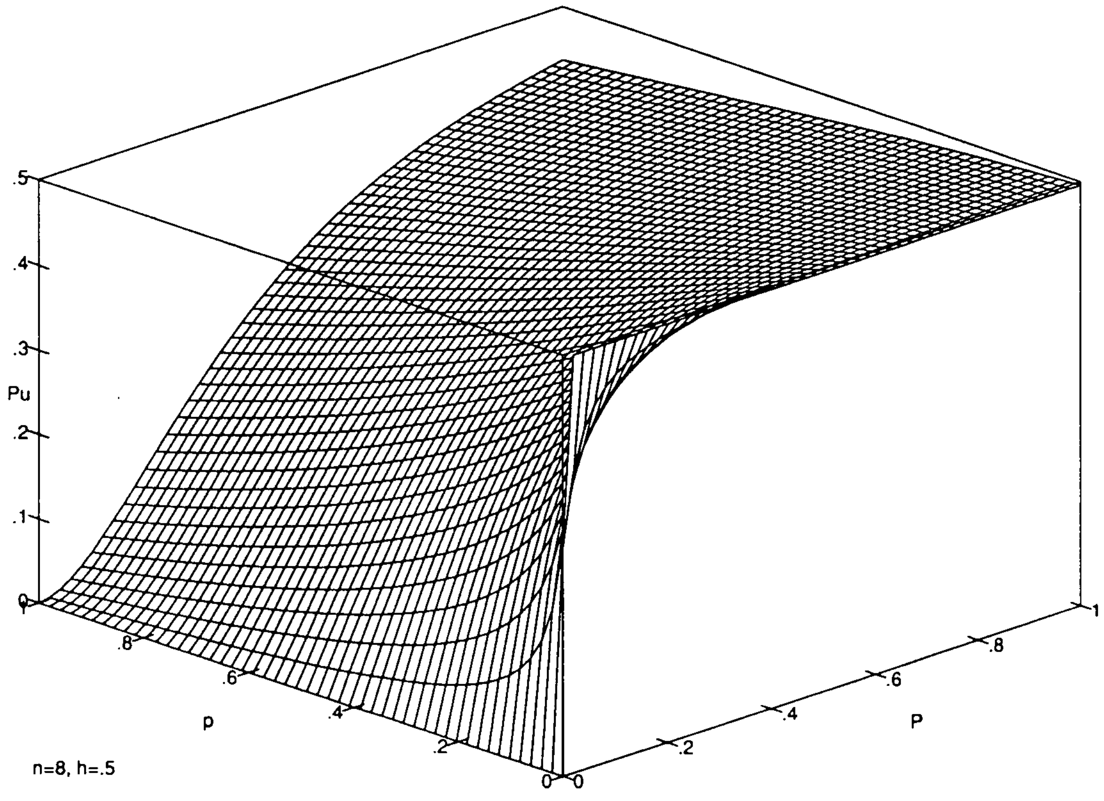


Figure 3.1 P_u for single parity check code, n=8, h=.5

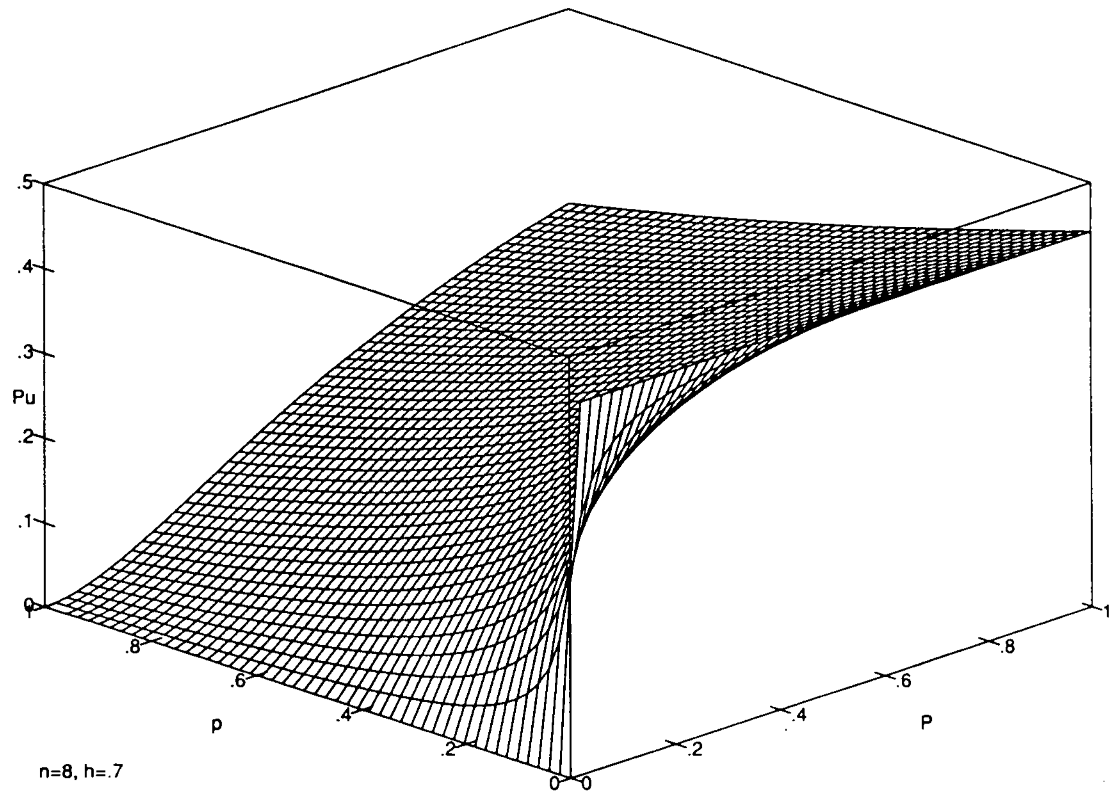


Figure 3.2 P_u for single parity check code, n=8, h=.7

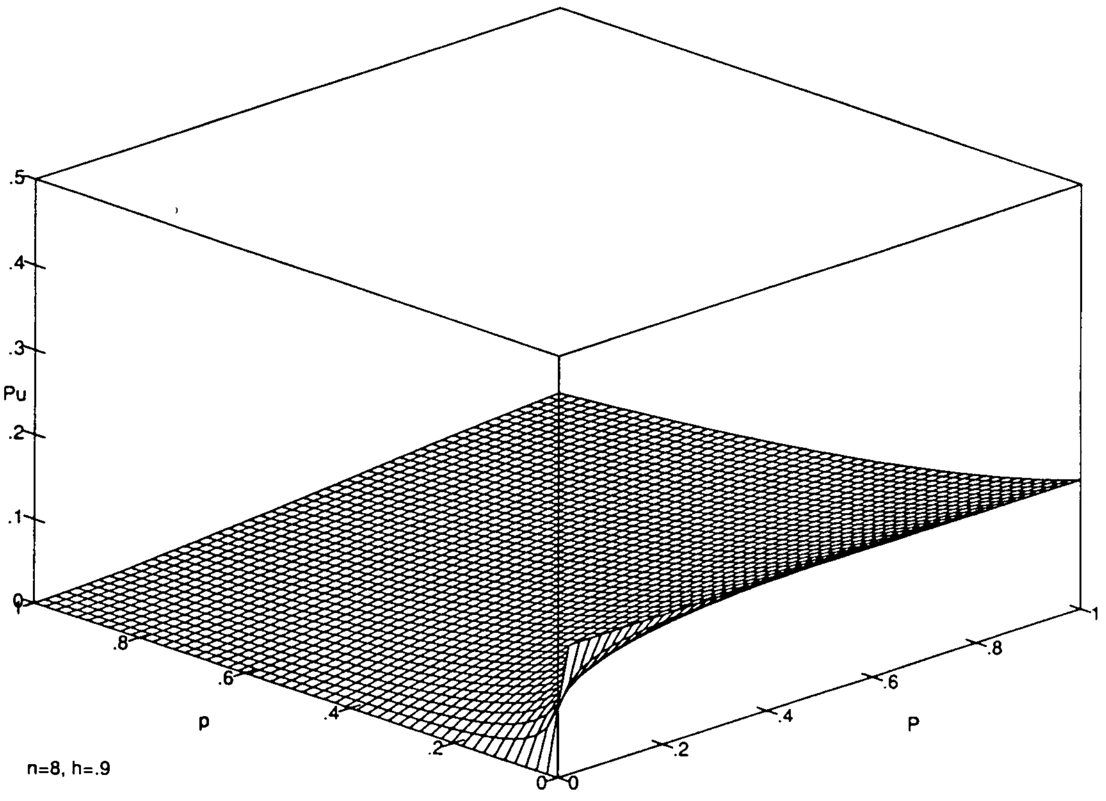


Figure 3.3 P_u for single parity check code, n=8, h=.9

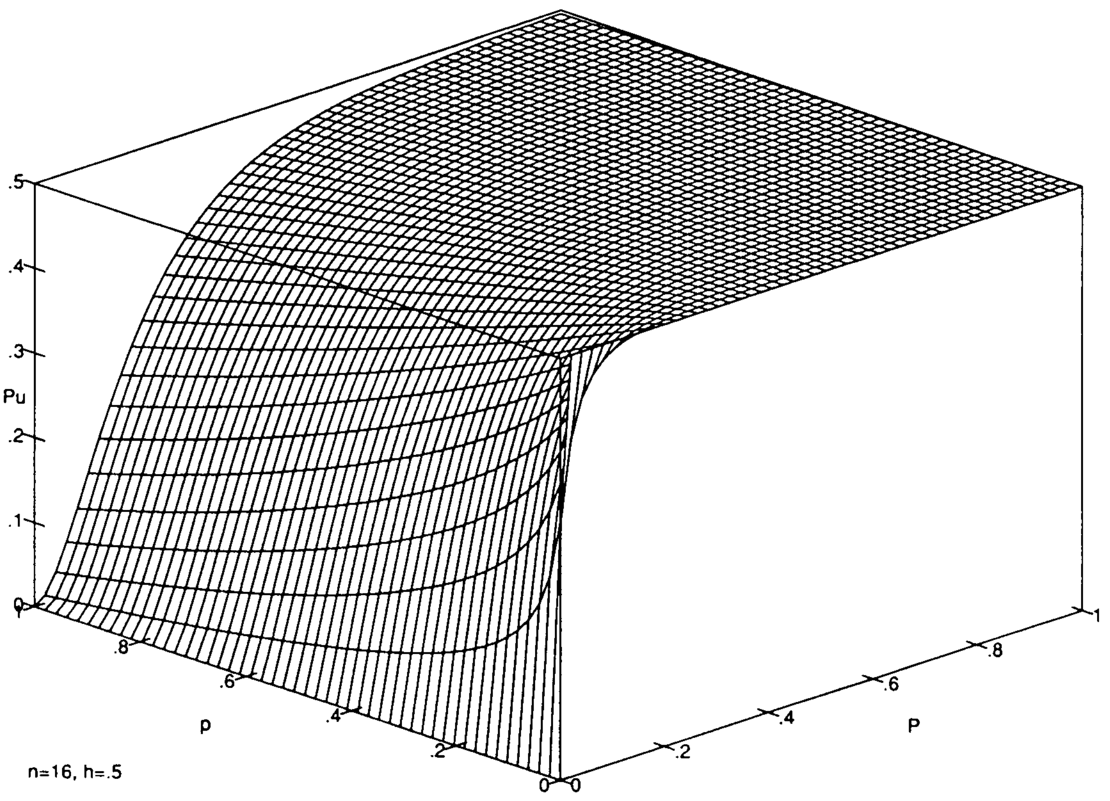


Figure 3.4 P_u for single parity check code, n=16, h=.5

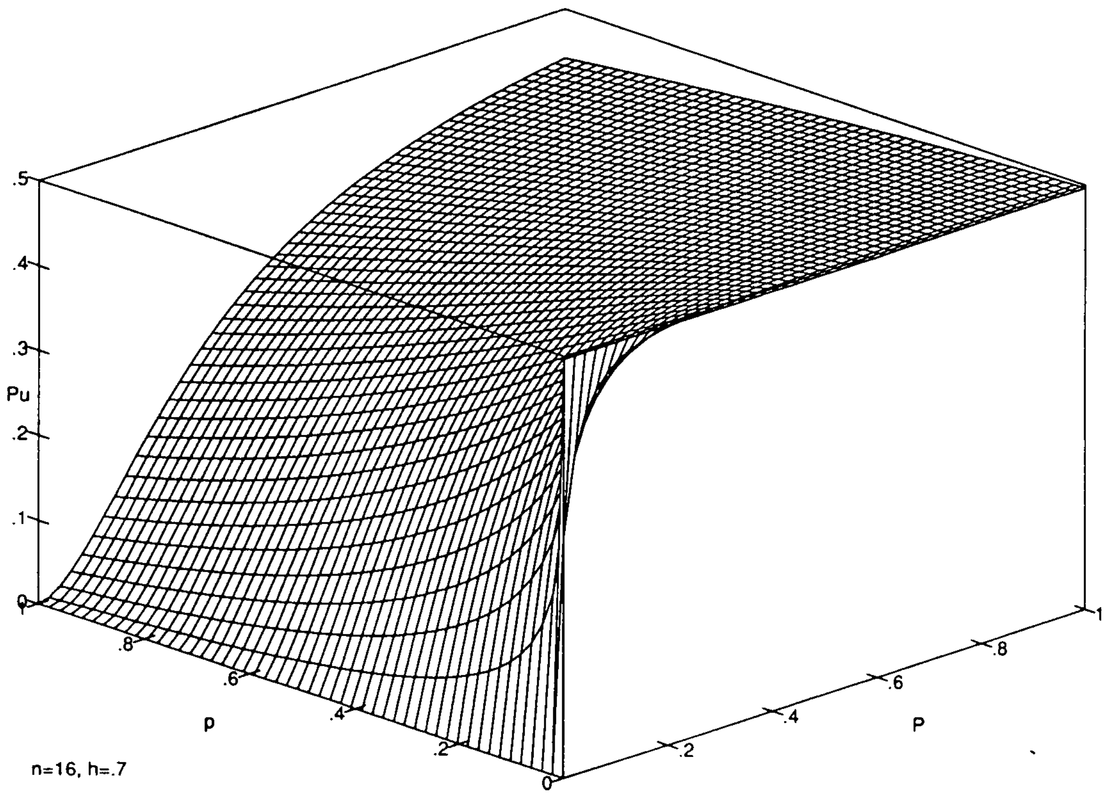


Figure 3.5 P_u for single parity check code, n=16, h=.7

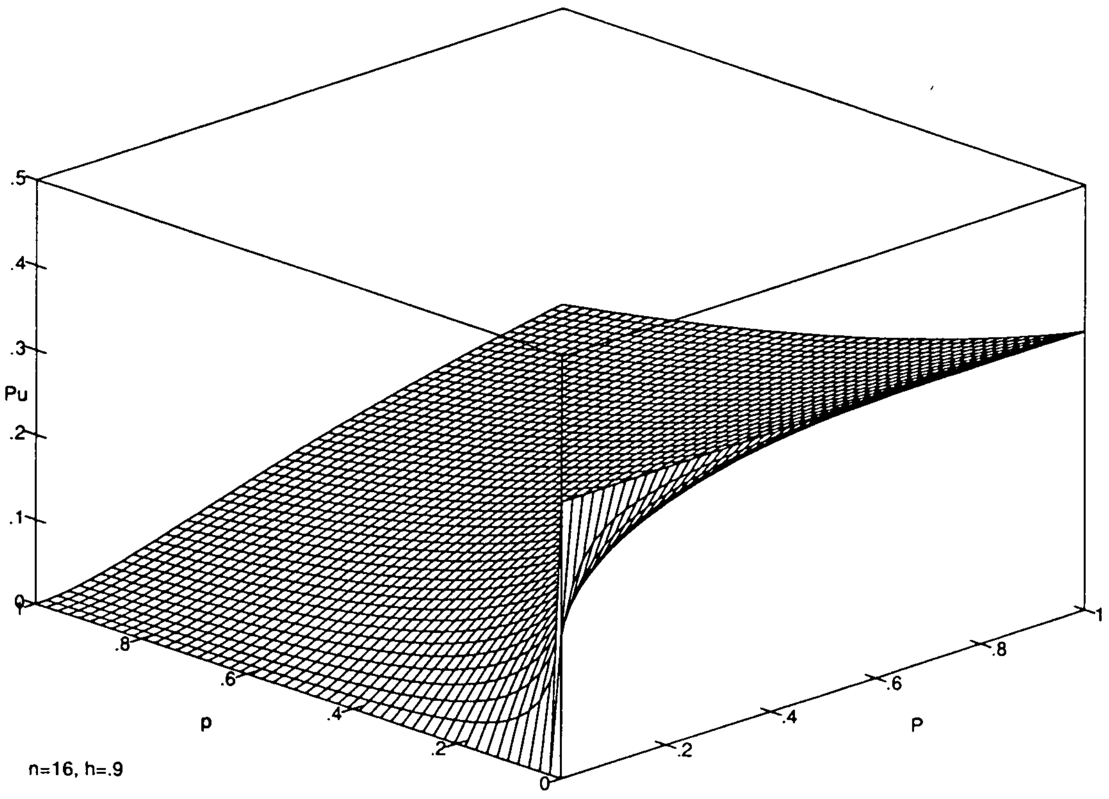
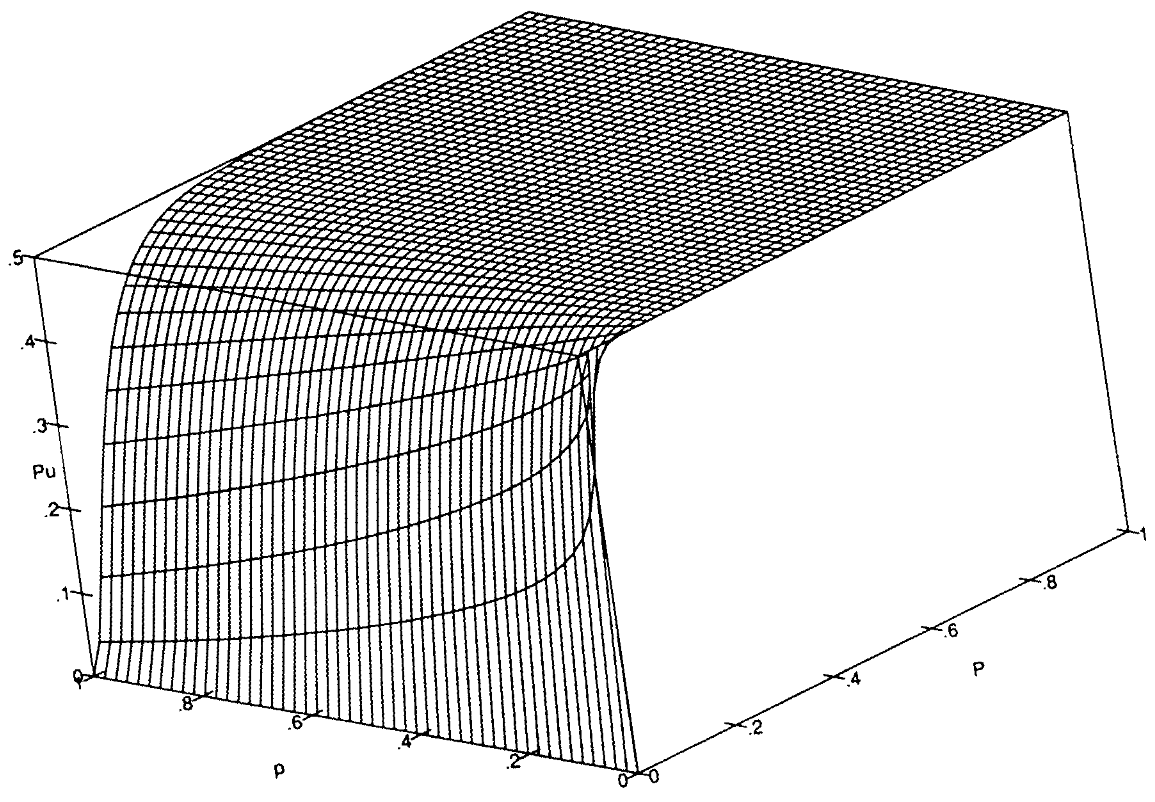


Figure 3.6 P_u for single parity check code, n=16, h=.9



n=32, h=.5

Figure 3.7 P_u for single parity check code, n=32, h=.5

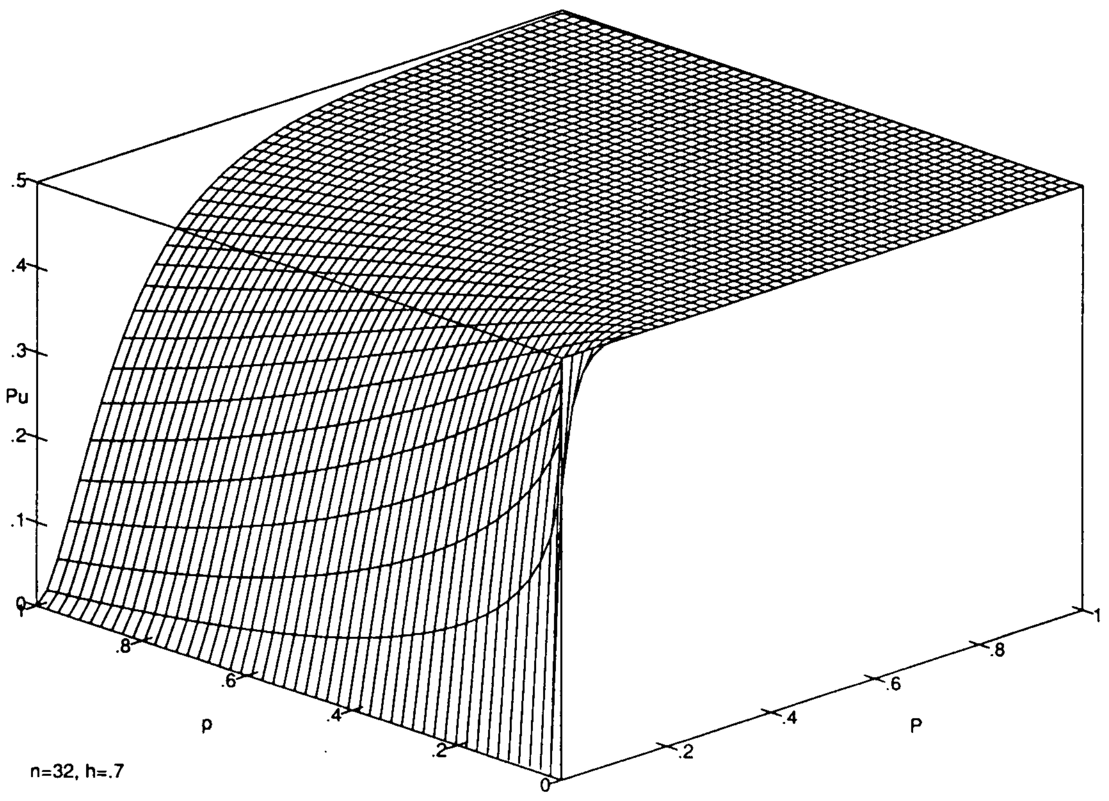


Figure 3.8 P_u for single parity check code, n=32, h=.7

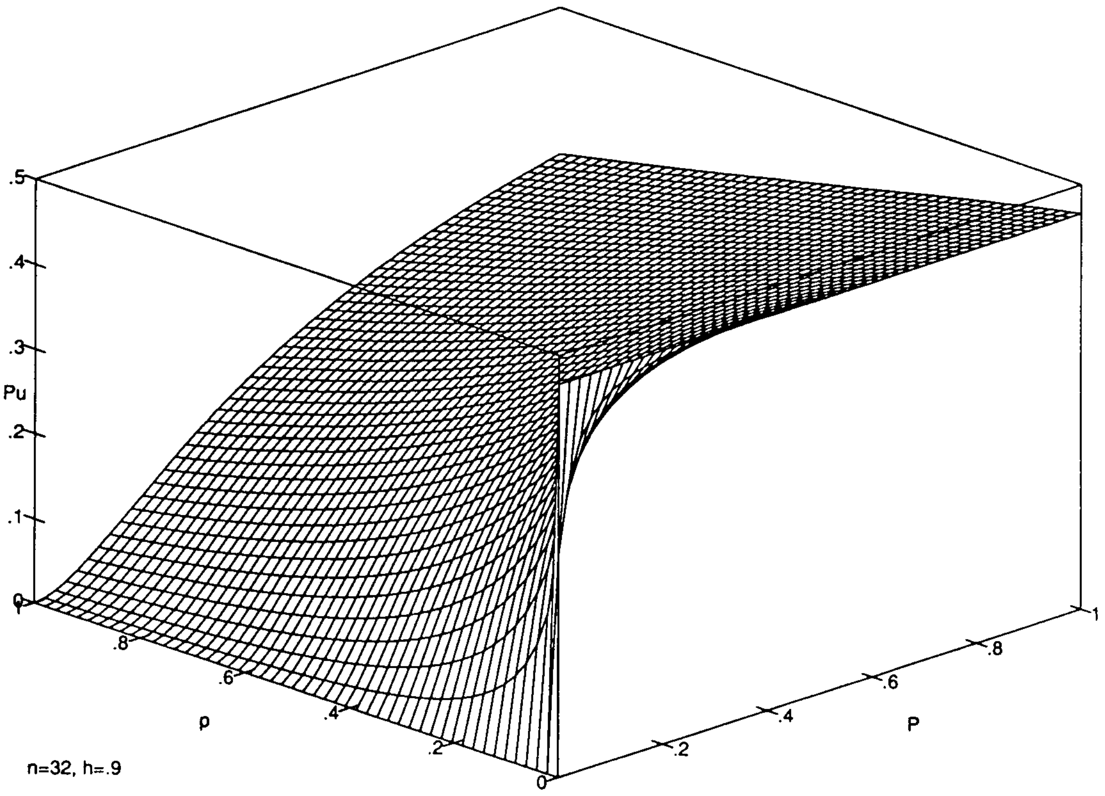


Figure 3.9 P_u for single parity check code, n=32, h=.9

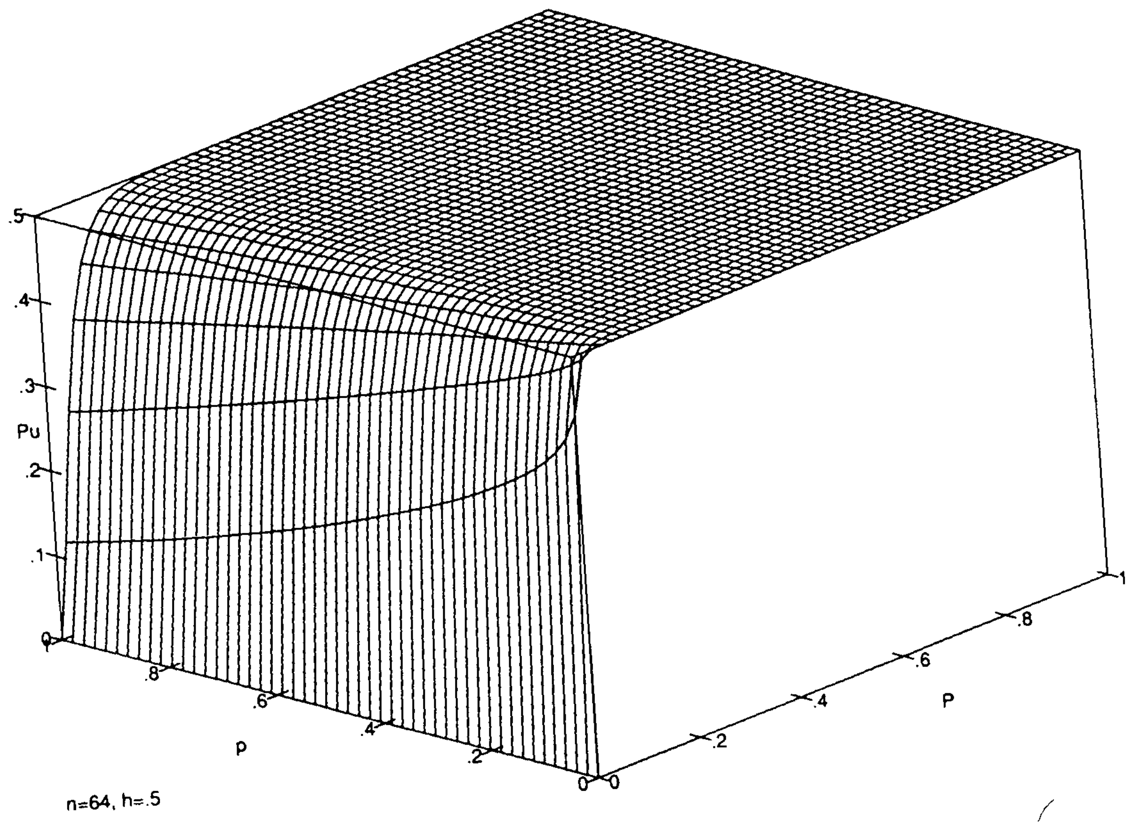


Figure 3.10 P_u for single parity check code, n=64, h=.5

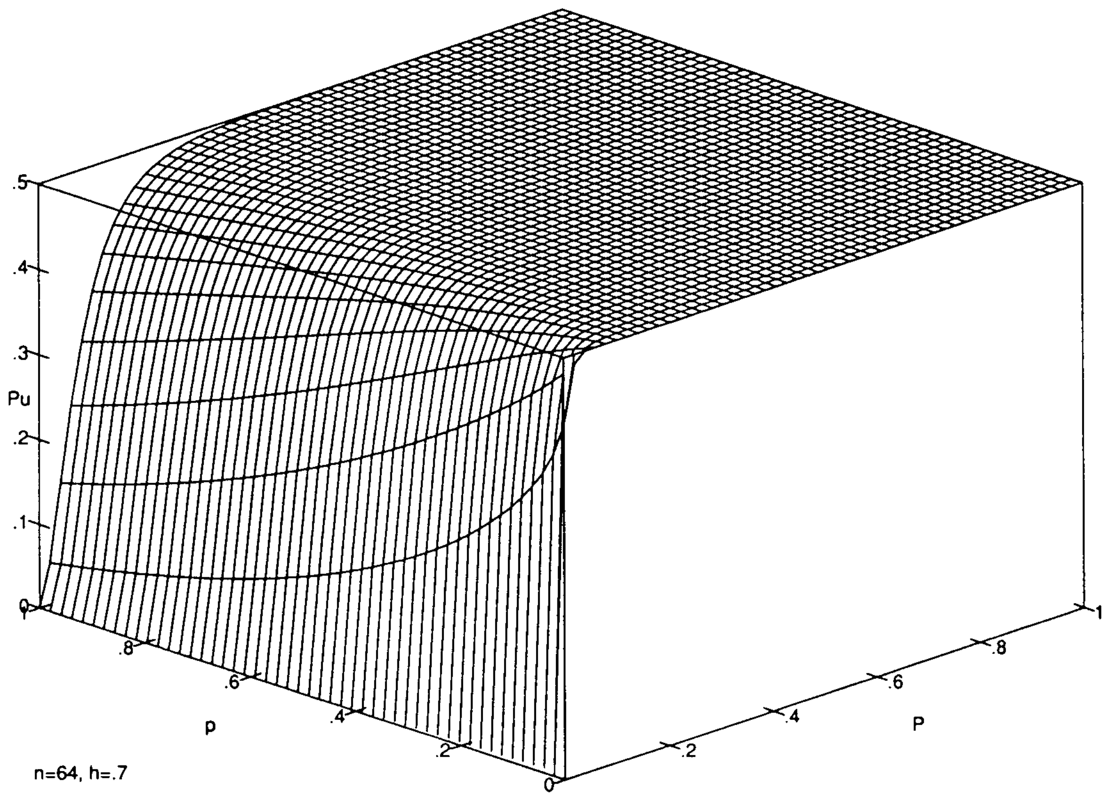


Figure 3.11 P_u for single parity check code, n=64, h=.7

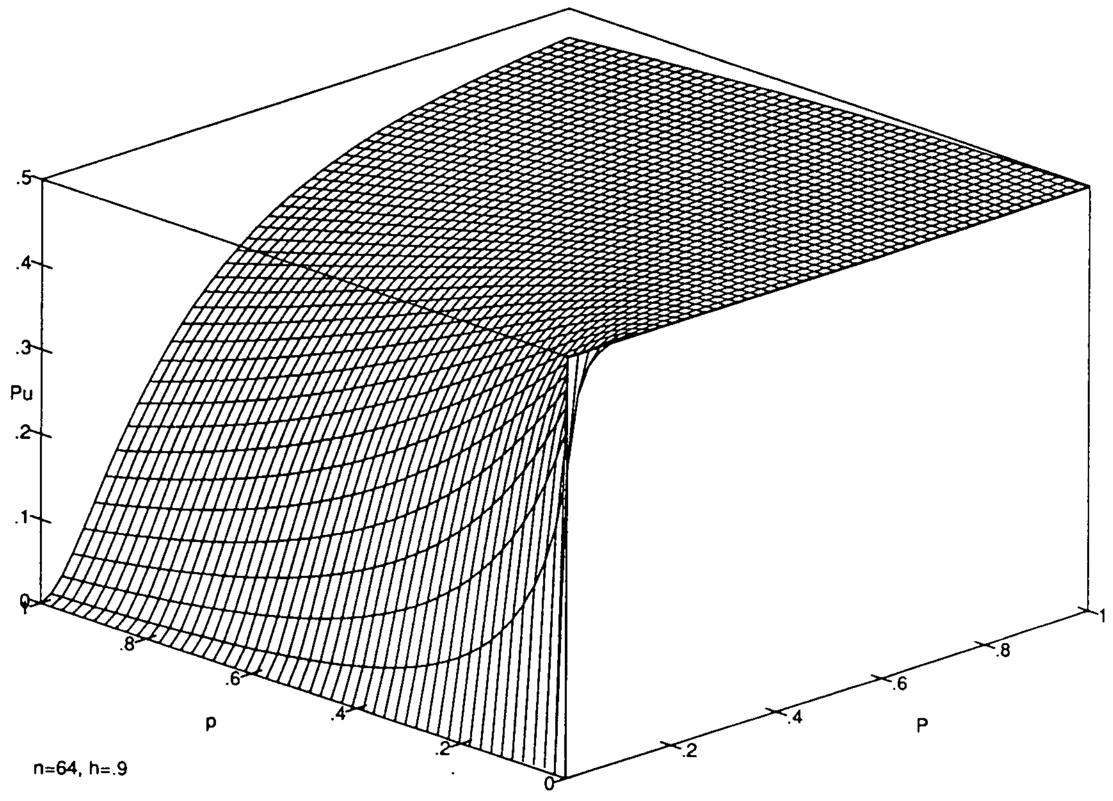


Figure 3.12 P_u for single parity check code, n=64, h=.9

where for convenience we have put $b_0=a+c$. The sum

$$u_2(d) = \sum_{s=0}^{d} u(s)u(d-s) \tag{3.20}$$

is easily handled using generating function methods. From [12], the generating function for $u(\cdot)$ is

$$U(t) = \frac{1+(p-Q)t}{J-L}\left\{\frac{J}{1-Jt} - \frac{L}{1-Lt}\right\} . \tag{3.21}$$

Since u_2 is the self convolution of u, the generating function for u_2 is

$$\begin{aligned} U_2(t) &= U(t)^2 \\ &= \frac{1+2(p-Q)t+(p-Q)^2t^2}{(J-L)^2}\left\{\frac{J^2}{(1-Jt)^2} - \frac{2JL}{J-L}\left[\frac{J}{1-Jt} - \frac{L}{1-Lt}\right] + \frac{L^2}{(1-Lt)^2}\right\} . \end{aligned} \tag{3.22}$$

By expanding $U_2(t)$ as a power series in t, it follows that

$$\begin{aligned} u_2(k) &= \frac{1}{(J-L)^2}\{(k+1)(J^{k+2}+L^{k+2})+2k(p-Q)(J^{k+1}+L^{k+1})+(k-1)(p-Q)^2(J^k+L^k)\} \\ &- \frac{2JL}{(J-L)^3}\{(J^{k+1}+L^{k+1})+2(p-Q)(J^k+L^k)+(p-Q)^2(J^{k-1}+L^{k-1})\} . \end{aligned} \tag{3.23}$$

Putting $b=(b_0,\ldots,b_{wt(e)-1})$, the total probability of e and all n of its cyclic shifts is therefore

$$\sum_{s=0}^{n-1} prob(\pi_s e) = \sum_{t=0}^{wt(e)-1} u_2((\pi_t b)_0)v((\pi_t b)_1)\ldots v((\pi_t b)_{wt(e)-1}) . \tag{3.24}$$

The last equation is invariant with respect to $e \rightarrow \pi_s e$ as it must be since it is the probability of a cyclic shift class of vectors of which e is but one member.

As an example, consider the dual Hamming codes. (A dual code of a linear block code C(n,k) is a linear block code C*(n,n-k) consisting of all vectors which have vanishing inner product with the codewords of C(n,k) [1-3].) The dual Hamming (2^m-1,m) code consists only of the zero vector and a single cyclic class of vectors of weight 2^{m-1} generated by the reciprocal of the parity polynomial $h(x)=(x^n+1)/g(x)$ where g(x) is the generating polynomial of the Hamming (2^m-1,2^m-m-1) code. For m=3 and $g(x)=1+x+x^3$, $h^*(x)=1+x^2+x^3+x^4$ so the probability of undetected error is the probability of the cyclic class generated by the codeword pattern (*1 0 1 1 1 0 0*), that is

$$\begin{aligned} P_u &= u_2(2)v(1)v(0)v(0) + u_2(1)v(0)v(0)v(2) + u_2(0)v(0)v(2)v(1) \\ &\quad + u_2(0)v(2)v(1)v(0) \\ &= v(0)\{u_2(2)v(1)v(0)+u_2(1)v(0)v(2)+2u_2(0)v(2)v(1)\} \ . \end{aligned} \tag{3.25}$$

Of course, in this simple case, one could just as well have summed the 7 codeword patterns explicitly and obtained the same result.

As another example, consider the BCH (31,6) code encountered in Chapter 2. The distinct generator polynomials (not counting reciprocals) are known to be

(i). $1+x+x^2+x^5+x^9+x^{11}+x^{13}+x^{14}+x^{15}+x^{16}+x^{18}+x^{19}+x^{21}+x^{24}+x^{25}$,

(ii). $1+x^4+x^5+x^6+x^8+x^{10}+x^{13}+x^{15}+x^{16}+x^{17}+x^{18}+x^{21}+x^{22}+x^{24}+x^{25}$,

(iii). $1+x+x^2+x^3+x^6+x^7+x^9+x^{11}+x^{14}+x^{15}+x^{16}+x^{17}+x^{21}+x^{22}+x^{25}$.

A BCH code always contains the all *1*'s vector since the generator polynomial never contains the (x+1) factor. The weights of the codewords corresponding to the generator polynomials are all 15, so the BCH (31,6) code must just consist of two cyclic classes of 31 codewords each, with respective weights 15 and 16 in addition to the all *0*'s and all *1*'s codewords. Writing out the components of the b vector as defined after (3.23) corresponding to the above code polynomials, we get (i). (5,0,0,2,3,1,1,0,0,0,1,0,1,2,0)

(ii). (5,3,0,0,1,1,2,1,0,0,0,2,0,1,0)

(iii). (5,1,0,0,1,0,1,1,2,0,0,0,3,0,2)

which clearly give rise to the same probabilities for the respective cyclic classes according to (3.24). The situation is the same with the weight 16 codewords which are just the complements of the weight 15 codewords. Hence without doing much work, we have shown that all the BCH (31,6) codes have the same P_u in the Gilbert channel.

Similarly, an explicit expression for P_u can be written using the rule (3.23) for other simple codes which have but a few readily identifiable cyclic classes of codewords such as the Hamming (7,4) code, etc. However, that is not the point of the exercise.

Rather, what we want to point out to conclude this section is the following. In a general channel, all 2^k codewords can conceivably have different probabilities. If there is no way to analytically relate the codewords, the exact determination of P_u (for large k) is surely an intractable problem. For the BSC, the codewords are divided into weight classes, each characterized by a certain probability. For the Gilbert channel, we know this tactic will not be immediately applicable. However, due to the underlying symmetry of the Gilbert channel, we see from the above that n vectors which are cyclic permutations of each other can be grouped together and have a total probability dependent on only a single pattern. The cyclic codes appear to fit naturally into this scheme. As there are no other obvious channel and code symmetries that can be identified this way, the cyclic codes probably are the best candidates for exact analysis. The problems are to first identify or enumerate the various cyclic classes and then provide summation rules between them. Of the latter, we would expect to apply techniques similar to that which has been given here for summing over collections of $u_2(\cdot)$ and $v(\cdot)$.

3. 4 Reciprocal Generator Polynomials

Here we derive an interesting fact regarding the cyclic code generated by the reciprocal of the generator polynomial of a given cyclic code. If g(x) is a generator polynomial of a cyclic code, it necessarily divides x^n+1. It follows that the reciprocal polynomial $g^*(x)$ must also divide x^n+1 and therefore it also generates a cyclic code of the same length [1-3]. Denote the degree of g(x) (and $g^*(x)$) by q. A code polynomial in the cyclic code generated by g(x) may be written $c(x)=g(x)s(x)$ where s(x) is of degree at most n-q-1. Write $s(x)=(x^a+...+1)x^b$ (where $a+b \leq n-q-1$) so that $c(x)=g(x)(x^a+...+1)x^b$. Now in the code generated by $g^*(x)$, there is a codeword given by $d(x)=g^*(x)(x^a+...+1)^*x^b$. Suppose first for simplicity that b=0, then $d(x)=c^*(x)$. In coordinate form, the first q+a+1 bits of the vectors c and d will be mirror image patterns of each other with the 1st and (q+a+1)th bits *1*'s. The remaining n-q-a-1 bits are *0*'s. From (1.3) and (1.4), it is clear that mirror image patterns have the same probability. The *0*'s following the last *1* just contribute a factor u(n-q-a-1) to the net probability of the codeword in both cases. Hence prob(c)=prob(d). Now if $b \neq 0$, then the vectors c and d will both have b *0*'s at one end (contributing a factor w(b) to the net probability) and n-q-a-b-1 *0*'s at the other end (contributing a factor u(n-q-a-b-1) to the net probability) with mirror image patterns of length

q+a+1 delimited by *1*'s in the middle. Therefore, again **prob(c)=prob(d)**. Clearly, the c(**x**) and d(**x**) above define a one to one correspondence between codewords in the cyclic codes generated by g(x) and g*(x) respectively. Hence cyclic codes generated by reciprocal generator polynomial pairs have identical P_u on the Gilbert channel.

4. Monte Carlo Simulation

4.1 The Monte Carlo Method

When analytical methods are unavailable, a common technique used to study the performance of communication systems is computer simulation. Usually, the formalism is phrased in terms of that for bit-error rate estimation [19] so first we will rewrite the terminology so that it is appropriate for our case.

The probability of undetected error of a code with block length n can be written

$$P_u = \sum_{v \in V_n} H(v) \mathrm{prob}(v) \tag{4.1}$$

where

$$H(v) = \begin{cases} 1 & v \in C, v \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

is the error detector. In other words, P_u is the expectation value of the error detector. A natural estimator of the expectation is just the sample mean

$$\hat{P}_u = \frac{1}{N} \sum_{i=1}^{N} H(v_i) \tag{4.3}$$

where v_i is the ith of a total of N blocks sent through the channel. Hence a suitable basis for estimating the error rate is by the observation of errors; this defines the Monte Carlo method. As $N \to \infty$, the estimate $\hat{P}_u$ converges to P_u. For finite N, the reliability of the estimator is quantified in terms of a confidence level α

and its associated confidence limits, $y_+(\hat{P}_u)$ and $y_-(\hat{P}_u)$ defined through the relation

$$\text{prob}(y_+ \leq P_u \leq y_-) = 1-\alpha \ . \tag{4.4}$$

$\hat{P}_u$ is binomially distributed. By applying the normal approximation to the binomial distribution, it can be shown that

$$y_\pm = \hat{P}_u \left\{ 1 + \frac{d_\alpha^2}{2N\hat{P}_u} \left[1 \pm \sqrt{1 + \frac{4N\hat{P}_u}{d_\alpha^2}} \right] \right\} \tag{4.5}$$

where d_α satisfies

$$\int_{-d_\alpha}^{d_\alpha} dt \frac{1}{\sqrt{2\pi}} e^{-t^2/2} = 1 - \alpha \ . \tag{4.6}$$

For our purposes, we always take α=.01, then d_α=2.575829, and the corresponding confidence limits define a 99% confidence interval.

4.2 Simulation Study of CRC-16 Codes

A CRC code is defined by a qth degree generator polynomial $g(x)=g_qx^q+g_{q-1}x^{q-1}+...+g_0$. Denote the polynomial representing the message by $s(x)=s_{k-1}x^{k-1}+s_{k-2}x^{k-2}+...+s_0$. A codeword is formed by appending to the message bits certain parity bits corresponding to the so called CRC polynomial $r(x)=\text{remainder}[x^qs(x)/g(x)]$. The polynomial representation of the codeword is then $c(x)=s(x)x^q+r(x)=s_{k-1}x^{q+k-1}+s_{k-2}x^{q+k-2}+...+s_0x^q+r_{q-1}x^{q-1}+r_{q-2}x^{q-2}+...+r_0$. CRC codes are often used in practical error control as the encoding (modulo-2 polynomial division) is particularly simple to implement by feedback shift register circuitry. If the generator polynomial is formed from the product of (x+1) and a primitive polynomial of degree q-1 (and the block length n is less than 2^{q}-1), it can be shown that the resulting CRC code detects all double errors, all odd numbers of errors, and all bursts (including end-around) of length q or less [21]. There are two commonly used standard CRC codes with q=16, both have the required form discussed above. Their generator polynomials are

$x^{16}+x^{15}+x^{2}+1$, for CRC-ANSI ,

$x^{16}+x^{12}+x^{5}+1$, for CRC-CCITT .

The performance of these CRC codes have been studied in the BSC by Witzke and Leung [9]. Both codes are not proper for all values of k. The probability of undetected error as a function of the error rate ε reaches a maximum before decreasing to the asymptotic value of $(1/2)^{q}$ at $\varepsilon=1/2$. The CRC-CCITT code performs better (in terms of probability of undetected error) than the CRC-ANSI code for all values of ε.

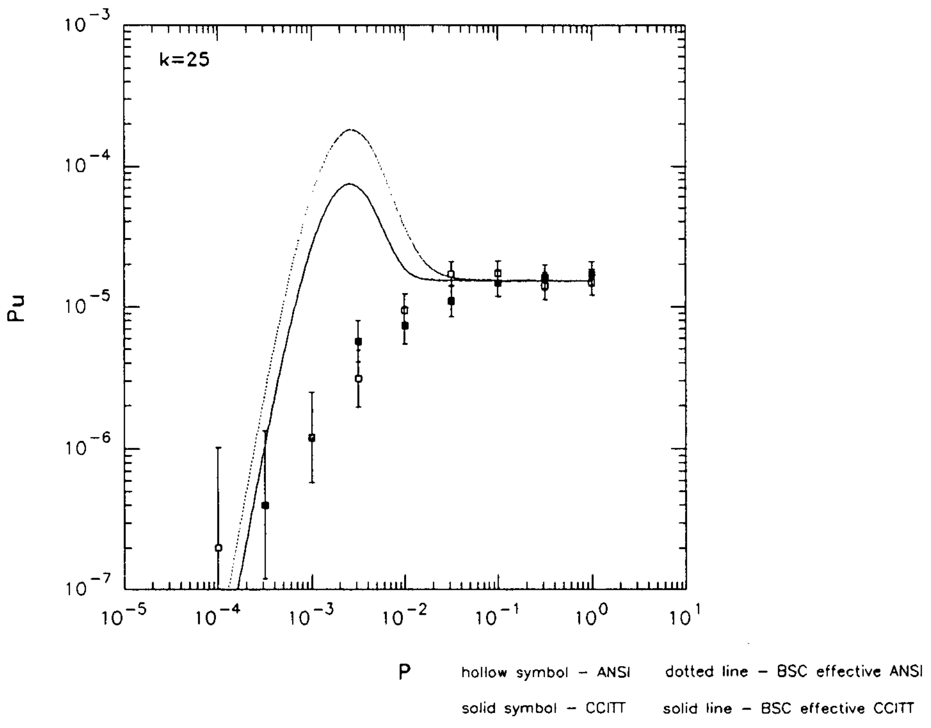
In view of the connection with burst detection and the previous study, we have chosen the CRC-ANSI and CRC-CCITT codes as the subject for Monte Carlo simulation study. While we are mainly interested in questions of practical concern such as whether the superiority of CRC-CCITT over CRC-ANSI persists over a Gilbert channel and how the results compare with those of the BSC, the study will also give some general idea of the size of problem that can be tackled in a practical implementation of the Monte Carlo technique on the Gilbert channel. We take k=25,50.

The simulation program is written in the C language, using the library function rand() for random number generation. Random bit-vectors

$v=(v_0,v_1,\ldots,v_{n-1})$ are generated according to the distribution implied by Figure 1.1. That is, before the first bit is generated, the state of the channel is set randomly to the G or B state according to the ratio p_G/p_B. At each successive bit, if the channel is in the G state, a *0* is generated and a transition to the B state occurs randomly with probability P. If the channel is in the B state, a *0* or *1* bit is generated according to the ratio h/h' and a transition to the G state occurs randomly with probability p. The procedure ends after n bits corresponding to an entire vector are generated. The probability of any vector so produced will be in accordance with (1.3) and (1.4). The vector is checked to determine whether it is a codeword using shift register arithmetic. If so, the error counter is incremented as indicated by (4.2). This procedure of vector generation and checking is repeated N times. The estimated probability of error and the 99% confidence levels are then computed using (4.3) and (4.5).

Running the simulation on the SPARC® station 1, we have found that for $N=10^7$ and k=25, a single run (corresponding to one set of parameters P,p,h) requires on the average of approximately 2.5 hours. For k=50, the average time required is approximately 4.0 hours. It is clear from the nature of this simulation that the computational time increases linearly with n. The parameter values which we have considered are $P=\{10^{-4},10^{-3.5},10^{-3},10^{-2.5},10^{-2},10^{-1.5},10^{-1},10^{-.5},1\}$, p={0.01,0.1,0.3}, h={0.5,0.7,0.9}; therefore, 81 "data points" $\hat{P}_u$ in all.

The results of our simulation are plotted in Figures 4.1 to 4.9. For clarity, we do not draw lines through the simulation data points. The error bars around these points bound the 99% confidence intervals. The lines on the plots are the P_u of the CRC codes for the BSC at the corresponding ε_{eff}. We have chosen to display the plots with P along the x-axis and fixed values of p and h. This is



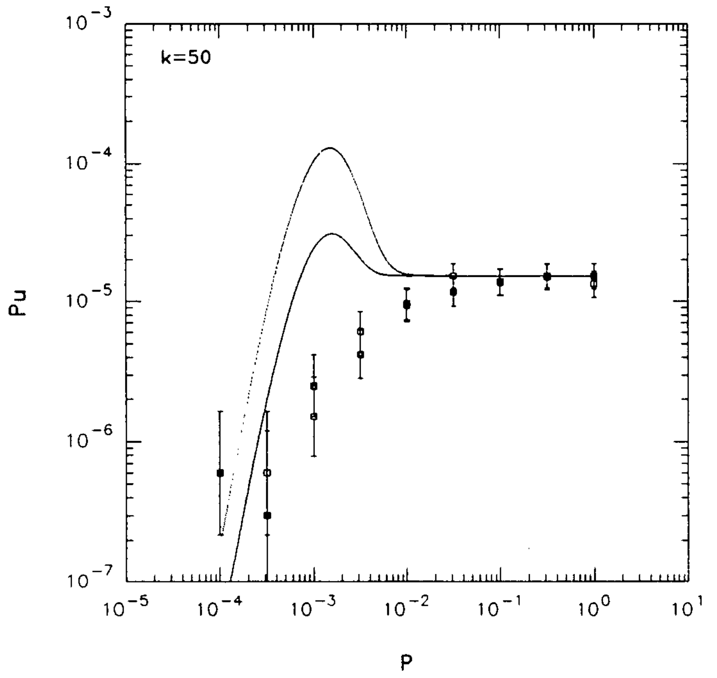
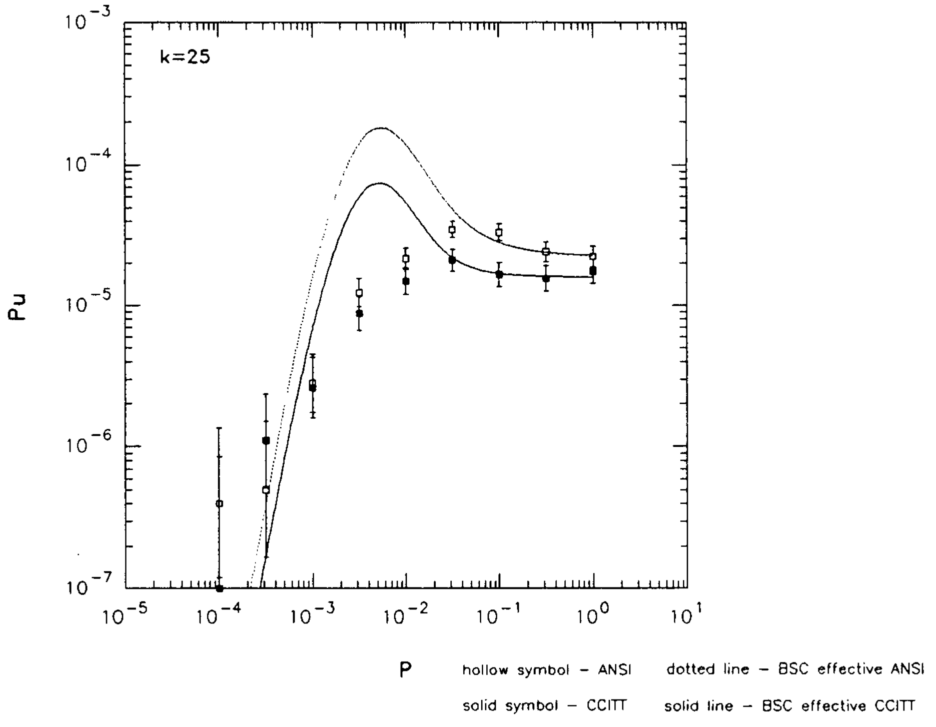


Figure 4.1 Simulation of CRC−16 codes, p=.01, h=.5



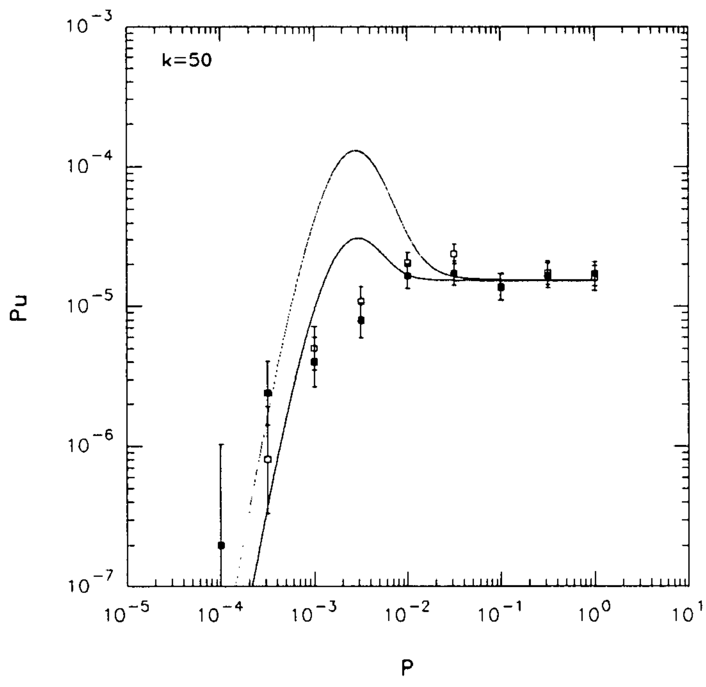
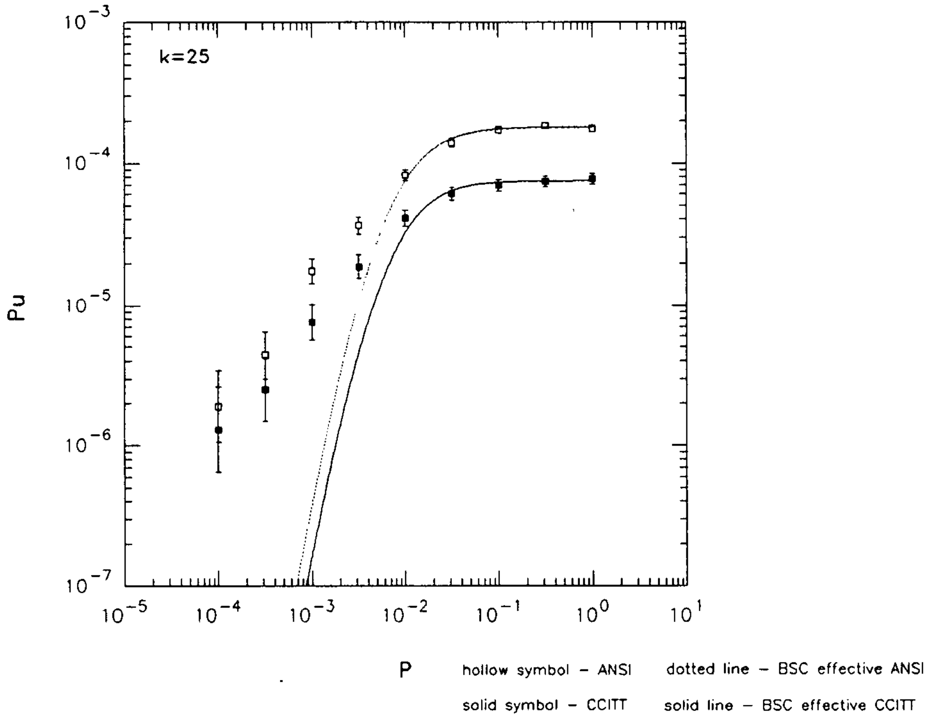


Figure 4.2 Simulation of CRC−16 codes, p=.01, h=.7



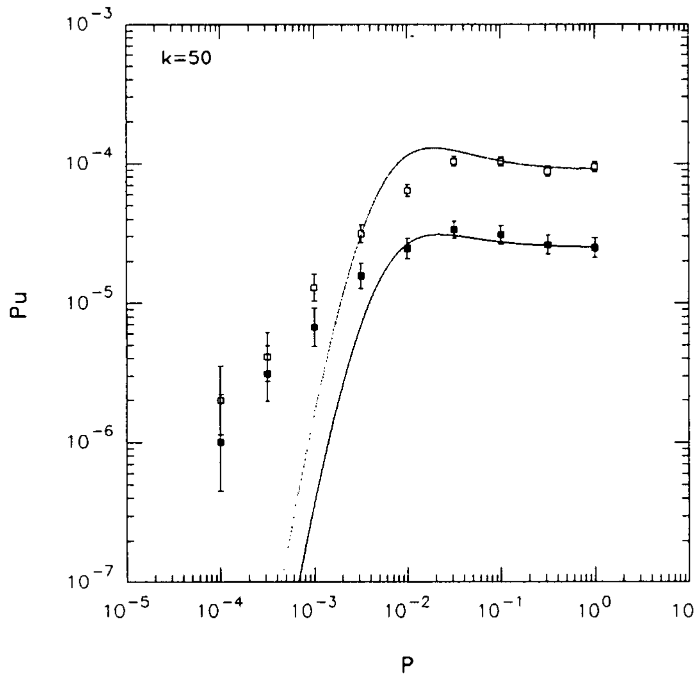


Figure 4.3 Simulation of CRC−16 codes, p=.01, h=.9

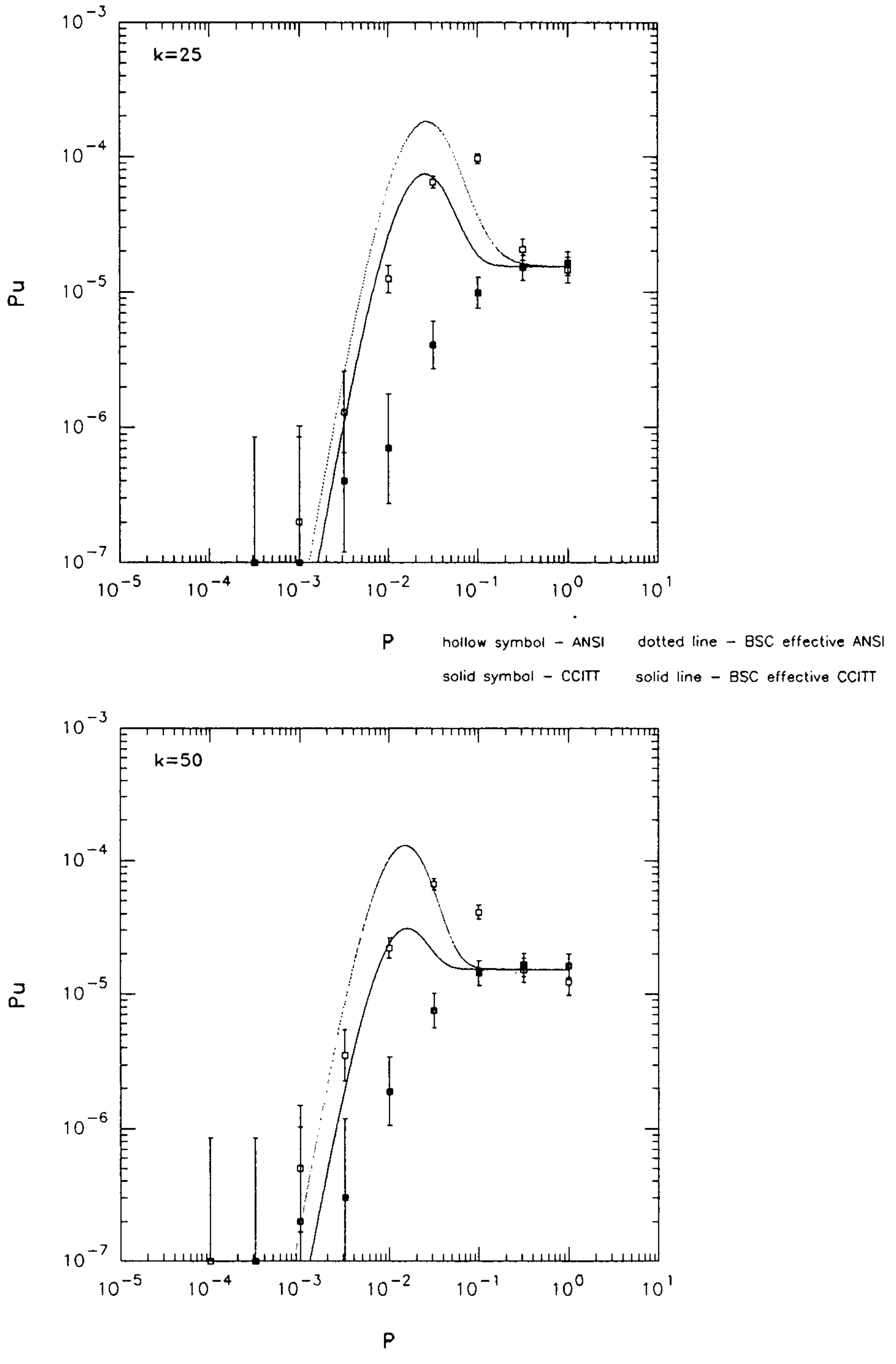
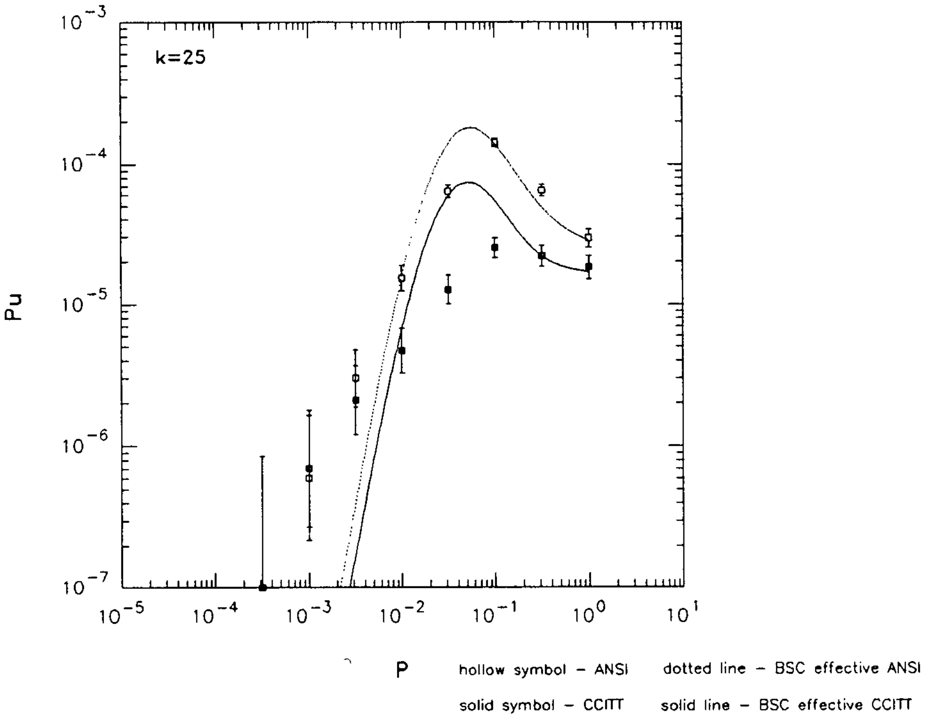


Figure 4.4 Simulation of CRC−16 codes, p=.1, h=.5



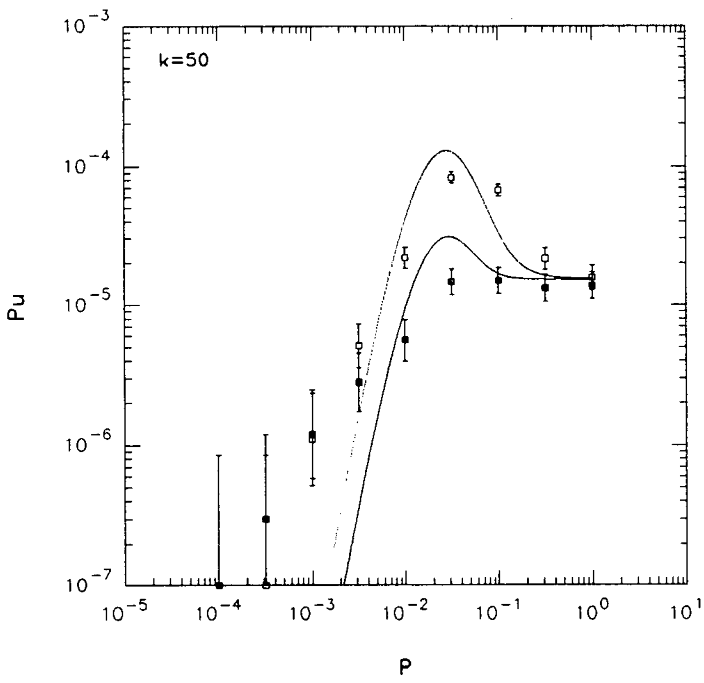
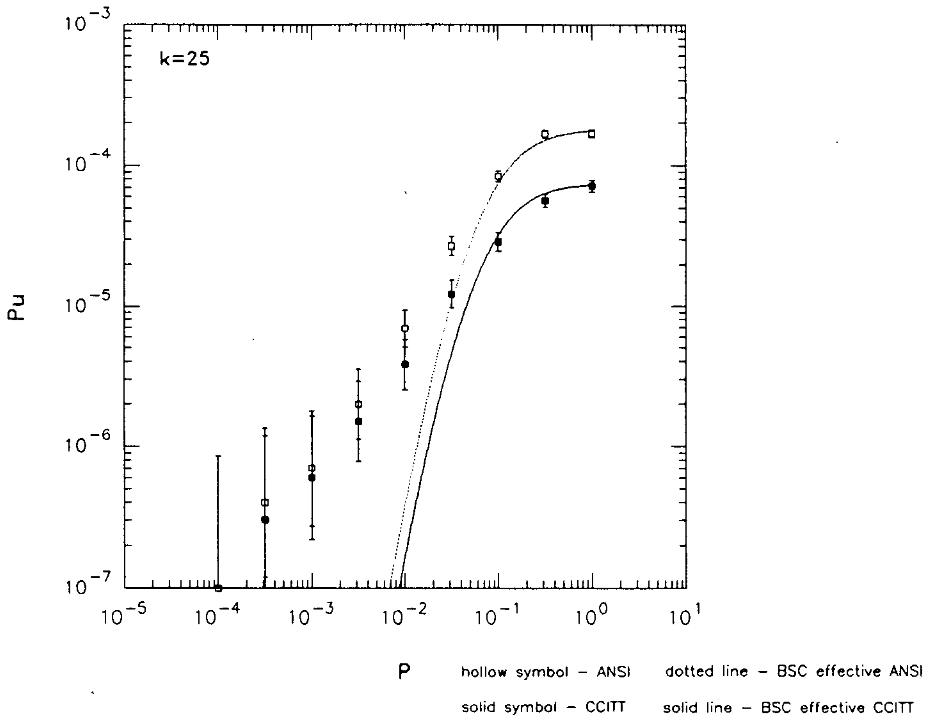


Figure 4.5 Simulation of CRC−16 codes, p=.1, h=.7



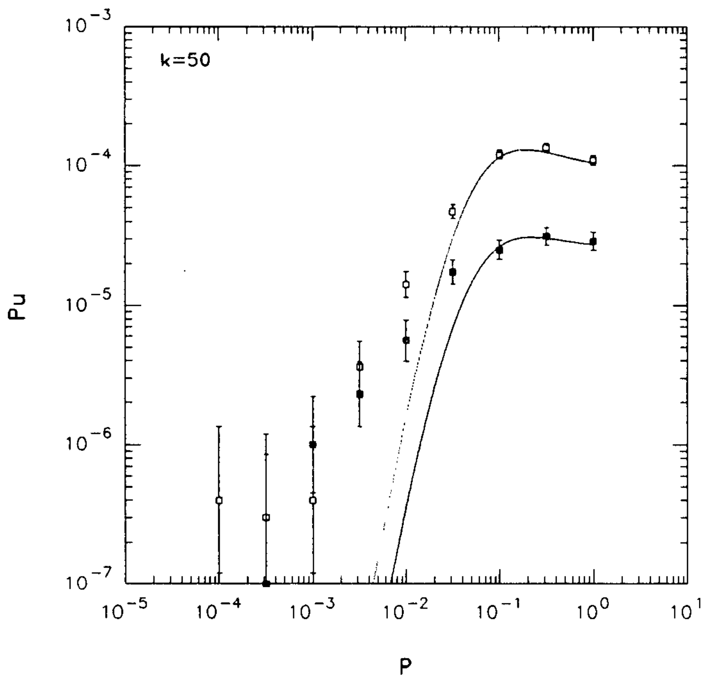
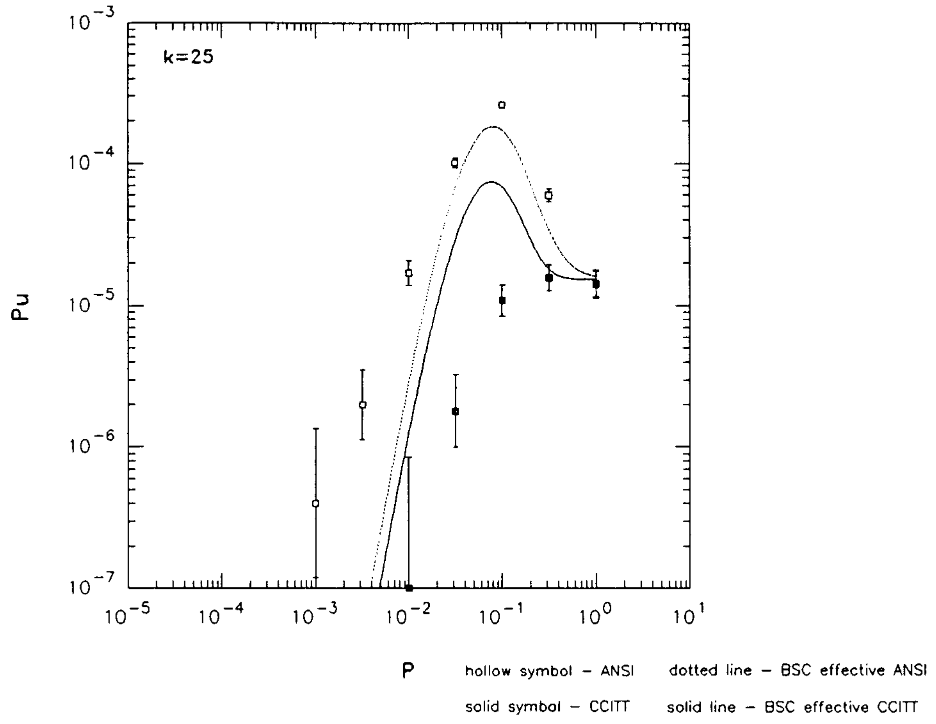


Figure 4.6 Simulation of CRC−16 codes, p=.1, h=.9



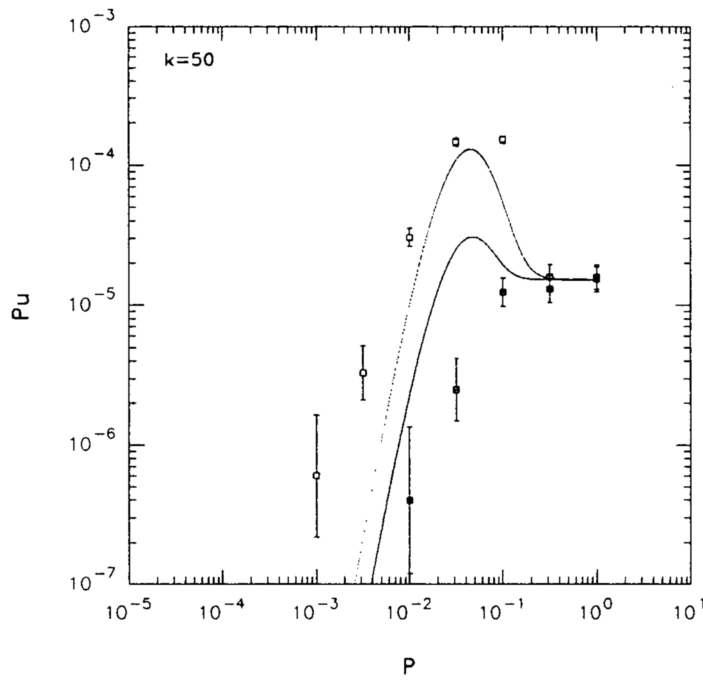


Figure 4.7 Simulation of CRC–16 codes, p=.3, h=.5

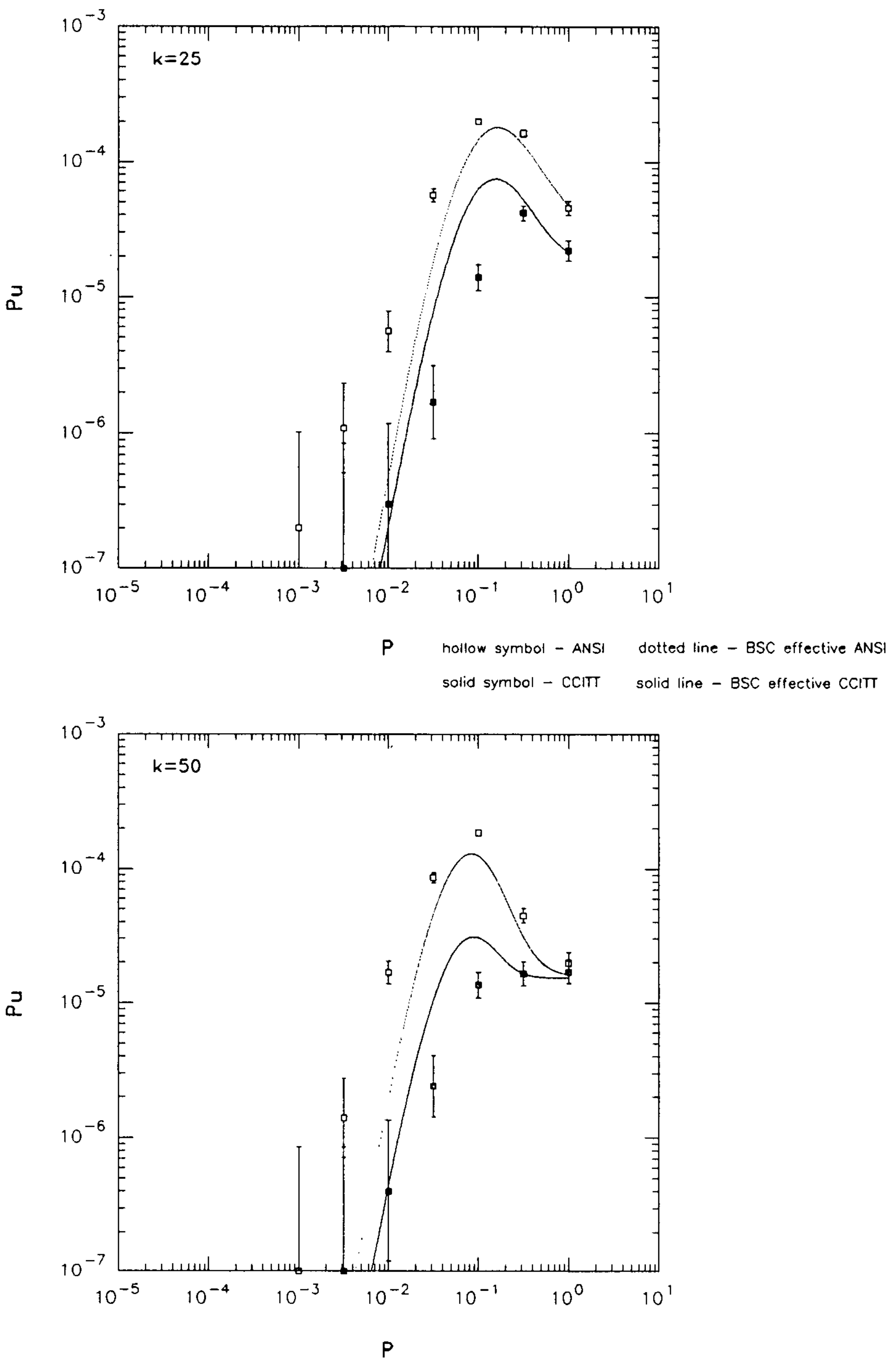
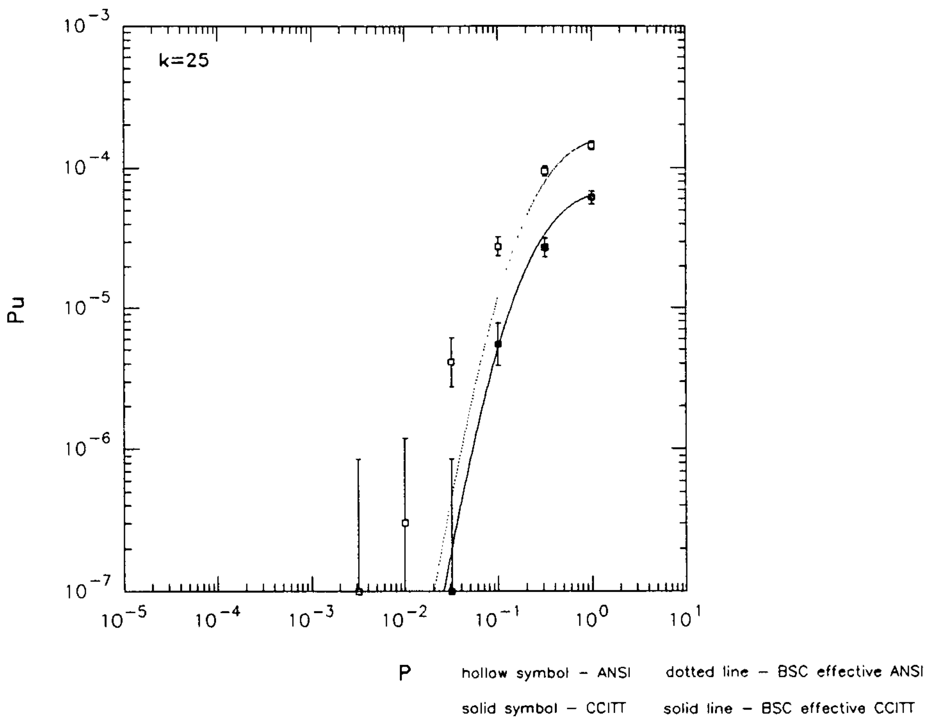


Figure 4.8 Simulation of CRC−16 codes, p=.3, h=.7



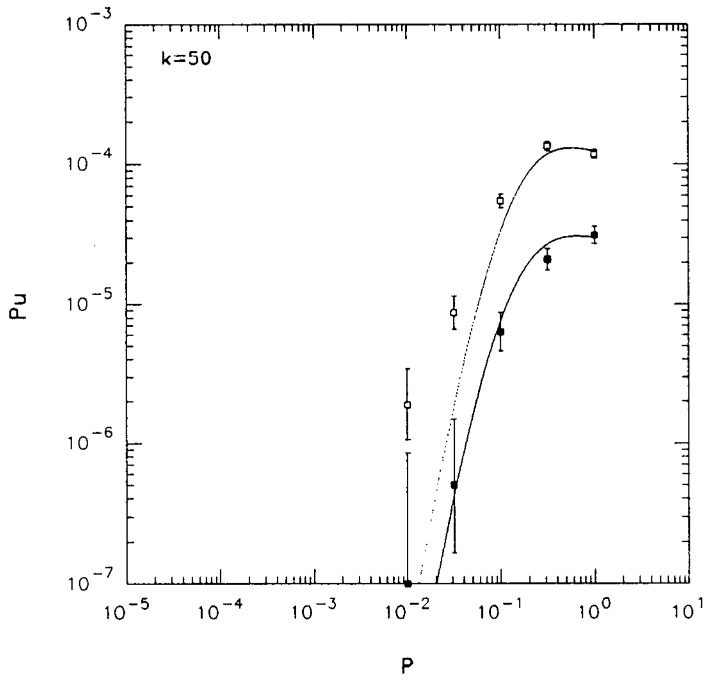


Figure 4.9 Simulation of CRC−16 codes, p=.3, h=.9

reasonable since P is the quantity most like the bit error rate in the BSC, that is, it is usually the small quantity. It is evident from the plots (and even before hand) that the minimum P_u we can hope to simulate reasonably with our sample size is $\gtrsim 10/N=10^{-6}$. This corresponds to the range $10^{-4} \lesssim P \lesssim 10^{-2}$ depending on the values of p and h. In spite of the limitations, a number of features in the plots are evident. Firstly, CRC-CCITT appears to maintain its superiority over CRC-ANSI over all parameter values. The difference between them is small for p=0.01, h=.5, but that difference increases to more than an order of magnitude difference in P_u with increasing p and h. This is reasonable since for small p, the tendency is for the channel to stay in the B state if it gets there, and if h=.5, which is the random vector limit where all bit strings are equally likely, the two codes have the same probability of undetected error $1/2^{16}$. Secondly, the qualitative behaviour of both codes for k=25 is similar to k=50. Quantatively, the k=50 codes reach the plateau at $1/2^{16}$ for smaller values of P which is just as expected. Thirdly, the general behaviour of both CRC-ANSI and CRC-CCITT are most like their BSC effective counterparts when p=.3 where a maximum at approximately 10^{-4} is also evident. This is expected since for large p, the mean time that the system stays in the B state is short so the correlation between errors is small. Finally, note that there are ranges of parameter values where the Gilbert channel P_u are quite different (order of magnitude or more) from the BSC effective values. For instance, when p=.01 and h=.5 and $10^{-4} \leq P \leq 10^{-2}$, P_u is consistently overestimated by the BSC effective values; and perhaps more significantly, the opposite is true for the region p=.01, h=.9, and $P \leq 10^{-3}$. These observations point out that while some of the properties of CRC codes on the BSC may persist over the Gilbert channel, there are certain parameter value ranges where interpolation is not possible, presumably because of fundamental differences between the channels.

5. Conclusions

We have examined in this thesis the general problem of computing the probability of undetected error for binary linear block codes on the Gilbert channel. Little on the subject is known to date so we have tried different approaches.

The accuracy of the code evaluation technique using the P(m,n) distribution is difficult to predict as we have seen even in limited observations on short BCH codes. Certainly, precise numerical results are not possible. The fact that equivalent codes, and in the case of cyclic codes, codes which have different generator polynomials but same weight distribution, give rise to different P_u is an additional albeit interesting complication. Perhaps the role of this approximation technique should be that of preliminary or cursory investigation. In that case, the efficient computation of P(m,n) which we have developed can be utilized most advantageously.

Even though exact analysis in the Gilbert channel is quite difficult in general, it is probably one of the few channel models other than the BSC where there is a realistic chance to obtain analytical results. We have made some exploratory attempts in this direction and have obtained a closed form expression for the single parity check code. The identification of the channel's underlying symmetries also allowed us to derive two results on cyclic codes which can be used to facilitate the computation of P_u.

The results of our simulation of CRC codes give us impetus for further study. For example, it is clear that the P_u for a code on a Gilbert Channel can be quite different from that on a BSC. In addition, we have observed some properties of CRC codes which are themselves of practical and theoretical interest, especially in comparison with the corresponding BSC results. On the subject of the simulation itself, the standard Monte Carlo technique which we used is the most rudimentary, and therefore not very efficient computationally. Other techniques for simulation can make use of partial analytical information and involve more specific details than standard Monte Carlo [19]. Perhaps one of these alternate methods can be adapted for use on the Gilbert channel.

Finally, though we have not considered it here, it should be worthwhile to examine the related problem of error correction [2]. On the Gilbert channel, neighbouring codewords which are the same Hamming distance away from a given vector can have different a posteriori probabilities. Therefore, it would be interesting to examine for instance, the effectivness of the nearest neighbour decoding rule (which is optimal on the BSC) or the modification of this rule to improve performance.

References

[1] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*, North-Holland, Amsterdam, 1977

[2] S. Lin and D. J. Costello, *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, New Jersey, 1983

[3] V. Pless, *Introduction to the Theory of Error-Correcting Codes*, Wiley, New York, 1989

[4] C. Leung, E. R. Barnes, and D. U. Friedman, "On some properties of the undetected error probability of linear codes," *I.E.E.E. Trans. Info. Theory*, vol. 22, pp. 235-237, Mar. 1979

[5] J. K. Wolf, A. M. Michelson, and A. H. Levesque, "On the probability of undetected error probability of linear codes," *I.E.E.E. Trans. Commun.*, vol. 30, pp. 317-324, Feb. 1982

[6] T. Kasami, T. Klove, and S. Lin, "Linear block codes for error detection," *I.E.E.E. Trans. Info. Theory*, vol. 29, pp. 131-136, Jan. 1983

[7] C. Leung, "Evaluation of the undetected error probability of single-parity check product codes," *IEEE Trans. Commun.*, vol. 31, pp. 250-253, Feb. 1983

[8] T. Fujiwara, T. Kasami, A. Kitai, and S. Lin, "On the undetected error probability for shortened Hamming codes," *IEEE Trans. Commun.*, vol. 33, pp. 570-574, June 1985

[9] K. A. Witzke and C. Leung, "A comparison of some error detecting CRC code standards," *I.E.E.E. Trans. Commun.*, vol. 33, pp. 996-998, Sept. 1985

[10] K. A. Witzke, "Examination of the undetected error probability of linear block codes," *MASc Thesis*, Univ. of British Columbia, Aug. 1984

[11] C. T. Ong, "On the undetected error probability of linear codes," *MASc Thesis*, Univ. of British Columbia, Mar. 1990

[12] E. N. Gilbert, "Capacity of a burst-noise channel," *Bell Sys. Tech. J.*, vol. 39, pp. 1253-1265, Sept. 1960

[13] A. A. Alexander, R. M. Gryb, and D. W. Nast, "Capability of the telephone network for data transmission," *Bell Sys. Tech. J.*, vol. 39, pp. 431-476, May 1960

[14] E. O. Elliott, "Estimates of error rates for codes on burst-noise channels," *Bell Sys. Tech. J.*, vol. 42, pp. 1977-1997, Sept. 1963

[15] W. Feller, *An Introduction to Probability Theory and Its Applications*, vol. 1, Wiley, New York, 1968

[16] E. O. Elliott, "A Model of the switched telephone network for data communications," *Bell Sys. Tech. J.*, vol. 44, pp. 89-109, Jan. 1965

[17] V. Cuperman, "An upper bound for the error probability on the Gilbert channel," *I.E.E.E. Trans. Commun. Tech.*, vol. 17, pp. 532-535, Oct. 1969

[18] J. B. Cain and R. S. Simpson, "The distribution of burst lengths on a Gilbert channel," *I.E.E.E. Trans. Info. Theory*, vol. 15, pp. 624-627, Sept. 1969

[19] M. C. Jeruchim, "Techniques for estimating the bit error rate in the simulation of digital communication systems," *IEEE J. Select. Areas Commun.*, vol. 2, pp. 153-170, Jan. 1984

[20] L. N. Kanal and A. R. K. Sastry, "Models for channels with memory and their applications to error control," *Proc. IEEE*, vol. 66, pp. 724-744, July, 1978

[21] D. Bertsekas and R. Gallager, *Data Networks*, Prentice-Hall, New Jersey, 1987

Appendix A

The derivation of (2.6) is presented here for more than just the sake of completeness as there is apparent confusion regarding the range of m for which the equation is valid in [17].

The generating function (see (3.9)) of the $P_0(m,n)$ distribution relative to n for $m \geq 1$ is given by [16,17]

$$\sum_{n=m}^{\infty} P_0(m,n)\, z^n = \frac{P}{P+p}\, z^m\, [q-z(q-P)]^{m-1}\, (1-zQ)^{-m-1}\, [1-z(q-P)]^2 \ . \tag{A.1}$$

Expanding the binomials and then gathering factors of z gives

$$\sum_{n=m}^{\infty} P_0(m,n)\, z^n = \frac{P}{P+p} \sum_{i=0}^{\infty} \sum_{j=0}^{m-1} \sum_{t=0}^{2} c_j^{m-1} c_i^{m+i} c_t^2\, q^{m-j-1} Q^i (P-q)^{j+t}\, z^{i+j+t+m} \ . \tag{A.2}$$

The coefficient of z^n on the right hand side is just $P_0(m,n)$, hence

$$P_0(m,n) = \frac{P}{P+p} \sum_{\substack{0 \leq j \leq m-1 \\ 0 \leq t \leq 2 \\ j+t \leq n-m}} c_j^{m-1} c_{n-m-j-t}^{n-j-t} c_t^2\, q^{m-j-1} Q^{n-m-j-t} (P-q)^{j+t} \ . \tag{A.3}$$

There are three cases to consider depending on the value of n-m. For n-m=0, (A.3) reduces to

$$P_0(n,n) = \frac{P}{P+p}\, q^{n-1} \tag{A.4}$$

For n-m=1, (A.3) reads

$$P_0(m,n) = \frac{P}{P+p} \sum_{\substack{0 \leq j \leq m-1 \\ 0 \leq t \leq 2 \\ j+t \leq 1}} c_j^{m-1} c_{n-m-j-t}^{n-j-t} c_t^2\, q^{m-j-1} Q^{n-m-j-t} (P-q)^{j+t} \tag{A.5}$$

but $n\text{-}m\text{-}j\text{-}t=1\text{-}j\text{-}t<0$ if $j+t>1$ while $n\text{-}j\text{-}t=m+1\text{-}j\text{-}t\geq 0$ so the $c^{n-j-t}_{n-m-j-t}$ factor in the summand is zero for $j+t>1$ hence (A.5) is the same as

$$P_0(m,n)=\frac{P}{P+p}\sum_{j=0}^{m-1}\sum_{t=0}^{2} c^{m-1}_{j}c^{n-j-t}_{n-m-j-t}c^{2}_{t}\, q^{m-j-1}Q^{n-m-j-t}(P-q)^{j+t}\quad . \tag{A.6}$$

For $n\text{-}m\geq 2$, (A.3) becomes

$$P_0(m,n)=\frac{P}{P+p}\sum_{t=0}^{2}\sum_{j=0}^{\min(m-1,n-m-t)} c^{m-1}_{j}c^{n-j-t}_{n-m-j-t}c^{2}_{t}\, q^{m-j-1}Q^{n-m-j-t}(P-q)^{j+t} \tag{A.7}$$

and there are three subcases to consider. If $m\text{-}1\leq n\text{-}m\text{-}2$, the upper limit of the summation over j is m-1 so (A.7) becomes identical to the expression in (A.6). If $m\text{-}1\geq n\text{-}m$, however

$$\begin{aligned}P_0(m,n)&=\frac{P}{P+p}\sum_{t=0}^{2}\sum_{j=0}^{n-m-t} c^{m-1}_{j}c^{n-j-t}_{n-m-j-t}c^{2}_{t}\, q^{m-j-1}Q^{n-m-j-t}(P-q)^{j+t}\\&=\frac{P}{P+p}\sum_{t=0}^{2}\left\{\sum_{j=0}^{m-t}-\sum_{j=n-m-t+1}^{m-1}\right\} c^{m-1}_{j}c^{n-j-t}_{n-m-j-t}c^{2}_{t}\, q^{m-j-1}Q^{n-m-j-t}(P-q)^{j+t}\quad .\end{aligned} \tag{A.8}$$

The second sum inside the braces (which should be ignored for $m\text{-}1=n\text{-}m$, $t=0$) is zero since $n\text{-}j\text{-}t\text{-}m\leq n\text{-}(n\text{-}m\text{-}t+1)\text{-}t\text{-}m=\text{-}1$ and $n\text{-}j\text{-}t\geq n\text{-}(m\text{-}1)\text{-}t=(n\text{-}m)+1\text{-}t\geq 3\text{-}t\geq 1$ means that $c^{n-j-t}_{n-m-j-t}$ is zero in the summand. Hence (A.8) is also the same as (A.6). Finally, if $m\text{-}1=n\text{-}m\text{-}1$, from (A.7)

$$\begin{aligned}P_0(m,n)&=\frac{P}{P+p}\sum_{t=0}^{2}\sum_{j=0}^{\min(m-1,m-t)} c^{m-1}_{j}c^{n-j-t}_{n-m-j-t}c^{2}_{t}\, q^{m-j-1}Q^{n-m-j-t}(P-q)^{j+t}\\&=\frac{P}{P+p}\left\{\sum_{t=0}^{1}\sum_{j=0}^{m-1}+\sum_{t=2}^{2}\sum_{j=0}^{m-2}\right\} c^{m-1}_{j}c^{n-j-t}_{n-m-j-t}c^{2}_{t}\, q^{m-j-1}Q^{n-m-j-t}(P-q)^{j+t}\\&=\frac{P}{P+p}\left\{\sum_{t=0}^{2}\sum_{j=0}^{m-1} c^{m-1}_{j}c^{n-j-t}_{n-m-j-t}c^{2}_{t}\, q^{m-j-1}Q^{n-m-j-t}(P-q)^{j+t}-c^{m-1}_{-1}\right\}\end{aligned} \tag{A.9}$$

so that (A.6) is again obtained since $c^{m-1}_{-1}=0$.

Consider now m=0. $P_0(0,n)$ is just the probability of being in state G for all n transmission intervals. For this to occur, the system must be initially in state G (with probability p_G) and remain there (with probability Q) for the next n-1 intervals hence

$$P_0(0,n) = \frac{p}{P+p} Q^{n-1} \ . \tag{A.10}$$

Summarizing for the various cases considered above, we have the result given in (2.6), namely

$$P_0(m,n) = \begin{cases} \dfrac{p}{P+p} Q^{n-1} & ; \ m=0 \\ \dfrac{P}{P+p} \sum\limits_{j=0}^{m-1} \sum\limits_{t=0}^{2} c^{m-1}_{j} c^{n-j-t}_{n-m-j-t} c^{2}_{t} \, q^{m-j-1} Q^{n-m-j-t} (P-q)^{j+t} & ; \ 1 \le m \le n-1 \\ \dfrac{P}{P+p} q^{n-1} & ; \ m=n \ . \end{cases} \tag{A.11}$$

Appendix B

Here we evaluate the sums required for the computation of $P_0(m,n;r)$ in Chapter 2. We assume in the below that $a \geq b+1$ and $b \geq 1$ so the following are certainly well defined:

$$U(a,b,c,d,e) = \sum_{j=0}^{b-1} \frac{(-1)^j (a-j-2)!}{(a-b-c-j)!(j-d)!(b-e-1-j)!} \tag{B.1}$$

$$V(a,b,c,d) = \sum_{j=0}^{b-1} \frac{(-1)^j (a-j-2)!}{(a-b-c-j)!(j-d)!(b-j-1)!} \tag{B.2}$$

$$W(a,b,c) = \sum_{j=0}^{b-1} \frac{(-1)^j (a-j-2)!}{(a-b-c-j)!j!(b-j-1)!} \cdot \tag{B.3}$$

By shifting the range of the summation variable, we can obtain relationships between the above quantities. Thus

$$\begin{aligned}
U(a,b,c,d,e) &= \sum_{j=0}^{b-1} \frac{(-1)^j (a-j-2)!}{(a-b-c-j)!(j-d)![b-1-(j+e)]!} \\
&= (-1)^e \sum_{j=e}^{b+e-1} \frac{(-1)^j [(a+e)-j-2]!}{[(a+e)-b-c-j]![j-(d+e)]!(b-1-j)!} \\
&= (-1)^e \left\{ \sum_{j=0}^{b-1} - \sum_{j=0}^{e-1} \right\} \frac{(-1)^j [(a+e)-j-2]!}{[(a+e)-b-c-j]![j-(d+e)]!(b-1-j)!} \quad \text{if } e \geq 1 \\
&= (-1)^e\, U(a+e,b,c,d+e,0) \quad \text{if } d \geq 0
\end{aligned} \tag{B.4}$$

so that

$$\begin{aligned}
U(a,b,c,d,e) &= (-1)^e\, U(a+e,b,c,d+e,0) \;\; ; \;\; d \geq 0,\, e \geq 0 \\
&= (-1)^e\, V(a+e,b,c,d+e) \,.
\end{aligned} \tag{B.5}$$

Also

$$V(a,b,c,d) = (-1)^d \sum_{j=-d}^{(b-d)-1} \frac{(-1)^j[(a-d)-j-2]!}{[(a-d)-(b-d)-(c+d)-j]!j![(b-d)-j-1]!}$$

$$= \begin{cases} 0 & ;\ d \geq 0,\ b-d \leq 0 \\ (-1)^d \displaystyle\sum_{j=0}^{(b-d)-1} \frac{(-1)^j[(a-d)-j-2]!}{[(a-d)-(b-d)-(c+d)-j]!j![(b-d)-j-1]!} & ;\ d \geq 0,\ b-d \geq 1 \end{cases}$$

$$= \begin{cases} 0 & ;\ d \geq 0,\ b-d \leq 0 \\ (-1)^d\, W(a-d,b-d,c+d) & ;\ d \geq 0,\ b-d \geq 1 \end{cases} \tag{B.6}$$

From standard references, for instance on pp. 62 of [15],

$$W(a,b,c) = \frac{1}{(c-1)!} \prod_{j=1}^{c-1} (a-b-j)(b-1+j) \ \ ; \ a \geq 2,\ b \geq 1,\ c \geq 2\ . \tag{B.7}$$

We can apply (B.7) to the last line in (B.6) if $a-d \geq 2$, $b-d \geq 1$, $c+d \geq 2$. The first condition is redundant since we always take $a \geq b+1$. Hence for $b-d \geq 1$, $c+d \geq 2$, $d \geq 0$,

$$V(a,b,c,d) = (-1)^d \frac{1}{(c+d-1)!} \prod_{j=1}^{c+d-1} (a-b-j)(b-d-1+j)\ . \tag{B.8}$$

However, note that in fact (B.8) holds even for $b-d \leq 0$ if $c \geq 1$ since then $b-d-1+1=b-d \leq 0$ and $b-d-1+(c+d-1)=b+c-2 \geq 0$ so one of the factors $(b-d-1+j)$ must be zero. That is, we have

$$V(a,b,c,d) = (-1)^d \frac{1}{(c+d-1)!} \prod_{j=1}^{c+d-1} (a-b-j)(b-d-1+j)$$

$$;\ a \geq b+1,\ b \geq 1,\ c \geq 1,\ d \geq 0,\ c+d \geq 2$$

$$= (-1)^d\, (c+d-1)!\ c_{c+d-1}^{a-b-1}\ c_{c+d-1}^{b+c-2}\ . \tag{B.9}$$

We require two further identities regarding V(a,b,c,d). Using the explicit form on the right hand side of (B.9), it is simple to verify that

$$V(a+1,b,c,d) = \frac{a-b}{a-b-c-d+1} V(a,b,c,d) \tag{B.10}$$

and

$$V(a,b,c,d+1) = \frac{-(a-b-c-d)(b-d-1)}{c+d} V(a,b,c,d) \ . \tag{B.11}$$

Appendix C

The following pages contain the source code listings for the C and Maple® programs used in the numerical computations and simulation. A brief description of the purpose of each program follows. The parameters to be entered at runtime and any include files required are specified by comments at the beginning of each listing.

1. p0mn_c.c - C program that computes $P_0(m,n)$ from the closed form summation expression (2.6).

2. pmn_r.c - C program that computes $P(m,n)$ from the recursion relations (2.3) and (2.4).

3. pmn_s.c - C program that computes $P_0^{(r)}(m,n)$ and $P^{(r)}(m,n)$ from (2.5), (2.17), and (2.18).

4. pu.c - C program that computes $P_u(C(n,k))$ by exhaustive computation using explicit codeword generation and (1.3), $E[P_u]$ using (2.2) and the series expansion method of computing $P(m,n)$, and their difference in percentage.

5. mc_crc.c - C program that generates a Monte Carlo simulation for CRC codes according to the procedure in Chapter 4.

6. pmn_c.map - Maple program that computes $P_0(m,n)$ from the closed form summation expression (2.6) and then $P(m,n)$ from (2.5).

7. pmn_s.map - Maple program that computes $P_0^{(r)}(m,n)$ and $P^{(r)}(m,n)$ from (2.5), (2.17), and (2.18).

p0mn_c.c

```
/* p0mn_c.c */
/* Computes each term of P0(m,n) and the total using a closed form summation
   expression given by Cuperman. */

/* parameters prompted for at runtime:
   n = block length
   m = number of errors in block
   P = G to B transition probability
   p = B to G transition probability
   h = B state bit error rate */


#include <stdio.h>
#include <math.h>


/* factorial function */
double fact(n)
int n;
{
  int i;
  double x=1.0;
  for (i=1; i<=n; i++)
    x = x * i;
  return x;
}

/* inverse factorial function */
double invfact(n)
int n;
{
  if (n<0)
    return 0.0;
  else
    return 1.0/fact(n);
}

/* function A(j)=(n-m-j)p+mQ */
double a(m,n,pg,pb,j)
int m, n, j;
double pg, pb;
{
  return (n-m-j)*pb + m*(1-pg);
}

/* jth term in summation expression for P0(m,n) */
double p0(m,n,pg,pb,j)
int m, n, j;
double pg, pb;
{
  return (pg/(pg+pb))*fact(n-j-2)*invfact(j)*invfact(m-j-1)*invfact(n-m-j)
                     *pow(1-pb, (double)(m-j-1))*pow(pg+pb-1, (double)(j))
                     *pow(1-pg, (double)(n-m-j-2))
                     *(a(m,n,pg,pb,j)*a(m,n,pg,pb,j) - pb*a(m,n,pg,pb,j)
                       - m*(1-pg-pb)*(1-pg)) / m;
}
```

```
void main()
{
  int m, n, j;
  double pg, pb, x, y;

  printf("n = ");
  scanf("%d", &n);
  printf("m = ");
  scanf("%d", &m);
  printf("P = ");
  scanf("%lf", &pg);
  printf("p = ");
  scanf("%lf", &pb);
  printf("  j          p0(m,n,j)\n");
  printf("-----     -------------\n");
  x = 0.0;
  for (j=0; j<=m-1; j++)
  {
    y = p0(m,n,pg,pb,j);
    x = x + y;
    printf("%5d     %13e\n", j, y);
  }
  printf("total     %13e\n", x);
  return;
}
```

pmn_r.c

```
/* pmn_r.c */
/* Computes P(m,n) for all m exactly using recurrence relations. */


#include <stdio.h>
#include <math.h>

#define N 1000  /* Define the value of N to be the block length. */


void main()
{
  int i, m, n;
  double x;
  double pg, pb, h;
  double u[N+1], v[N+1], p[N+1];
  double r[N+1][N+1];

  printf("P = ");
  scanf("%lf", &pg);
  printf("p = ");
  scanf("%lf", &pb);
  printf("h = ");
  scanf("%lf", &h);

  u[0] = 1.0;
  u[1] = h + (1-h)*pb;
  for (i=2; i<=N; i++)
  {
    u[i] = (1-pg+h*(1-pb))*u[i-1] + h*(pg+pb-1)*u[i-2];
  }

  v[0] = (1-h)*(1-pb);
  v[1] = (1-h)*(pb*pg+h*(1-pb)*(1-pb));
  for (i=2; i<=N; i++)
  {
    v[i] = (1-pg+h*(1-pb))*v[i-1] + h*(pg+pb-1)*v[i-2];
  }

  for (n=1; n<=N; n++)
  {
    r[1][n] = u[n-1];
    for (m=2; m<=n; m++)
    {
      x = 0.0;
      for (i=1; i<=n-m+1; i++)
        x = x + v[i-1]*r[m-1][n-i];
      r[m][n] = x;
    }
  }

  for (m=1; m<=N; m++)
  {
    x = 0.0;
    for (i=1; i<=N-m+1; i++)
      x = x + (pg/(pg+pb))*(1-h)*u[i-1]*r[m][N-i+1];
    p[m] = x;
  }

  printf("N = %d\n", N);
  printf("   m         P(m,N)\n");
  printf("----     ------------\n");
  for (m=1; m<=N; m++)
    printf("%4.d     %e\n", m, p[m]);

  return;
}
```

pmn_s.c

```
/* pmn_s.c */
/* Calculates P0(m,n) approximately by series expansion in P and then
   the P(m,n) distribution for all m in the Gilbert channel.

/* parameters prompted for at runtime:
   P = G to B transition probability
   p = B to G transition probability
   h = B state bit error rate
   n = block length
   r = number of terms taken in expansion */


#include <stdio.h>
#include <math.h>

#define N 4095  /* Set N to at least n */
#define D 4.0   /* Set D to fix overflow possibility in p(m,n,h,vector). */


/* (-1)^l for integer l. */
int parity(l)
int l;
{
  if ((l/2)*2-l==0)
    return 1;
  else
    return -1;
}

/* Binomial coefficients. */
double binom(m,n)
int m, n;
{
  int i;
  double x=1.0;
  if (n>=0 && m>=0)
  {
    if (n<m)
      return 0.0;
    else
    {
      for (i=0; i<=m-1; i++)
        x = x * (double)(n-i)/(double)(m-i);
      return x;
    }
  }
  if (n>=0 && m<0)
    return 0.0;
  if (n<0 && m>=0)
  {
    for (i=0; i<=m-1; i++)
      x = x * (double)(m-n-1-i)/(double)(m-i);
    return parity(m)*x;
  }
  if (n<0 && m<0)
  {
    if (n<m)
      return 0.0;
    else
    {
      for (i=0; i<=n-m-1; i++)
        x = x * (double)(-m-1-i)/(double)(n-m-i);
      return parity(n-m)*x;
    }
  }
}

/* The rth term in the expansion of P0(m,n)/pG in P. */
double p0r(m,n,pg,pb,r)
int m, n, r;
double pg, pb;
{
  int b, c;
  double x=0.0;
  if (r==0)
  {
    if (m==0)
      return 1.0;
    else
      return 0.0;
  }
  else if (r==1)
       {
         if (m==0)
           return (1-n)*pg;
         if (m>0 && m<n)
           return 2*pg*pow(1-pb,(double)(m-1))*(1+((n-m-1)/2.0)*pb);
         if (m==n)
           return pg*pow(1-pb,(double)(n-1))/pb;
       }
       else
       {
         if (m==0)
         {
           return parity(r)*pow(pg,(double)(r))*binom(r,n-1);
         }
         if (m>0 && m<n)
         {
           for (b=0; b<=r; b++)
             x = x + (n-m-b+1)*(n-m-b)
                   *binom(b,r)*binom(r-1,m+b-1)*pow(pb-1,(double)(b));
           return pow(pg,(double)(r))*pow(1-pb,(double)(m-r))
                  *binom(r-1,n-m)*x/(r*(n-m));
         }
         if (m==n)
           return 0.0;
       }
}

/* P0(m,n) accurate to order r in P. */
double p0(m,n,pg,pb,r)
int m, n;
double pg, pb;
{
  int l;
  double x=0.0;
  for (l=0; l<=r; l++)
```

```
    x = x + p0r(m,n,pg,pb,1);
  return (pb/(pg+pb))*x;
}

/* Sum of (1-h)^m*h^(i-m)*C(m,i)*vector[i] over i=m..n. */
double p(m,n,h,vector)
int m, n;
double h;
double vector[];
{
  int i;
  double x=0.0, y=1.0;

  for (i=m; i<=n; i++)
  {
    x = x + y*vector[i];
    y = (y*h*(i+1)) / (i+1-m);
  }
  return pow((pow(x,1/D)*pow(1-h,m/D)),D);
}


void main()
{
  int m, n, r;
  double pg, pb, h;
  double p0_o[N+1];

  printf("P = ");
  scanf("%lf", &pg);
  printf("p = ");
  scanf("%lf", &pb);
  printf("h = ");
  scanf("%lf", &h);
  printf("n = ");
  scanf("%d", &n);
  printf("r = ");
  scanf("%d", &r);
  printf("  m         P0(m,n)\n");
  printf("-----     ----------\n");
  for (m=0; m<=n; m++)
    p0_o[m] = p0(m,n,pg,pb,r);
  for (m=0; m<=n; m++)
  {
    printf("%5d     ", m);
    printf("%.4e\n", p0_o[m]);
  }
  printf("\n");
  printf("  m         P(m,n)\n");
  printf("-----     ----------\n");
  for (m=0; m<=n; m++)
  {
    printf("%5d     ", m);
    printf("%.4e\n", p(m,n,h,p0_o));
  }
  return;
}
```

pu.c

```
/* pu.c */
/* Calculates occurrence probability of all nonzero codewords of a code
   by exhaustive computation, the average of all equivalents to that
   code using the P(m,n) (approximate) distribution, and their percentage
   difference in the Gilbert channel. */

/* include file required:
   "codegen" - see below for format
/* parameters prompted for at runtime:
   P = G to B transition probability
   p = B to G transition probability
   h = B state bit error rate
   r = number of terms taken in expansion */


/* Sample "codegen" Code Specification File

#define N    7
#define K    4
#define M   16

const char codename[] = "Hamming (7,4), cyclic, 1+z+z^3";

const int gen[K][N] = { 1, 1, 0, 1, 0, 0, 0,
                        0, 1, 1, 0, 1, 0, 0,
                        0, 0, 1, 1, 0, 1, 0,
                        0, 0, 0, 1, 1, 0, 1 };

const int wt[N+1] = { 1, 0, 0, 7, 7, 0, 0, 1 }; */


#include <stdio.h>
#include <math.h>
#include "codegen"


double u1[N+1], v1[N+1], w1[N+1];


/* Binomial coefficients. */
double binom(m,n)
int m, n;
{
  int i;
  double x=1.0;
  if (n>=0 && m>=0)
  {
    if (n<m)
      return 0.0;
    else
    {
      for (i=0; i<=m-1; i++)
        x = x * (double)(n-i)/(double)(m-i);
      return x;
    }
  }
  if (n>=0 && m<0)
    return 0.0;
  if (n<0 && m>=0)
  {
    for (i=0; i<=m-1; i++)
      x = x * (double)(m-n-1-i)/(double)(m-i);
    return parity(m)*x;
  }
  if (n<0 && m<0)
  {
    if (n<m)
      return 0.0;
    else
    {
      for (i=0; i<=n-m-1; i++)
        x = x * (double)(-m-1-i)/(double)(n-m-i);
      return parity(n-m)*x;
    }
  }
}

/* Binary addition routine. */
int xor(i,j)
int i, j;
{
  if (i == j)
    return 0;
  else
    return 1;
}

/* (-1)^l for integer l. */
int parity(l)
int l;
{
  if ((l/2)*2-l==0)
    return 1;
  else
    return -1;
}

/* First root of x^2-(Q+h*q)*x+h*(Q-p)=0. */
double jp(pg,pb,h)
double pg, pb, h;
{
  double b, c;
  b = -(1-pg+h*(1-pb));
  c = h*(1-pg-pb);
  return (-b+sqrt(b*b-4*c))/2;
}

/* Second root of x^2-(Q+h*q)*x+h*(Q-p)=0. */
double jm(pg,pb,h)
double pg, pb, h;
{
  double b, c;
  b = -(1-pg+h*(1-pb));
  c = h*(1-pg-pb);
  return (-b-sqrt(b*b-4*c))/2;
}
```

```
/* Probability of k 0's following a 1. */
double u(k,pg,pb,h)
int k;
double pg, pb, h;
{
  return ( (jp(pg,pb,h)+pg+pb-1)*pow(jp(pg,pb,h),(double)(k))
          -(jm(pg,pb,h)+pg+pb-1)*pow(jm(pg,pb,h),(double)(k)) )
         /(jp(pg,pb,h)-jm(pg,pb,h));
}

/* Probability of k 0's between 1's. */
double v(k,pg,pb,h)
int k;
double pg, pb, h;
{
  return ( ((1-pb)*jp(pg,pb,h)+pg+pb-1)*pow(jp(pg,pb,h),(double)(k))
          -((1-pb)*jm(pg,pb,h)+pg+pb-1)*pow(jm(pg,pb,h),(double)(k)) )
         *(1-h)/(jp(pg,pb,h)-jm(pg,pb,h));
}

/* Probability of k 0's preceding a 1. */
double w(k,pg,pb,h)
int k;
double pg, pb, h;
{
  return ( (jp(pg,pb,h)+pg+pb-1)*pow(jp(pg,pb,h),(double)(k))
          -(jm(pg,pb,h)+pg+pb-1)*pow(jm(pg,pb,h),(double)(k)) )
         *(pg/(pg+pb))*(1-h)/(jp(pg,pb,h)-jm(pg,pb,h));
}

/* Probability of bit pattern s. */
double prob(s)
int s[];
{
  int i;
  int c=0;
  char first_one='Y';
  double x=1.0;
  for (i=0; i<N; i++)
  {
    if (s[i] == 0)
      c++;
    else
    {
      if (first_one == 'Y')
      {
        x = x*w1[c];
        first_one = 'N';
      }
      else
      {
        x = x*v1[c];
      }
      c = 0;
    }
  }
  x = x*u1[c];
  return x;
}
```

```
/* The rth term in the expansion of P0(m,n)/pG in P. */
double p0r(m,n,pg,pb,r)
int m, n, r;
double pg, pb;
{
  int b, c;
  double x=0.0;
  if (r==0)
  {
    if (m==0)
      return 1.0;
    else
      return 0.0;
  }
  else if (r==1)
       {
         if (m==0)
           return (1-n)*pg;
         if (m>0 && m<n)
           return 2*pg*pow(1-pb,(double)(m-1))*(1+((n-m-1)/2.0)*pb);
         if (m==n)
           return pg*pow(1-pb,(double)(n-1))/pb;
       }
       else
       {
         if (m==0)
         {
           return parity(r)*pow(pg,(double)(r))*binom(r,n-1);
         }
         if (m>0 && m<n)
         {
           for (b=0; b<=r; b++)
             x = x + (n-m-b+1)*(n-m-b)
                   *binom(b,r)*binom(r-1,m+b-1)*pow(pb-1,(double)(b));
           return pow(pg,(double)(r))*pow(1-pb,(double)(m-r))
                  *binom(r-1,n-m)*x/(r*(n-m));
         }
         if (m==n)
           return 0.0;
       }
}

/* P0(m,n) accurate to order r in P. */
double p0(m,n,pg,pb,r)
int m, n;
double pg, pb;
{
  int l;
  double x=0.0;
  for (l=0; l<=r; l++)
    x = x + p0r(m,n,pg,pb,l);
  return (pb/(pg+pb))*x;
}

/* Sum of (1-h)^m*h^(i-m)*C(m,i)*vector[i] over i=m..n. */
double p(m,n,h,vector)
int m, n;
double h;
double vector[];
{
  int i;
  double x=0.0;
  for (i=m; i<=n; i++)
    x = x + binom(m,i)*pow(h, (double)(i-m))*vector[i];
  return ((((x*pow(1-h,m/4.0))*pow(1-h,m/4.0))*pow(1-h,m/4.0))*pow(1-h,m/4.0));
}


void main()
{
  char carry;
  int b, c, i, m, r;
  int k=0, n=0;
  int mes[K], cw[N];
  double pg, pb, h;
  double total=0.0, totala=0.0;
  double p0_o[N+1];

  printf("%s\n", codename);
  printf("P = ");
  scanf("%lf", &pg);
  printf("p = ");
  scanf("%lf", &pb);
  printf("h = ");
  scanf("%lf", &h);
  printf("r = ");
  scanf("%d", &r);
  for (m=0; m<=N; m++)
  {
    u1[m] = u(m,pg,pb,h);
    v1[m] = v(m,pg,pb,h);
    w1[m] = (pg/(pg+pb))*(1-h)*u1[m];
  }
  for (k=0; k<K; k++)
    mes[k] = 0;
  for (b=1; b<M; b++)
  {
    carry = 'Y';
    for (k=0; k<K && carry=='Y'; k++)
    {
      if ((mes[k]=xor(mes[k],1)) == 1)
        carry = 'N';
    }
    for (n=0; n<N; n++)
    {
      c = 0;
      for (k=0; k<K; k++)
        c = xor(c,mes[k]*gen[k][n]);
      cw[n] = c;
    }
    total = total + prob(cw);
  }
  for (m=0; m<=N; m++)
    p0_o[m] = p0(m,N,pg,pb,r);
  for (i=1; i<=N; i++)
    totala = totala + (wt[i]/binom(i,N))*p(i,N,h,p0_o);
  printf("Pu = %.4e    ", total);
  printf("Pua = %.4e    ", totala);
  printf("%% diff. = %.4e\n", 100*(totala-total)/total);
  return;
}
```

mc_crc.c

```
/* mc_crc.c */
/* Computes the approximate probability of undetected error for CRC codes
   in the Gilbert channel by Monte Carlo simulation and the 99% confidence
   limits. */

/* include file required:
   "codepar" - see below for format
/* parameters prompted for at runtime:
   P = G to B transition probability
   p = B to G transition probability
   h = B state bit error rate
   N = number of samples */


/* Sample "codepar" Code Specification File
#define N 21
#define P 16

const char codename[] = "CRC-16, 1+z^2+z^15+z^16";

const int g[P+1] = {1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1}; */


#include <stdio.h>
#include <math.h>
#include <values.h>
#include "codepar"


const seed[4] = {1, 2, 3, 4};  /* random number seed values */
const double d99 = 2.575829;

int z[N];
int flag;


/* Generate a random vector z[N], set flag to 1 if it's nonzero. */
void generate(pg,pb,h)
double pg,pb,h;
{
  char state;
  int i;

  flag = 0;
  if (rand() > (pb/(pg+pb))*MAXINT)
    state = 'B';
  else
    state = 'G';
  for (i=0; i<N; i++)
  {
    if (state == 'G')
    {
      z[i]=0;
      if (rand() > (1-pg)*MAXINT)
        state = 'B';
    }
    else
    {
      if (rand() > h*MAXINT)
      {
        z[i] = 1;
        flag = 1;
      }
      else
        z[i] = 0;
      if (rand() > (1-pb)*MAXINT)
        state = 'G';
    }
  }
}

/* Binary addition routine. */
int xor(i,j)
int i, j;
{
  if (i == j)
    return 0;
  else
    return 1;
}

/* Check to see if vector z[N] is a non-zero codeword. */
int check()
{
  int i, j, q;
  int sr[P];

  if (flag == 0)
    return 0;
  for (i=0; i<P; i++)
    sr[P-1-i]=z[i];
  for (i=P; i<N; i++)
  {
    q = sr[P-1];
    for (j=P-1; j>0; j--)
      sr[j]=xor(sr[j-1],g[j]*q);
    sr[0]=xor(z[i],g[0]*q);
  }
  for (i=0; i<P; i++)
  {
    if (sr[i] != 0)
      return 0;
  }
  return 1;
}


void main()
{
  int sample, fseed, s, t;
  int m=0;
  double pg, pb, h, p;
  double ym99, yp99;

  printf("%s\n", codename);
  printf("k = %d\n", N-16);
  printf("P = ");
  scanf("%lf", &pg);
  printf("p = ");
  scanf("%lf", &pb);
  printf("h = ");
  scanf("%lf", &h);
  printf("N = ");
  scanf("%d", &sample);
  for (t=0; t<4; t++)
  {
    srand(seed[t]);
    for (s=0; s<sample/4; s++)
    {
      generate(pg,pb,h);
      if (check() == 1)
        m = m + 1;
    }
  }
  p = ((double)(m)) / sample;
  ym99 = p * (1+((d99*d99)/(2*p*sample))*(1-sqrt((4*p*sample)/(d99*d99)+1)));
  yp99 = p * (1+((d99*d99)/(2*p*sample))*(1+sqrt((4*p*sample)/(d99*d99)+1)));
  printf("Pu = %e\n", p);
  printf("lower error for 99%% confidence = %e\n", p-ym99);
  printf("upper error for 99%% confidence = %e\n", yp99-p);
  return;
}
```

pmn_c.map

```
# pmn_c.map
# Calculates P0(m,n) using the exact closed form expression given by
# Cuperman and then P(m,n).

# procedures:
#  s - computes a term in the sum for P0(m,n)
#  sall - computes all terms in the sum for P0(m,n)
#  p0c - computes P0(m,n)
#  p0call - computes P0(m,n) for all m
#  pc - computes P(m,n)
#  pcall - computes P(m,n) for all m


a := proc(m,n,pg,pb,j)
     (n-m-j)*pb+m*(1-pg) end;

s := proc(m,n,pg,pb,j)
     (pg/(pg+pb))*((1-pb)^(m-j-1))*((1-pg)^(n-m-j-2))*((pg+pb-1)^j)
     *(a(m,n,pg,pb,j)^2-pb*a(m,n,pg,pb,j)-m*(1-pg-pb)*(1-pg))
     *(n-j-2)!/(m*j!*(m-j-1)!*(n-m-j)!) end;

sall := proc(m,n,pg,pb)
        for j from 0 to m-1 do
        print(j,s(m,n,pg,pb,j));
        od;
        end;

p0c := proc(m,n,pg,pb)
       if m=0 then (pb/(pg+pb))*((1-pg)^(n-1))
       elif m<n then sum('s(m,n,pg,pb,j)','j'=0..min(m-1,n-m))
       else (pg/(pg+pb))*((1-pb)^(m-1)) fi end;

fillarray := proc(n,pg,pb)
             p0carr:=array(0..n);
             for m from 0 to n do
             p0carr[m]:=p0c(m,n,pg,pb);
             od;
             end;

p0call := proc(n,pg,pb)
          fillarray(n,pg,pb);
          for m from 0 to n do
          print(m,p0carr[m]);
          od;
          end;

pc := proc(m,n,pg,pb,h)
      fillarray(n,pg,pb);
      y := 0.0;
      for i from m to n do
      y := y + binomial(i,m)*(h^(i-m))*p0carr[i];
      od;
      y*((1-h)^m);
      end;

pcl := proc(m,n,pg,pb,h)
       y := 0.0;
       for i from m to n do
       y := y + binomial(i,m)*(h^(i-m))*p0carr[i];
       od;
       y*((1-h)^m);
       end;

pcall := proc(n,pg,pb,h)
         fillarray(n,pg,pb);
         for m from 0 to n do
         print(m,pcl(m,n,pg,pb,h));
         od;
         end;
```

pmn_s.map

```
# pmn_s.map
# Compute P0(m,n) by series expansion in P and then P(m,n).

# procedures:
#  p0r - computes rth term of series expansion for P0(m,n)
#  p0rall - computes all terms of series expansion for P0(m,n) up to r
#  p0 - computes P0(m,n) up to order r accuracy
#  p0all - computes P0(m,n) up to order r accuracy for all m
#  p - computes P(m,n) to order r accuracy
#  pall - computes P(m,n) to order r accuracy for all m


p0r := proc(m,n,pg,pb,r)
       if r=0 then
                      if m=0 then 1
                      else 0
                      fi
       elif r=1 then
                      if m=0 then (1-n)*pg
                      elif m=n then (pg/pb)*(1-pb)^(n-1)
                      else 2*pg*(1+(n-m-1)*pb/2)*(1-pb)^(m-1)
                      fi
       else
            if m=0 then ((-pg)^r)*binomial(n-1,r)
            elif m=n then 0
            else ((pg^r)*((1-pb)^(m-r))*binomial(n-m,r-1)/(r*(n-m)))
                 *sum('(n-m-b+1)*(n-m-b)*binomial(r,b)*binomial(m+b-1,r-1)
                       *(pb-1)^b','b'=0..r)
            fi
       fi
       end;

p0rall := proc(m,n,pg,pb,r)
          for j from 0 to r do
          print(p0r(m,n,pg,pb,j));
          od;
          end;

p0 := proc(m,n,pg,pb,r)
      x := 0.0;
      for l from 0 to r do
      x := x + p0r(m,n,pg,pb,l);
      od;
      x*(pb/(pg+pb));
      end;

fillarray := proc(n,pg,pb,r)
             p0arr:=array(0..n);
             for m from 0 to n do
             p0arr[m]:=p0(m,n,pg,pb,r);
             od;
             end;

p0all := proc(n,pg,pb,r)
         fillarray(n,pg,pb,r);
         for m from 0 to n do
         print(m,p0arr[m]);
         od;
         end;

p := proc(m,n,pg,pb,h,r)
     fillarray(n,pg,pb,r);
     y := 0.0;
     for i from m to n do
     y := y + binomial(i,m)*(h^(i-m))*p0arr[i];
     od;
     y*((1-h)^m);
     end;

p1 := proc(m,n,pg,pb,h,r)
      y := 0.0;
      for i from m to n do
      y := y + binomial(i,m)*(h^(i-m))*p0arr[i];
      od;
      y*((1-h)^m);
      end;

pall := proc(n,pg,pb,h,r)
        fillarray(n,pg,pb,r);
        for m from 0 to n do
        print(m,p1(m,n,pg,pb,h,r));
        od;
        end;
```