Chapter 1

Introduction

Claude Shannon's 1948 paper "A Mathematical Theory of Communication" gave birth to the twin disciplines of information theory and coding theory. The basic goal is efficient and reliable communication in an uncooperative (and possibly hostile) environment. To be efficient, the transfer of information must not require a prohibitive amount of time and effort. To be reliable, the received data stream must resemble the transmitted stream to within narrow tolerances. These two desires will always be at odds, and our fundamental problem is to reconcile them as best we can.

At an early stage the mathematical study of such questions broke into the two broad areas. Information theory is the study of achievable bounds for communication and is largely probabilistic and analytic in nature. Coding theory then attempts to realize the promise of these bounds by models which are constructed through mainly algebraic means. Shannon was primarily interested in the information theory. Shannon's colleague Richard Hamming had been laboring on error-correction for early computers even before Shannon's 1948 paper, and he made some of the first breakthroughs of coding theory.

Although we shall discuss these areas as mathematical subjects, it must always be remembered that the primary motivation for such work comes from its practical engineering applications. Mathematical beauty can not be our sole gauge of worth. Here we shall concentrate on the algebra of coding theory, but we keep in mind the fundamental bounds of information theory and the practical desires of engineering.

1.1 Basics of communication

Information passes from a source to a sink via a conduit or channel. In our view of communication we are allowed to choose exactly the way information is structured at the source and the way it is handled at the sink, but the behaviour of the channel is not in general under our control. The unreliable channel may take many forms. We may communicate through space, such as talking across

a noisy room, or through time, such as writing a book to be read many years later. The uncertainties of the channel, whatever it is, allow the possibility that the information will be damaged or distorted in passage. My conversation may be drowned out or my manuscript might weather.

Of course in many situations you can ask me to repeat any information that you have not understood. This is possible if we are having a conversation (although not if you are reading my manuscript), but in any case this is not a particularly efficient use of time. ("What did you say?" "What?") Instead to guarantee that the original information can be recovered from a version that is not too badly corrupted, we add redundancy to our message at the source. Languages are sufficiently repetitive that we can recover from imperfect reception. When I lecture there may be noise in the hallway, or you might be unfamiliar with a word I use, or my accent could confuse you. Nevertheless you have a good chance of figuring out what I mean from the context. Indeed the language has so much natural redundancy that a large portion of a message can be lost without rendering the result unintelligible. When sitting in the subway, you are likely to see overhead and comprehend that "IF U CN RD THS U CN GT A JB."

Communication across space has taken various sophisticated forms in which coding has been used successfully. Indeed Shannon, Hamming, and many of the other originators of mathematical communication theory worked for Bell Telephone Laboratories. They were specifically interested in dealing with errors that occur as messages pass across long telephone lines and are corrupted by such things as lightening and crosstalk. The transmission and reception capabilities of many modems are increased by error handling capability embedded in their hardware. Deep space communication is subject to many outside problems like atmospheric conditions and sunspot activity. For years data from space missions has been coded for transmission, since the retransmission of data received faultily would be very inefficient use of valuable time. A recent interesting case of deep space coding occurred with the Galileo mission. The main antenna failed to work, so the possible data transmission rate dropped to only a fraction of what was planned. The scientists at JPL reprogrammed the onboard computer to do more code processing of the data before transmission, and so were able to recover some of the overall efficiency lost because of the hardware malfunction.

It is also important to protect communication across time from inaccuracies. Data stored in computer banks or on tapes is subject to the intrusion of gamma rays and magnetic interference. Personal computers are exposed to much battering, so often their hard disks are equipped with "cyclic redundancy checking" CRC to combat error. Computer companies like IBM have devoted much energy and money to the study and implementation of error correcting techniques for data storage on various mediums. Electronics firms too need correction techniques. When Phillips introduced compact disc technology, they wanted the information stored on the disc face to be immune to many types of damage. If you scratch a disc, it should still play without any audible change. (But you probably should not try this with your favorite disc; a really bad scratch can cause problems.) Recently the sound tracks of movies, prone to film

breakage and scratching, have been digitized and protected with error correction techniques.

There are many situations in which we encounter other related types of communication. Cryptography is certainly concerned with communication, however the emphasis is not on efficiency but instead upon security. Nevertheless modern cryptography shares certain attitudes and techniques with coding theory.

With source coding we are concerned with efficient communication but the environment is not assumed to be hostile; so reliability is not as much an issue. Source coding takes advantage of the statistical properties of the original data stream. This often takes the form of a dual process to that of coding for correction. In data compaction and compression[1] redundancy is removed in the interest of efficient use of the available message space. Data compaction is a form of source coding in which we reduce the size of the data set through use of a coding scheme that still allows the perfect reconstruction of the original data. Morse code is a well established example. The fact that the letter "e" is the most frequently used in the English language is reflected in its assignment to the shortest Morse code message, a single dot. Intelligent assignment of symbols to patterns of dots and dashes means that a message can be transmitted in a reasonably short time. (Imagine how much longer a typical message would be if "e" was represented instead by two dots.) Nevertheless, the original message can be recreated exactly from its Morse encoding.

A different philosophy is followed for the storage of large graphic images where, for instance, huge black areas of the picture should not be stored pixel by pixel. Since the eye can not see things perfectly, we do not demand here perfect reconstruction of the original graphic, just a good likeness. Thus here we use data compression, "lossy" data reduction as opposed to the "lossless" reduction of data compaction. The subway message above is also an example of data compression. Much of the redundancy of the original message has been removed, but it has been done in a way that still admits reconstruction with a high degree of certainty. (But not perfect certainty; the intended message might after all have been nautical in thrust: "IF YOU CANT RIDE THESE YOU CAN GET A JIB.")

Although cryptography and source coding are concerned with valid and important communication problems, they will only be considered tangentially here.

One of the oldest forms of coding for error control is the adding of a parity check bit to an information string. Suppose we are transmitting strings composed of 26 bits, each a 0 or 1. To these 26 bits we add one further bit that is determined by the previous 26. If the initial string contains an even number of 1's, we append a 0. If the string has an odd number of 1's, we append a 1. The resulting string of 27 bits always contains an even number of 1's, that is, it has even parity. In adding this small amount of redundancy we have not compromised the information content of the message greatly. Of our 27 bits, 26 of them carry information. But we now have some error handling ability.

[1]We follow Blahut by using the two terms compaction and compression in order to distinguish lossless and lossy compression.

If an error occurs in the channel, then the received string of 27 bits will have odd parity. Since we know that all transmitted strings have even parity, we can be sure that something has gone wrong and react accordingly, perhaps by asking for retransmission. Of course our error handling ability is limited to this possibility of detection. Without further information we are not able to guess the transmitted string with any degree of certainty, since a received odd parity string can result from a single error being introduced to any one of 27 different strings of even parity, each of which might have been the transmitted string. Furthermore there may have actually been more errors than one. What is worse, if two bit errors occur in the channel (or any even number of bit errors), then the received string will still have even parity. We may not even notice that a mistake has happened.

Can we add redundancy in a different way that allows us not only to detect the presence of bit errors but also to decide which bits are likely to be those in error? The answer is yes. If we have only two possible pieces of information, say 0 for "by sea" and 1 for "by land," that we wish to transmit, then we could repeat each of them three times — 000 or 111 . We might receive something like 101 . Since this is not one of the possible transmitted patterns, we can as before be sure that something has gone wrong; but now we can also make a good guess at what happened. The presence of two 1's but only one 0 points strongly to a transmitted string 111 plus one bit error (as opposed to 000 with two bit errors). Therefore we guess that the transmitted string was 111. This "majority vote" approach to decoding will result in a correct answer provided at most one bit error occurs.

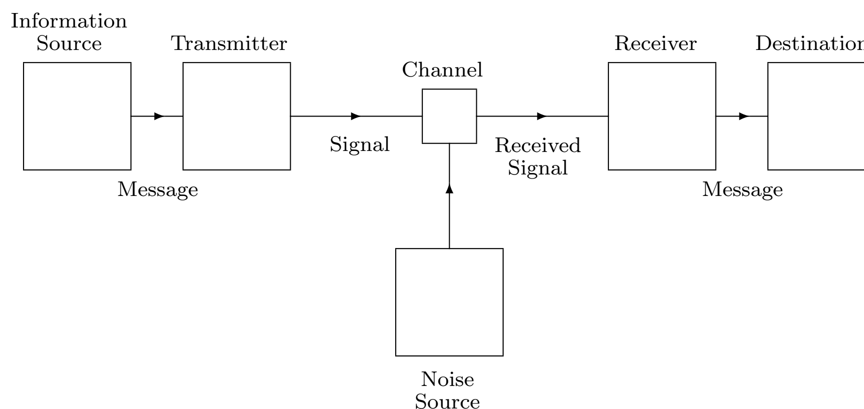
Now consider our channel that accepts 27 bit strings. To transmit each of our two messages, 0 and 1, we can now repeat the message 27 times. If we do this and then decode using "majority vote" we will decode correctly even if there are as many as 13 bit errors! This is certainly powerful error handling, but we pay a price in information content. Of our 27 bits, now only one of them carries real information. The rest are all redundancy.

We thus have two different codes of length 27 — the parity check code which is information rich but has little capability to recover from error and the repetition code which is information poor but can deal well even with serious errors. The wish for good information content will always be in conflict with the desire for good error performance. We need to balance the two. We hope for a coding scheme that communicates a decent amount of information but can also recover from errors effectively. We arrive at a first version of

> **The Fundamental Problem** — Find codes with both reasonable information content and reasonable error handling ability.

Is this even possible? The rather surprising answer is, "Yes!" The existence of such codes is a consequence of the Channel Coding Theorem from Shannon's 1948 paper (see Theorem 2.3.2 below). Finding these codes is another question. Once we know that good codes exist we pursue them, hoping to construct practical codes that solve more precise versions of the Fundamental Problem. This is the quest of coding theory.

Figure 1.1: Shannon's model of communication



1.2 General communication systems

We begin with Shannon's model of a general communication system, Figure 1.2. This setup is sufficiently general to handle many communication situations. Most other communication models, such as those requiring feedback, will start with this model as their base.

Our primary concern is block coding for error correction on a discrete memoryless channel. We next describe these and other basic assumptions that are made here concerning various of the parts of Shannon's system; see Figure 1.2. As we note along the way, these assumptions are not the only ones that are valid or interesting; but in studying them we will run across most of the common issues of coding theory. We shall also honor these assumptions by breaking them periodically.

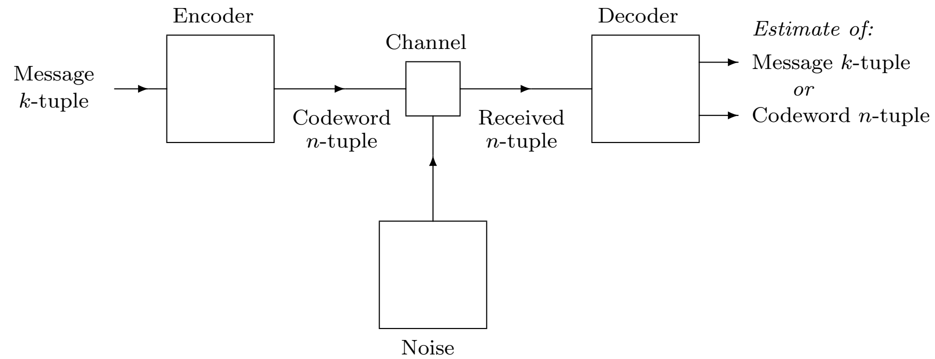
We shall usually speak of the transmission and reception of the words of the code, although these terms may not be appropriate for a specific envisioned application. For instance, if we are mainly interested in errors that affect computer memory, then we might better speak of storage and retrieval.

1.2.1 Message

Our basic assumption on messages is that each possible message k-tuple is as likely to be selected for broadcast as any other.

We are thus ignoring the concerns of source coding. Perhaps a better way to say this is that we assume source coding has already been done for us. The original message has been source coded into a set of k-tuples, each equally likely. This is not an unreasonable assumption, since lossless source coding is designed to do essentially this. Beginning with an alphabet in which different

Figure 1.2: A more specific model



letters have different probabilities of occurrence, source coding produces more compact output in which frequencies have been levelled out. In a typical string of Morse code, there will be roughly the same number of dots and dashes. If the letter "e" was mapped to two dots instead of one, we would expect most strings to have a majority of dots. Those strings rich in dashes would be effectively ruled out, so there would be fewer legitimate strings of any particular reasonable length. A typical message would likely require a longer encoded string under this new Morse code than it would with the original. Shannon made these observations precise in his Source Coding Theorem which states that, beginning with an ergodic message source (such as the written English language), after proper source coding there is a set of source encoded k-tuples (for a suitably large k) which comprises essentially all k-tuples and such that different encoded k-tuples occur with essentially equal likelihood.

1.2.2 Encoder

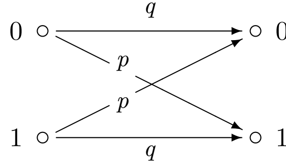
block coding

We are concerned here with *block coding*. That is, we transmit blocks of symbols of fixed length n from a fixed alphabet A. These blocks are the codewords, and that codeword transmitted at any given moment depends only upon the present message, not upon any previous messages or codewords. Our encoder has no memory. We also assume that each codeword from the code (the set of all possible codewords) is as likely to be transmitted as any other.

Some work has been done on codes over mixed alphabets, that is, allowing the symbols at different coordinate positions to come from different alphabets. Such codes occur only in isolated situations, and we shall not be concerned with them at all.

Convolutional codes, trellis codes, lattice codes, and others come from en-

Figure 1.3: The Binary Symmetric Channel



coders that have memory. We lump these together under the heading of *convolutional codes*. The message string arrives at the decoder continuously rather than segmented into unrelated blocks of length k, and the code string emerges continuously as well. That n-tuple of code sequence that emerges from the encoder while a given k-tuple of message is being introduced will depend upon previous message symbols as well as the present ones. The encoder "remembers" earlier parts of the message. The coding most often used in modems is of convolutional type.

convolutional codes

1.2.3 Channel

As already mentioned, we shall concentrate on coding on a *discrete memoryless channel* or DMC. The channel is discrete because we shall only consider finite alphabets. It is memoryless in that an error in one symbol does not affect the reliability of its neighboring symbols. The channel has no memory, just as above we assumed that the encoder has no memory. We can thus think of the channel as passing on the codeword symbol-by-symbol, and the characteristics of the channel can described at the level of the symbols.

discrete memoryless channel
DMC

An important example is furnished by the m-ary symmetric channel. The *m-ary symmetric channel* has input and output an alphabet of m symbols, say $x_1, \ldots, x_m$. The channel is characterized by a single parameter p, the probability that after transmission of any symbol x_j the particular symbol $x_i \neq x_j$ is received. That is,

m-ary symmetric channel

$$p = \text{Prob}(x_i \,|\, x_j), \text{ for } i \neq j\,.$$

Related are the probability

$$s = (m-1)p$$

that after x_j is transmitted it is not received correctly and the probability

$$q = 1 - s = 1 - (m-1)p = \text{Prob}(x_j \,|\, x_j))$$

that after x_j is transmitted it is received correctly. We write $m\text{SC}(p)$ for the m-ary symmetric channel with *transition probability* p. The channel is symmetric in the sense $\text{Prob}(x_i \,|\, x_j)$ does not depend upon the actual values of i and j but only on whether or not they are equal. We are especially interested in the 2-ary symmetric channel or binary symmetric channel $\text{BSC}(p)$ (where $p = s$).

$m\text{SC}(p)$
transition probability
$\text{BSC}(p)$

Of course the signal that is actually broadcast will often be a measure of some frequency, phase, or amplitude, and so will be represented by a real (or complex)

number. But usually only a finite set of signals is chosen for broadcasting, and the members of a finite symbol alphabet are modulated to the members of the finite signal set. Under our assumptions the modulator is thought of as part of the channel, and the encoder passes symbols of the alphabet directly to the channel.

Gaussian channel

There are other situations in which a continuous alphabet is the most appropriate. The most typical model is a *Gaussian channel* which has as alphabet an interval of real numbers (bounded due to power constraints) with errors introduced according to a Gaussian distribution.

The are also many situations in which the channel errors exhibit some kind of memory. The most common example of this is burst errors. If a particular symbol is in error, then the chances are good that its immediate neighbors are also wrong. In telephone transmission such errors occur because of lightening and crosstalk. A scratch on a compact disc produces burst errors since large blocks of bits are destroyed. Of course a burst error can be viewed as just one type of random error pattern and be handled by the techniques that we shall develop. We shall also see some methods that are particularly well suited to dealing with burst errors.

One final assumption regarding our channel is really more of a rule of thumb. We should assume that the channel machinery that carries out modulation, transmission, reception, and demodulation is capable of reproducing the transmitted signal with decent accuracy. We have a

Reasonable Assumption — Most errors that occur are not severe.

Otherwise the problem is more one of design than of coding. For a DMC we interpret the reasonable assumption as saying that an error pattern composed of a small number of symbol errors is more likely than one with a large number. For a continuous situation such as the Gaussian channel, this is not a good viewpoint since it is nearly impossible to reproduce a real number with perfect accuracy. All symbols are likely to be received incorrectly. Instead we can think of the assumption as saying that whatever is received should resemble to a large degree whatever was transmitted.

1.2.4 Received word

We assume that the decoder receives from the channel an n-tuple of symbols from the transmitter's alphabet A.

This assumption could be included in our discussion of the channel, since it really concerns the demodulator, which we think of as part of the channel just as we do the modulator. Many implementations combine the demodulator with the decoder in a single machine. This is the case with computer modems which serve as encoder/modulator and demodulator/decoder (MOdulator-DEModulator).

Think about how the demodulator works. Suppose we are using a binary alphabet which the modulator transmits as signals of amplitude $+1$ and -1. The demodulator receives signals whose amplitudes are then measured. These

received amplitudes will likely not be exactly $+1$ or -1. Instead values like .750, and $-.434$ and .003 might be found. Under our assumptions each of these must be translated into a $+1$ or -1 before being passed on to the decoder. An obvious way of doing this is to take positive values to $+1$ and negative values to -1, so our example string becomes $+1, -1, +1$. But in doing so, we have clearly thrown away some information which might be of use to the decoder. Suppose in decoding it becomes clear that one of the three received symbols is certainly not the one originally transmitted. Our decoder has no way of deciding which one to mistrust. But if the demodulator's knowledge were available, the decoder would know that the last symbol is the least reliable of the three while the first is the most reliable. This improves our chances of correct decoding in the end.

In fact with our assumption we are asking the demodulator to do some initial, primitive decoding of its own. The requirement that the demodulator make precise (or hard) decisions about code symbols is called *hard quantization*. The alternative is *soft quantization*. Here the demodulator passes on information which suggests which alphabet symbol might have been received, but it need not make a final decision. At its softest, our demodulator would pass on the three real amplitudes and leave all symbol decisions to the decoder. This of course involves the least loss of information but may be hard to handle. A mild but still helpful form of soft quantization is to allow channel *erasures*. The channel receives symbols from the alphabet A but the demodulator is allowed to pass on to the decoder symbols from $A \cup \{\mathbf{?}\}$, where the special symbol "**?**" indicates an inability to make an educated guess. In our three symbol example above, the decoder might be presented with the string $+1, -1, \mathbf{?}$, indicating that the last symbol was received unreliably. It is sometimes helpful to think of an erasure as a symbol error whose location is known.

hard quantization

soft quantization

erasures

1.2.5 Decoder

Suppose that in designing our decoding algorithms we know, for each n-tuple $\mathbf{y}$ and each codeword $\mathbf{x}$, the probability $p(\mathbf{y}|\mathbf{x})$ that $\mathbf{y}$ is received after the transmission of $\mathbf{x}$. The basis of our decoding is the following principle:

> **Maximum Likelihood Decoding** — When $\mathbf{y}$ is received, we must decode to a codeword $\mathbf{x}$ that maximizes $\mathrm{Prob}(\mathbf{y} \mid \mathbf{x})$.

We often abbreviate this to **MLD**. While it is very sensible, it can cause problems similar to those encountered during demodulation. Maximum likelihood decoding is "hard" decoding in that we must always decode to some codeword. This requirement is called *complete decoding*.

The alternative to complete decoding is *incomplete decoding*, in which we either decode a received n-tuple to a codeword or to a new symbol ∞ which could be read as "errors were detected but were not corrected" (sometimes abbreviated to "error detected"). Such *error detection* (as opposed to correction) can come about as a consequence of a *decoding default*. We choose this default alternative when we are otherwise unable (or unwilling) to make a sufficiently reliable decoding choice. For instance, if we were using a binary repetition code

MLD

complete decoding

incomplete decoding

error detection

decoding default

of length 26 (rather than 27 as before), then majority vote still deals effectively with 12 or fewer errors; but 13 errors produces a 13 to 13 tie. Rather than make an arbitrary choice, it might be better to announce that the received message is too unreliable for us to make a guess. There are many possible actions upon default. Retransmission could be requested. There may be other "nearby" data that allows an undetected error to be estimated in other ways. For instance, with compact discs the value of the uncorrected sound level can be guessed to be the average of nearby values. (A similar approach can be take for digital images.) We will often just declare "error detected but not corrected."

IMLD

Almost all the decoding algorithms that we discuss in detail will not be **MLD** but will satisfy **IMLD**, the weaker principle:

> **Incomplete Maximum Likelihood Decoding** — When $\mathbf{y}$ is received, we must decode either to a codeword $\mathbf{x}$ that maximizes $\mathrm{Prob}(\mathbf{y}\,|\,\mathbf{x})$ or to the "error detected" symbol $\boldsymbol{\infty}$.

decoder error

decoder failure

Of course, if we are only interested in maximizing our chance of successful decoding, then any guess is better than none; and we should use **MLD**. But this longshot guess may be hard to make, and if we are wrong then the consequences might be worse than accepting but recognizing failure. When correct decoding is not possible or advisable, this sort of error detection is much preferred over making an error in decoding. A *decoder error* has occurred if $\mathbf{x}$ has been transmitted, $\mathbf{y}$ received and decoded to a codeword $\mathbf{z} \neq \mathbf{x}$. A decoder error is much less desirable than a decoding default, since to the receiver it has the appearance of being correct. With detection we know something has gone wrong and can conceivably compensate, for instance, by requesting retransmission. Finally *decoder failure* occurs whenever we do not have correct decoding. Thus decoder failure is the combination of decoding default and decoder error.

Consider a code C in A^n and a decoding algorithm $\mathbf{A}$. Then $\mathcal{P}_{\mathbf{x}}(\mathbf{A})$ is defined as the error probability (more properly, failure probability) that after $\mathbf{x} \in C$ is transmitted, it is received and not decoded correctly using $\mathbf{A}$. We then define

$$\mathcal{P}_C(\mathbf{A}) = |C|^{-1} \sum_{\mathbf{x}\in C} \mathcal{P}_{\mathbf{x}}(\mathbf{A})\,,$$

error expectation $\mathcal{P}_C$

the average error expectation for decoding C using the algorithm $\mathbf{A}$. This judges how good $\mathbf{A}$ is as an algorithm for decoding C. (Another good gauge would be the worst case expectation, $\max_{\mathbf{x}\in C} \mathcal{P}_{\mathbf{x}}(\mathbf{A})$.) We finally define the *error expectation* $\mathcal{P}_C$ for C via

$$\mathcal{P}_C = \min_{\mathbf{A}} \mathcal{P}_C(\mathbf{A})\,.$$

If $\mathcal{P}_C(\mathbf{A})$ is large then the algorithm is not good. If $\mathcal{P}_C$ is large, then no decoding algorithm is good for C; and so C itself is not a good code. In fact, it is not hard to see that $\mathcal{P}_C = \mathcal{P}_C(\mathbf{A})$, for every **MLD** algorithm $\mathbf{A}$. (It would be more consistent to call $\mathcal{P}_C$ the failure expectation, but we stick with the common terminology.)

We have already remarked upon the similarity of the processes of demodulation and decoding. Under this correspondence we can think of the detection

symbol ∞ as the counterpart to the erasure symbol **?** while decoder errors correspond to symbol errors. Indeed there are situations in concatenated coding where this correspondence is observed precisely. Codewords emerging from the "inner code" are viewed as symbols by the "outer code" with decoding error and default becoming symbol error and erasure as described.

A main reason for using incomplete rather than complete decoding is efficiency of implementation. An incomplete algorithm may be much easier to implement but only involve a small degradation in error performance from that for complete decoding. Again consider the length 26 repetition code. Not only are patterns of 13 errors extremely unlikely, but they require different handling than other types of errors. It is easier just to announce that an error has been detected at that point, and the the algorithmic error expectation $\mathcal{P}_C(\mathbf{A})$ only increases by a small amount.

1.3 Some examples of codes

1.3.1 Repetition codes

These codes exist for any length n and any alphabet A. A message consists of a letter of the alphabet, and it is encoded by being repeated n times. Decoding can be done by plurality vote, although it may be necessary to break ties arbitrarily.

The most fundamental case is that of binary repetition codes, those with alphabet $A = \{0, 1\}$. Majority vote decoding always produces a winner for binary repetition codes of odd length. The binary repetition codes of length 26 and 27 were discussed above.

1.3.2 Parity check and sum-0 codes

Parity check codes form the oldest family of codes that have been used in practice. The parity check code of length n is composed of all binary (alphabet $A = \{0, 1\}$) n-tuples that contain an even number of 1's. Any subset of $n - 1$ coordinate positions can be viewed as carrying the information, while the remaining position "checks the parity" of the information set. The occurrence of a single bit error can be detected since the parity of the received n-tuple will be odd rather than even. It is not possible to decide where the error occurred, but at least its presence is felt. (The parity check code is able to correct single erasures.)

The parity check code of length 27 was discussed above.

A versions of the parity check code can be defined in any situation where the alphabet admits addition. The code is then all n-tuples whose coordinate entries sum to 0. When the alphabet is the integers modulo 2, we get the usual parity check code.

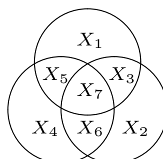
1.3.3 The $[7, 4]$ binary Hamming code

We quote from Shannon's paper:

> An efficient code, allowing complete correction of [single] errors and transmitting at the rate C [$=4/7$], is the following (found by a method due to R. Hamming):
>
> Let a block of seven symbols be $X_1, X_2, \ldots, X_7$ [each either 0 or 1]. Of these X_3, X_5, X_6, and X_7 are message symbols and chosen arbitrarily by the source. The other three are redundant and calculated as follows:
>
> $$\begin{aligned} X_4 \text{ is chosen to make } \alpha &= X_4 + X_5 + X_6 + X_7 \text{ even} \\ X_2 \text{ is chosen to make } \beta &= X_2 + X_3 + X_6 + X_7 \text{ even} \\ X_1 \text{ is chosen to make } \gamma &= X_1 + X_3 + X_5 + X_7 \text{ even} \end{aligned}$$
>
> When a block of seven is received, α, β, and γ are calculated and if even called zero, if odd called one. The binary number $\alpha\,\beta\,\gamma$ then gives the subscript of the X_i that is incorrect (if 0 then there was no error).

This describes a $[7,4]$ binary Hamming code together with its decoding. We shall give the general versions of this code and decoding in a later chapter.

R.J. McEliece has pointed out that the $[7,4]$ Hamming code can be nicely thought of in terms of the usual Venn diagram:



The message symbols occupy the center of the diagram, and each circle is completed to guarantee that it contains an even number of 1's (has even parity). If, say, received circles A and B have odd parity but circle C has even parity, then the symbol within $A \cap B \cap \overline{C}$ is judged to be in error at decoding.

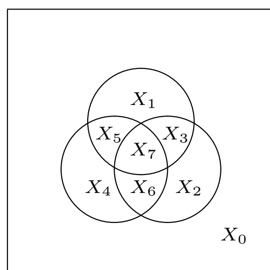
1.3.4 An extended binary Hamming code

An extension of a binary Hamming code results from adding at the beginning of each codeword a new symbol that checks the parity of the codeword. To the $[7,4]$ Hamming code we add an initial symbol:

$$X_0 \text{ is chosen to make } X_0 + X_1 + X_2 + X_3 + X_4 + X_5 + X_6 + X_7 \text{ even}$$

The resulting code is the $[8,4]$ extended Hamming code. In the Venn diagram the symbol X_0 checks the parity of the universe.

The extended Hamming code not only allows the correction of single errors (as before) but also detects double errors.



1.3.5 The $[4, 2]$ ternary Hamming code

This is a code of nine 4-tuples $(a, b, c, d) \in A^4$ with ternary alphabet $A = \{0, 1, 2\}$. Endow the set A with the additive structure of the integers modulo 3. The first two coordinate positions a, b carry the 2-tuples of information, each pair $(a, b) \in A^2$ exactly once (hence nine codewords). The entry in the third position is sum of the previous two (calculated, as we said, modulo 3):

$$a + b = c \ ,$$

for instance, with $(a, b) = (1, 0)$ we get $c = 1 + 0 = 1$. The final entry is then selected to satisfy

$$b + c + d = 0 \ ,$$

so that $0 + 1 + 2 = 0$ completes the codeword $(a, b, c, d) = (1, 0, 1, 2)$. These two equations can be interpreted as making ternary parity statements about the codewords; and, as with the binary Hamming code, they can then be exploited for decoding purposes. The complete list of codewords is:

$$\begin{array}{ccc} (0,0,0,0) & (1,0,1,2) & (2,0,2,1) \\ (0,1,1,1) & (1,1,2,0) & (2,1,0,2) \\ (0,2,2,2) & (1,2,0,1) & (2,2,1,0) \end{array}$$

(1.3.1) PROBLEM. *Use the two defining equations for this ternary Hamming code to describe a decoding algorithm that will correct all single errors.*

1.3.6 A generalized Reed-Solomon code

We now describe a code of length $n = 27$ with alphabet the field of real number $\mathbb{R}$. Given our general assumptions this is actually a nonexample, since the alphabet is not discrete or even bounded. (There are, in fact, situations where these generalized Reed-Solomon codes with real coordinates have been used.)

Choose 27 distinct real numbers $\alpha_1, \alpha_2, \ldots, \alpha_{27}$. Our message k-tuples will be 7-tuples of real numbers $(f_0, f_1, \ldots, f_6)$, so $k = 7$. We will encode a given message 7-tuple to the codeword 27-tuple

$$\mathbf{f} = (f(\alpha_1), f(\alpha_2), \ldots, f(\alpha_{27}))\,,$$

where

$$f(x) = f_0 + f_1x + f_2x^2 + f_3x^3 + f_4x^4 + f_5x^5 + f_6x^6$$

is the polynomial function whose coefficients are given by the message. Our Reasonable Assumption says that a received 27-tuple will resemble the codeword transmitted to a large extent. If a received word closely resembles each of two codewords, then they also resemble each other. Therefore to achieve a high probability of correct decoding we would wish pairs of codewords to be highly dissimilar.

The codewords coming from two different messages will be different in those coordinate positions i at which their polynomials $f(x)$ and $g(x)$ have different values at α_i. They will be equal at coordinate position i if and only if α_i is a root of the difference $h(x) = f(x) - g(x)$. But this can happen for at most 6 values of i since $h(x)$ is a nonzero polynomial of degree at most 6. Therefore:

> distinct codewords differ in at least 21 $(= 27 - 6)$ coordinate positions.

Thus two distinct codewords are highly different. Indeed as many up to 10 errors can be introduced to the codeword $\mathbf{f}$ for $f(x)$ and the resulting word will still resemble the transmitted codeword $\mathbf{f}$ more than it will any other codeword.

The problem with this example is that, given our inability in practice to describe a real number with arbitrary accuracy, when broadcasting with this code we must expect almost all symbols to be received with some small error — 27 errors every time! One of our later objectives will be to translate the spirit of this example into a more practical setting.