



Error-Correcting Codes

Information is stored and exchanged in the form of streams of characters from some alphabet. An *alphabet* is a finite set of symbols, such as the *lower-case Roman alphabet* $\{a, b, c, \dots, z\}$. Larger digits may be formed by including the upper-case Roman letters, punctuation marks, the digits 0 through 9, and possibly other symbols such as ‘\$’, ‘%’, etc. In the other extreme, we may choose the simplest possible alphabet, the *binary alphabet* $\{0, 1\}$. We prefer the binary alphabet for several reasons:

1. its ease of implementation as the on/off state of an electric circuit, the North/South polarization of positions on a magnetic tape or disk, etc.;
2. the fact that letters of any alphabet can be easily represented as strings of 0’s and 1’s; and
3. access to powerful algebraic tools for encoding and decoding.

Strings of letters from our chosen alphabet are called *words* or, when the binary alphabet is in use, *bitstrings*.

All such information is subject to corruption due to imperfect storage media (dust, scratches or manufacturing defects on optical CD’s; demagnetizing influences on magnetic tape or disks) or noisy transmission channels (electromagnetic static in telephone lines or atmosphere). A bit error occurs when a 0 is changed to a 1, or a 1 to a 0, due to such influences. The goal of error-correcting codes is to protect information against such errors. Thanks to error-correcting codes, we can expect that a copy of a copy of a copy of a copy of an audio CD will sound exactly like the original when played back (at least 99.999% of the time); or that a large binary file downloaded off the internet will be a perfect copy of the original (again, 99.999% of the time).

What makes such recovery of the original binary file possible? Roughly, an error-correcting code adds redundancy to a message. Without this added redundancy, no error-correction would be possible. The trick, however, is to add as little redundancy as necessary, since longer messages are more costly to transmit. Finding the optimal balance between achieving a high error-correction and keeping the size of the encoded message as small as possible (i.e. achieving a high information rate, to be defined later) is one of the prime concerns of coding theory. These concepts are best explained through examples.

Suppose we wish to send a bitstring of length 4, i.e. one of the sixteen possible message words 0000, 0001, 0010, 0011, ..., 1111. We refer to these sixteen strings as *message words*, and the (If a message were more than 4 bits in length, it could be divided into blocks of 4 bits each, which could then be encoded and sent individually.) Note that these bitstrings are the binary representations of the integers 0, 1, 2, ..., 15; they also correspond to the hexadecimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Scheme 1: “As is”

One possibility is that we send each bitstring “as is”: for example, the message 1011 would be sent as 1011. This scheme does not allow for any error correction, and so it is only practical for a noiseless channel. We call this scheme a *0-error correcting code*.

The *information rate* of this code is $4/4 = 100\%$, since no redundant information has been included in the message; all bits transmitted are directly interpreted as information.

Scheme 2: Parity Check Bit

As a second example, consider appending a parity check bit to the end of each message word. This last bit is chosen so that each codeword has an even number of 1’s. For example, the message 1011 would be encoded as 10111; the message 0110 would be encoded as 01100; see Table A. This parity check bit allows for the detection of a single bit error during transmission; however this error cannot be corrected.

For example, if the word 10111 is received, this would be accepted as a valid codeword and would be decoded as 1011. If the word 11010 is received, no decoding is possible and this word would be rejected. In practice the receiver would ask the sender to resend the message if possible.

We call this code a *1-error detecting code*. Its information rate is $4/5 = 80\%$, meaning that 80% of the bits transmitted contain the desired information; the remaining 20% of bits transmitted serve only for detecting the occurrence of bit errors.

Scheme 3: A 3-Repetition Code

In order to allow for correction of up to one bit error, we consider the possibility of sending each bit three times. Under this scheme, the message word 0100 would be encoded as the codeword 000111000000 of length 12. Suppose this word is transmitted and that, due to a single bit error during transmission, the word 000111001000 is received. Each triple of bits is decoded according to a ‘majority rules’ principle, thus 000 yields 0; 111 yields 1; 001 yields 0; and 000 yields 0, so the original message word 0100 is recovered despite the bit error introduced during transmission. Some patterns of 2-bit errors may also be safely corrected (those where the two bits affected occur in distinct bit triples). But 2-bit errors are in general not correctable; accordingly we refer to this scheme as a *1-error correcting code*. Since it simply repeats each message bit 3 times, it is known as a 3-repetition code.

The information rate of this code is $4/12 = 1/3 = 33\frac{1}{3}\%$, meaning that 1/3 of the bits transmitted carry all the information; the remaining bits carry redundancy useful only in the error-correction process.

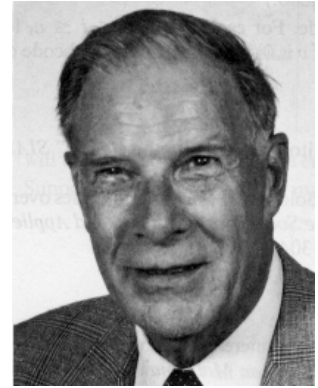
Table A: Four Schemes for Encoding of 4-bit Message Words

Msg. No.	Message Text	Scheme 1 (“As Is”) Codeword	Scheme 2 (Parity Check) Codeword	Scheme 3 (3-Repetition) Codeword	Scheme 4 (Hamming) Codeword
0	0000	0000	00000	000000000000	0000000
1	0001	0001	00011	000000000111	0001111
2	0010	0010	00101	000000111000	0010110
3	0011	0011	00110	000000111111	0011001
4	0100	0100	01001	000111000000	0100101
5	0101	0101	01010	000111000111	0101010
6	0110	0110	01100	000111111000	0110011
7	0111	0111	01111	000111111111	0111100
8	1000	1000	10001	111000000000	1000011
9	1001	1001	10010	111000000111	1001100
10	1010	1010	10100	111000111000	1010101
11	1011	1011	10111	111000111111	1011010
12	1100	1100	11000	111111000000	1100110
13	1101	1101	11011	111111000111	1101001
14	1110	1110	11101	111111111000	1110000
15	1111	1111	11110	111111111111	1111111

Scheme 4: The Hamming Code

Finally we consider a scheme that corrects errors, yet is more efficient than the repetition code. In this scheme every 4-bit message word is encoded as a 7-bit codeword according to Table A. Note that we have simply appended three bits to every message word; the rule for choosing these bits is more complicated than the rule used in Scheme 2 but will be revealed later. Under this scheme, the codeword for message 1011 is 1011010. Suppose this word suffers from a single bit error transmission, and is received as 1010010, say. This word will be safely recognized as 1011010 (and decoded as 1011) since all other Hamming codewords differ from 1010010 in at least two positions. The property of the Hamming code which guarantees unique decodability with at most

one bit error, is that any two Hamming codewords differ from in each other in at least three positions. This 1-error correcting code was discovered by Richard Hamming, 1915–1998, a pioneer in the theory of error-correcting codes who was primarily interested in their application to early electronic computers for achieving fault-tolerant computation.



Richard Hamming
1915–1998

Remarkably, this code has an information rate of $4/7 = 57\%$, which greatly exceeds the information rate of the 3-repetition code, while still allowing for the correction of single bit errors.

The one shortfall of our presentation of the Hamming code is its apparently ad hoc description, and the presumed need to look up codewords in a complicated table. We will soon see that the encoding and decoding can be done much more efficiently than this, using some simple linear algebra. This is significant since if error-correcting codes are to be useful, they should not only allow for error correction in principle, as well as having a high information rate; they should also have simple encoding and decoding algorithms. This means that it should be possible for simple electronic circuits, implemented on silicon chips perhaps, to perform the encoding and decoding easily and in ‘real time’.

Matrix Multiplication

The linear algebra we need to understand the encoding and decoding processes involves matrices. An $m \times n$ matrix is simply an array of numbers, having m rows and n columns, usually enclosed in brackets or parentheses; thus for example

$$A = \begin{bmatrix} 3 & 4 & -2 \\ 2 & -1 & 0 \end{bmatrix}$$

is a 2×3 matrix. It has six *entries* 3, 4, ..., 0 which are located by row and column number; for example the (1,3)-entry of A is -2 .

How do we multiply two matrices? Consider a 3×2 matrix

$$B = \begin{bmatrix} 1 & 2 \\ -1 & 3 \\ 5 & 0 \end{bmatrix}.$$

The product of these two matrices is the 2×2 matrix

$$AB = \begin{bmatrix} 3 & 4 & -2 \\ 2 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ -1 & 3 \\ 5 & 0 \end{bmatrix} = \begin{bmatrix} -11 & 18 \\ 3 & 1 \end{bmatrix}.$$

Note that the (i,j) -entry of AB is the dot product of the i th row of A with the j th row of B ; for example the $(2,1)$ -entry of AB is $2 \times 2 + (-1) \times (-1) + 0 \times 5 = 3$. Note however that the product BA is different from AB :

$$BA = \begin{bmatrix} 1 & 2 \\ -1 & 3 \\ 5 & 0 \end{bmatrix} \begin{bmatrix} 3 & 4 & -2 \\ 2 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 7 & 2 & -2 \\ 3 & -7 & 2 \\ 15 & 20 & -10 \end{bmatrix}.$$

The product of an $m \times n$ matrix with an $n \times p$ matrix, will always give an $m \times p$ matrix. Each entry will be found by taking the dot product of two vectors of length n . The product of two matrices is not defined unless the number of columns in the first matrix, equals the number of rows in the second matrix. Although matrix multiplication is not commutative in general (we have seen an example where $AB \neq BA$), it is always associative: $(AB)C = A(BC)$ whenever the matrix products are defined (i.e. the number of columns of A equals the number of rows of B , and the number of columns of B equals the number of rows of C).

Hamming Encoding and Decoding using Matrices

Encoding and decoding with the Hamming code is accomplished using matrix multiplication *modulo 2*: here the only constants are 0 and 1, using the addition and multiplication tables supplied here:

+	0	1
0	0	1
1	1	0

×	0	1
0	0	0
1	0	1

For encoding we use the 4×7 matrix

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

A message word x of length 4 may be considered as a vector of length 4, or equivalently, a 1×4 matrix. The codeword corresponding to x is then simply xG , which is a 1×7 matrix, or simply a bitstring of length 7. For example the message $x = 1101$ is encoded as

$$xG = [1 \quad 1 \quad 0 \quad 1] \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} = [1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1].$$

Note that this gives the same answer as Table A, namely 1101001, for the codeword corresponding to the message word 1101. The point is that matrix multiplication is easier to implement in an

electronic circuit, and requires less real time to implement, than lookup in a list such as Table A. Moreover this gives us insight into the structure of the Hamming code, using the tools of linear algebra.

How can we efficiently decode? If a word y of length 7 is received, we anticipate first checking to see if y is a codeword; if so, the original message is recovered as the first 4 bits of y . But how do we check to see if y is in the code without performing a const-intensive search through Table A? Our answer uses the 3×7 check matrix

$$H = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}.$$

Consider the Hamming codeword 1101001, which we denote by y , thus:

$$y = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Note that we have written y as a column vector (i.e. as a 7×1 matrix) rather than as a row vector (i.e. a 1×7 matrix). Now the matrix product

$$Hy = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

gives the zero vector, which is our evidence that y is a valid codeword, and so we take its first four characters 1101 to recover the original message word.

What if y had suffered from a single bit error during transmission? Suppose that its third bit had been altered, so that instead of y , we receive the word

$$y' = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

The bit error would be detected by computing the matrix product

$$Hy' = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}.$$

Since the result is not the zero vector, this alerts us to the fact that y' is not a valid codeword. This alerts us to the presence of a bit error, and we assume that only one bit error occurred during transmission. But how can we tell which of the seven bits of y' is in error? Simply: the vector above is the word 011, which is the binary representation of the number 3; this tells us that the third bit of y' is erroneous. Switching it recovers the valid codeword 1101001, then taking the first four bits recovers the codeword 1101.

The vector Hy is called the *syndrome* (or *error syndrome*) of the vector y . If the syndrome is zero, then y is a codeword; otherwise the syndrome represents one of the integers $1, 2, \dots, 7$ in binary, and this tells us which of the seven bits of y to switch to recover a valid Hamming codeword from y .

Sphere Packing

The problem of finding good error-correcting codes can be viewed as the problem of packing spheres. It has long been recognized that the densest possible packing of disks of equal area in the plane, is the packing seen in Figure B:



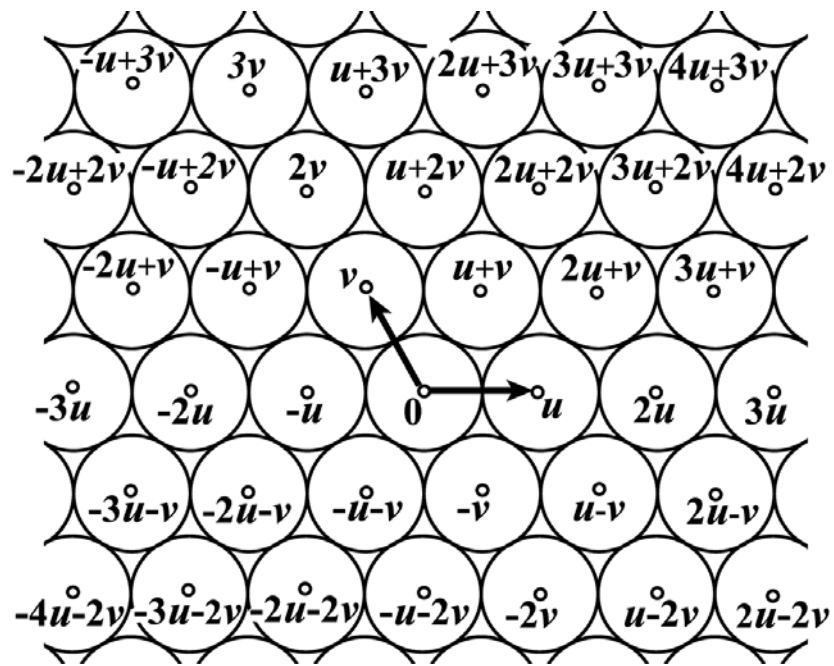
Figure A:
Loosely packed pennies



Figure B:
Densely packed pennies

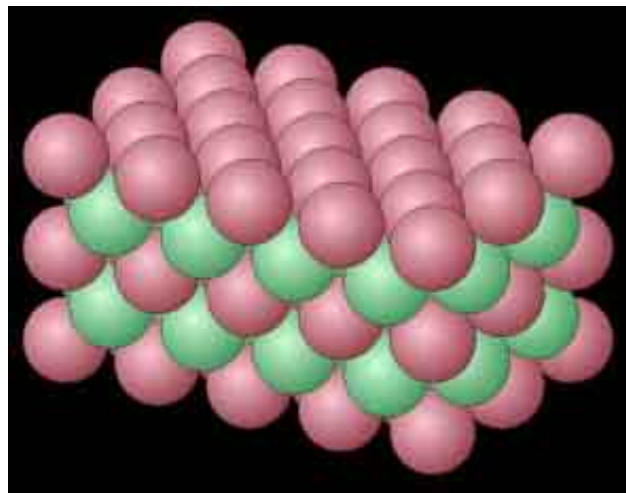
We require that the disks do not overlap, and we want to fill as many as possible into a given large plane region. The main observation to draw from this picture is that the centers of the disks form a lattice in the plane, by which we mean the set of points of the form $au+bv$ where a and b are integers, and u, v are vectors representing two of the disks bordering a fixed disk centered at the origin, as shown in Figure C:

Figure C:
Lattice packing
in 2 dimensions



The regularity of this arrangement is summarized by the following rule: for any two disks in this arrangement, if the vectors corresponding to their centers are added (using the usual parallelogram law for addition of vectors in the plane), the resulting vector is the center of another disk in the packing. There is a similar familiar lattice packing of equal-sized balls in 3 dimensions, shown in Figure D:

Figure D:
Lattice packing
in 3 dimensions



It was shown only recently (by Hales, in 1997) that this packing is in fact the densest possible packing of space by equal-sized balls. For every $n = 1, 2, 3, 4, \dots$, we may ask what is the densest possible packing of equal-sized balls in Euclidean space of n dimensions. For $n > 3$ this problem is open, but the problem is intimately related to the problem of constructing good error-correcting codes. For example the Hamming

code described above is the consequence of a surprisingly dense packing of balls in 8-dimensional space.

We explain the connection between sphere-packing and construction of good codes, using the Hamming code of length 7 as an example. In this case we may view the codewords as points in a 7-dimensional space, albeit a *discrete* space with coordinates 0,1 rather than real number coordinates (so that this space has only $2^7=128$ points in all). Note that points in this space are the same as vectors, or bitstrings, or binary words, of length 7. The *distance* between two of these points is defined as the number of coordinates in which they differ (for example, the distance between the Hamming codewords 0001111 and 0010110 is three). The distance between two different Hamming codewords is always at least 3 (in fact, this distance is always 3, 4 or 7). It is this property of the Hamming code that guarantees that single bit errors are correctible. Heuristically, the fact that the codewords are far apart (distance at least 3 apart) means that they are not easily confused with each other without several bit errors. The large number of codewords (16 is the maximum possible number of binary words of length 7 at minimum distance 3) guarantees the high information rate of the code. We imagine a ball of radius 1 centered at each codeword; this gives a dense packing of our discrete space with balls. The decoding algorithm amounts to taking any point of this 7-dimensional space, and locating the center of the ball in this packing. The regularity of the Hamming code is expressible by the fact that if any two Hamming codewords are added (modulo 2), we get another Hamming codeword. This property (not coincidentally) reflects the property of the familiar dense packings of disks in the Euclidean plane, or balls in Euclidean 3-space, that the sum of any two centers of disks or balls gives the center of another disk or ball. Intuition suggests that this regularity is a requirement if we are to have a dense packing, and indeed most good codes, as well as most known dense sphere packings in higher dimensions, have this property.