Digital data processing

So far we have discussed techniques broadly belonging to the field of numerical analysis ("number-crunching" in the vernacular), for solving problems that arise in physical contexts. Now we turn to computational methods that arise in the context of experimental technique: data transmission—specifically, error analysis, prevention and correction; data compression; and data encryption.

Physicists and astronomers often use computers to collect and analyze data from their measuring instruments. A modern experiment in high energy physics can have event rates as high as $10^7 - 10^8$/sec. Each such datum may be represented by 8192 bytes/particle (representing, *e.g.*, the particle's trajectory through a 256-plane spark chamber of 16×16 wires/plane)—and there may be an average of 10 particles seen per event. That is, a 3-month run may generate 637 gigabytes of data, enough to fill 1000 CD-ROM disks, for example; or 100 high-capacity hard drives. Analyzing these data might take several years using a dedicated 500 MHz computer.

But the vast majority of the data are uninteresting, in the sense that they represent processes that are well measured and well understood. They are not the reason an experiment costing from two to ten million dollars was carried out. The "interesting" events might occur as infrequently as once per ten million uninteresting events, *i.e.* one (or fewer) per second. The problem the experimentalist faces is how to winnow all that chaff for the few worthwhile wheat-grains of interesting data, in real time, so as to reduce at least 100-fold the storage and data-processing requirements of his magnum opus. That is, he must design an apparatus capable of deciding, in the available time of 10-100 nanoseconds, which of the events are worth storing, and which to ignore.

1. Analog to digital conversion

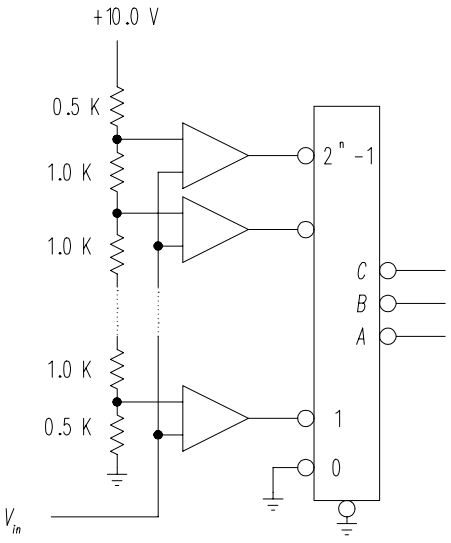
Although the world may be quantized at heart, many of the measurements we make are analog[1] in character. In order to analyze such data by digital methods they must first be represented in digital form—that is, as streams of 0's and 1's rather than as continuously varying voltage levels. Often this conversion must be accomplished at high speed. A typical example is the telemetry of visual data—images gathered by a satellite or other remote probe. The image on a photographic plate consists of the deposition of grains of metallic silver at varying densities. The spatial resolution is limited only by the average size of these grains. Where the grains actually lie on the film is of course random. Thus an image scanner must measure intensity and spatial location, and digitize that. The data gathered by an electronic equivalent—an array of photocells, an orthicon tube or a channel plate—on the other hand, is already digitized to an extent, since the device is configured as a regular

1. The use of the terms "digital" and "analog" to characterize the results of counting experiments (like public opinion sampling or radioactive decay rates) *vs.* measurements of continuously varying quantities (like voltages or angular velocities) dates back to the earliest days of computing and the two types of computer then in use.

2-dimensional array of pixels. That is, the coordinates of the pixels are integers. However, both the light levels and the wavelength distributions at each pixel are analog.

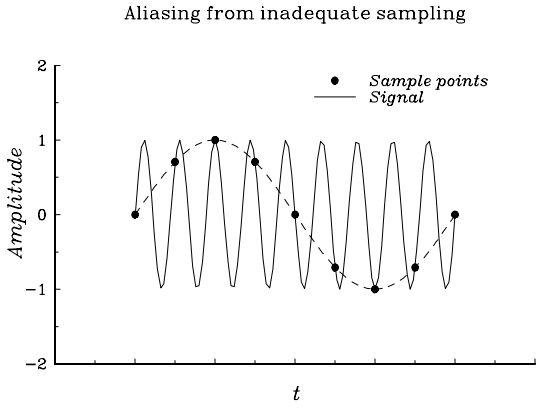
How do we go about digitizing analog information? Usually this is done with electronic circuits called comparators[2]. They come in several varieties—a simple one is the "flash" comparator, so called because of its speed. A circuit for one is shown to the right. The triangular objects are op-amps connected as comparators: they produce an output signal only if the input voltage exceeds the reference voltage. Thus the first m inputs to the encoder chip will be "high" if the input voltage lies in the range

$$m\frac{V_{ref}}{2^n} \le V_{in} < (m+1)\frac{V_{ref}}{2^n}.$$



The encoder is simply a device that converts the integers in the range $[0, 2^n - 1]$ to their binary representation. Thus if we want to be able to convert input voltages with a step-size of ⅛ of the input range, 3 output bits are needed. The entire circuit is called an analog to digital converter, or ADC. Many other types of ADC have been invented. See Ref. 2 for details.

The speed of the ADC is of some interest when converting a time-varying signal to digital form. For example sounds in the audio range—speech or music, for example—are typically complex waveforms with frequency components up to 20 KHz. A basic criterion (Nyquist) states that in order to represent adequately the input it must be sampled at least twice that fast, *i.e.* at intervals less than 0.000025 sec. This is a useful rule of thumb to keep in mind, as we see from the example to the right: if we do not sample often enough, a signal of lower frequency goes through the same set of points, *i.e.* information has been lost.



It is perhaps worth noting that conversion of an analog quantity to digital representation need not be linear. Taking the example of light– or sound intensity, it might be more sensibleto digitize the logarithm rather than the quantity itself. First,

2. See, *e.g.*, P. Horowitz and W. Hill, *The Art of Electronics*, 2nd ed. (Cambridge University Press, New York, 1989).

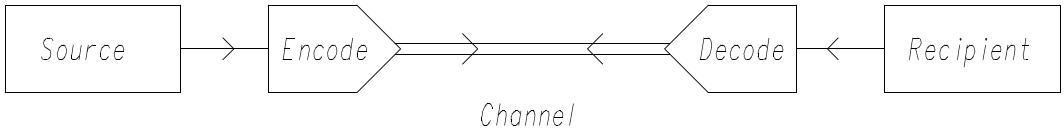
these quantities have a large dynamic range; and second, human senses empirically obey a logarithmic response rule.

Having discussed how analog signals (that is, most inputs from measuring instruments) can be digitized, it is time to discuss how the digital information itself must be handled.

2. Information content

To discuss the issues involved in data communication we visualize the entire system as consisting of input, encoding, transmission, decoding, and output, as shown schematically below:

We shall not concern ourselves here with either the source or the receiver of the data, but rather



with the encoder, the communications channel, and the decoder. The key to understanding data transmission is the *information content* of the data. According to Claude Shannon, the father of information theory, a message consisting of a string of symbols contains information

$$H = -\sum_k p_k \log_2\left(p_k\right), \qquad (1)$$

where p_k is the frequency (probability of occurrence) of the k'th symbol. Why does the logarithm to the base 2 appear here? Shannon notes that we can express the data in binary numbers. Hence we may define the information content as the absolute minimum number of bits that express the message. For example, suppose we flip a coin thrice—assuming the probabilities of heads and tails are equal, 0.5, there are eight possible, equally likely, outomes, or possible "messages". Each has probability ⅛ . Since $p_k = ⅛$ for each k, we have

$$H = -\sum_{k=1}^{8} ⅛ \log_2 (⅛) \equiv 8 \times ⅛ \times 3 = 3 .$$

In other words, 3 bits is the minimum required to send the message revealing which of the outcomes took place. This is not exactly a surprise, since we could assign a 1 to a head and a 0 to a tail and see directly that 3 bits can describe any 3-toss sequence. Shannon's formula, Eq. 1 (sometimes called the "entropy" of a message—guess why?) is more useful for analyzing more complex cases.

3. Data compression

The information content H estimates the minimum number of bits required to express a given message, hence it measures how much we can compress a message by some encoding technique. Of course compression is risky, as we shall see: it reduces message size, but necessarily also reduces the message's immunity to transmission errors.

Consider a message in single-case English, ignoring punctuation (lawyers will love this!). English has 26 letters plus a space to separate words, hence we need an alphabet of 27 characters. Based on the actual frequencies of letters and letter pairs (TH is common, QZ has probability 0) in actual English spelling, it is found that English can be compressed (for example using the Soundex algorithm) to about 1 bit per letter. However a message consisting of purely random strings of the 27 characters, where each character appeared with equal frequency, would have

$$H = \sum_{k=1}^{27} p_k \log_2 (1/p_k) = 27 \times \tfrac{1}{27} \times \log_2 27 = 4.76 .$$

Because it requires so few bits to transmit English using an efficient compression scheme, compared with random sequences from the same alphabet, we conclude there is actually much less information content in a typical English message than the maximum information carrying capacity of a 27-letter alphabet. Or in other words, the redundency of English is almost 80%. We can see this by examining a message without vowels:

MST PPL HV LTTL TRBL N RDNG THS MSSG

(MOST PEOPLE HAVE LITTLE TROUBLE IN READING THIS MESSAGE)

Given this fact, why is English—not to mention other human languages—so highly redundant? The answer must surely be that the redundency has evolved into our use of language to reduce errors in transmission, errors which at critical moments can be fatal.

Redundancy is our defense against *noise* (which we can define as random flipping of bits). Thus it will not do to compress a message maximally, since the loss of even one bit might render it undecipherable.

Consider how we might compress a message using a hypothetical 4-letter alphabet ACTG, with

$$p(A) = \tfrac{1}{2} \qquad p(C) = \tfrac{1}{4} \qquad p(T) = p(G) = \tfrac{1}{8} .$$

Now the obvious encoding, shown in the second column of the following table,

Character	Obvious Encoding	Alternate Encoding
A	00	0
C	01	10
T	10	110
G	11	111

uses 2 bits per character. But an alternate encoding, shown in the third column, uses, on the average,

$$\tfrac{1}{2} \times 1 + \tfrac{1}{4} \times 2 + \tfrac{1}{8} \times 3 + \tfrac{1}{8} \times 3 = 1.75$$

bits per character. Now, interestingly, the Shannon information content, Eq. 1, gives

$$H = \tfrac{1}{2} \times \log_2(2) + \tfrac{1}{4} \times \log_2(4) + \tfrac{1}{8} \times \log_2(8) + \tfrac{1}{8} \times \log_2(8) = 1.75 ,$$

illustrating the principle that H is the minimum number of bits per character.

This is the essence of Huffmann encoding, as applied to our hypothetical alphabet. An alternative method is to count 0's and 1's. That is, suppose we had, for some reason, the string

```
111111110000
```

We could write it as 8:4 (that is, 8 1's followed by 4 0's). In binary this would be 1000100, almost a 2-fold decrease in the number of bits. This method is called "run-length" encoding. It is most useful when we might expect long sequences of 1's and 0's. But this is exactly the nature of bit-mapped graphics files.

The compression algorithms currently most favored, when the data channel has high reliability (such as a disk drive), are based on the Lempel-Ziv algorithms. They use Huffmann encoding (or something closely related to it) for the short choppy sequences, and switch to run-length encoding for long sequences of 1's and 0's.

Here is an important fact: *any* compression method must make some files longer. If this were not so, then we could compress arbitrary sequences to zero length! How do we see this? Say $T(f)$ is the transformed file from input file f. Let $L(f)$ be the length of the file f. Clearly, if

$$L(\,T(f)\,) < L(f)$$

for any input file, then by applying the transformation again we would have

$$L(\,T(\,T(f)\,)\,) < L(\,T(f)\,)\,.$$

Manifestly, the lower bound of this sequence is 0.

4. Error correcting codes

We can define the *capacity* of a communications channel as the number of bits per unit time it can carry, call it C. The rate at which information is actually carried is $\frac{dH}{dt}$. A major theorem of information theory states that states that if

$$\frac{dH}{dt} < \mathrm{C}$$

there exists an encoding scheme (of length N) that makes the probability of a transmission error smaller than ε. That is, for a given $\varepsilon > 0$, $\exists$ a code of length $N(\varepsilon)$ for which $p_{\text{error}} < \varepsilon$.

On the other hand, if

$$\frac{dH}{dt} > \mathrm{C}$$

it is impossible to find such a code.

Here is an example of an error-correcting code, based on parity checking. Consider datat that can be expressed as a group of 4 bits $(b_0\,,\,b_1\,,\,b_2\,,\,b_3)$—that is, 16 possible messages. Let us add 3 more bits $(b_4\,,\,b_5\,,\,b_6)$ to the group:

$$b_4 = Q(b_0\, b_1\, b_2) \stackrel{df}{=} (b_0 \oplus b_1) \oplus b_2$$
$$b_5 = Q(b_0\, b_1\, b_3)$$
$$b_6 = Q(b_0\, b_2\, b_3)\,.$$

These 3 bits are called parity bits. The operation $\oplus$ means XOR or "exclusive-or", whose truth table is

Inputs		Output
A	B	A⊕B
1	1	0
1	0	1
0	1	1
0	0	0

Suppose the original message is 1001 but is received as 0001. The parity bits computed from the original message are 100, but the parity bits computed from the received message are 011. Since all three of the (computed) bits disagree with what was received (assuming the parity bits were not corrupted!), we realize that the b_0 was incorrectly transmitted.

Consider the case if 1001 were received as 1101 (that is, the second bit was mis-transmitted): then the computed parity bits after reception would be 010, and since the first two bits disagree, the error was in b_1 of the message. Again, if the message were received as 1011 the computed parity bits would be 001; the disagreement is in the first and third parity bits so the error was in b_2. Finally if 1000 were received, the computed parity would be 111, disagreeing in the second and third places, so the error in the mesage would be in b_3.

In other words (and this is worth checking), we find the following error table:

error	check
b_0	none agree
b_1	only b_6 agrees
b_2	only b_5 agrees
b_3	only b_4 agrees

Three check bits only yield a partial check/correction in cases where two of the message bits are wrong. There are 6 such cases, and we can see easily that there are not 6 unique patterns of agreeing pairs of parity bits, under the above rule. That is, we do not dare instruct the receiver to simply correct the presumed wrong bit because there might have been two wrong bits. For example, suppose the message 1101 were received as 1110. The check bits would disagree at b_6, suggesting we flip b_1. This would give the (wrong) message 1010, whose check bits 010 agree with the transmitted ones. In other words, our only recourse is to note the error and request retransmission of the data.

Suppose we are interested in data that it might be difficult or impossible to retransmit. In this case we need a stronger encoding scheme that, with high probability, permits the correction of the error(s).

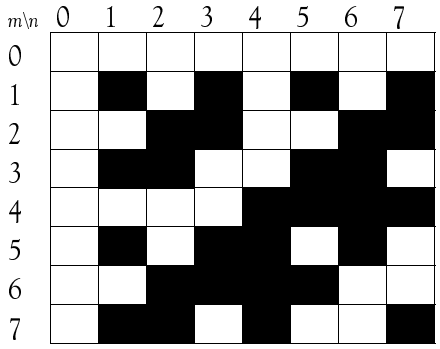
The $(2^n, n)$ Reed-Muller code[3,4] maps the 2^n integers in the range $[0, 2^n - 1]$ onto the rows of a square $2^n \times 2^n$ matrix constructed as follows. Let us define a 2×2 matrix

$$H_1 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

and define the direct product of matrices H_n and H_1,

$$H_{n+1} = H_n \otimes H_1$$

to be the matrix obtained by replacing all the elements of H_n with the matrix H_1. Then every place in the matrices H_n so obtained we replace 1 by 0 and –1 by 1. The 8×8 matrix so obtained from H_3 is shown below, where the cells containing 0's are left white and those containing 1's colored black. (This sequence defines the *Hadamard* matrices.)



Suppose we have data that lie in the range 0-7. These integers would normally require 3 bits to encode. However, if we encode them by mapping k into the kth row of the matrix, each integer will require 8 bits. However, by doing that we greatly increase its probability of being correctly received. The crucial point is that, considered as strings of bits, each row in the matrix differs from every other row in 2^{n-1} positions, or in the above case, 4 positions. This is called the "Hamming distance" between two rows. Suppose an error is made in transmission. If it is a 1-bit error, then the 8-bit string will differ from all the others in the encoding scheme. But it will be closest to 1 of them, and can be identified with that one. What if 2 bits were corrupted. Then the received string will be equidistant from two of the rows, and we cannot tell which one it was supposed to be. That is, if the number of corrupted bits exceeds $\lfloor (2^n - 1)/2 \rfloor$, where $\lfloor\ \rfloor$ means "the largest integer ≤ the contents", we cannot correct the error. The encoding of the Mariner space probe's video telemetry used $n=5$; each of the 32 rows of H_5 differs from the others in 16 places, so the Hamming distance is 16, and the maximum number of bits that can go awry without losing information is 7. If the noise event that corrupts one bit is

3. Richard W. Hamming, *Coding and Information Theory* (Prentice-Hall, Englewood Cliffs, NJ, 1980).
4. A good exposition is given in A.K. Dewdney, *The Turing Omnibus* (Computer Science Press, Rockville, MD, 1989), Ch. 11.

independent of that corrupting any other; and if the probability of each such event is $\varepsilon \ll 1$, then the probability of 7 going bad in a sequence of 32 bits is

$$p_7 = \frac{32!}{7!\,25!}\varepsilon^7(1-\varepsilon)^{25} \approx \frac{(32\varepsilon)^7}{7!}e^{-32\varepsilon}$$

To see what this means in practice, the following table gives the probability of 8 or more bits being corrupted in a 32-bit sequence, for several values of ε :

bit-flip probability ε	cumulative probability $\sum_{k=8}^{32} p_k(\varepsilon)$
0.1	0.0117
0.05	0.00014
0.01	8.5×10^{-10}

That is, using this error correction technique, only about 1 in 85 data will be corrupted beyond repair even at the noisiest level. The cost is that instead of the 5 bits necessary to send an integer in the range 0–31, we must send 32 bits, a ratio of a little more than 6 to 1.