FOUR

CODING AND INFORMATION THEORY

In this chapter we move from abstract considerations of computation to the more concrete realities of computer structure. I want to examine the limitations on machines resulting from the *unreliability* of their component parts. A typical machine may be built from millions of logic gates and other bits and pieces and if these components have a tendency to malfunction in some way, the operation of the machine could be seriously affected.

Components can let us down chiefly in two ways. Firstly, they may contain faults: these can arise during manufacture and are obviously of extreme importance. For example, when making a memory chip from silicon, flaws can be anywhere – where there was a bit of dirt in the material, or where the machine making it made a mistake – and the smallest fault can screw up an element of memory. If your memory system is such that all the cells have to work or the whole thing is useless, then just one tiny mistake can be very costly. A neat way to resolve this problem is to design systems which work around such flaws, spotting them and, perhaps, sealing them off from further usage. However, I will look at the physical structure of components later.

What I want to focus on now is a second way in which an element can let us down. This is when it fails *randomly*, once in a while, perhaps because of the random Brownian motion of atoms or just irregular noise in a circuit. Any such glitch can cause a component to fail, either temporarily or permanently. Now the odds against a particular element failing in such a way may be a million to one, but if we have billions of such elements in our machine, we will have thousands failing, all over the place, at any one time. When the earliest Von Neumann machines were in operation, they were constructed from relays and vacuum tubes whose failure rate was very high (of the order of one in a thousand), and the problem of unreliability was acute: with a million components one could expect a thousand of them to be acting up at any one time! Now it has turned out that as we have developed better and better systems with transistors, the failure rate has been going down for almost every machine we build. Indeed, until recently, the problem has ceased to be considered very serious. But as we manufacture computers with more and more parts, and get them to work faster and faster, and particularly as we miniaturize things more and more, this might not remain true. There are something like 10^{11} atoms in a modern transistor, but if we try to get this number down, to build switching devices with, say, a thousand atoms, the importance of noise and random failure examine the matter of unreliability in some detail. Besides, it is an interesting subject, and that should be reason enough to study it!

4.1: Computing and Communication Theory

We begin our discussion of unreliability by considering the aspect of computers for which it is most problematic, that of memory storage. For example, suppose that we have some data stored somewhere for a long time, and at some point the system makes a mistake and switches a bit somewhere — a one gets changed to a zero, for example. This sort of error can occur elsewhere in the machine, in its CPU for example, but this is less likely than it happening in memory, where the number of transistors and elements is so much larger. To examine this situation I am going to draw a useful analogy with another area of engineering; namely, with *communication theory*. In a communication system you send out a bunch of bits at one end, the transmitter, and at the other end, the receiver, you take them in. This is just sending a message. In the process errors can creep in: noise could affect the message in transit, reception could be bad, we might get glitches. Any of these could mean that the message we receive differs from the one sent: this is the so-called "communication problem in the presence of noise". Now this isn't exactly the same situation as with memory — which is like sending a message through time rather than space — but you can see the similarities. We store something in memory and at a later time we read it back out — in the interim the stored "message" is subject to noise. When it comes to the reliability of stored memory and sent messages there are important practical differences. It is possible in principle, for example, to continually check on the contents of our memory, whereas, if NASA sends a radio communication to a Jupiter probe there is no way of checking its contents while in transit. Nevertheless, the analogy is strong enough to make a look at communication theory worthwhile. We will start with a look at how we might go about detecting errors, an essential step before we can correct them.

4.2: Error Detecting and Correcting Codes

From now on I am going to use the language of communications, and will generally leave it to the reader to make the connection with memory systems in machines. Let us suppose we have a transmitted message, which we take to be some sequence of symbols[1], and we are going to be doing the receiving.

[1]From here on, we restrict these symbols to be binary digits, 1s and 0s. [RPF]

Obviously, we would like to know how trustworthy the received message is, and this brings us to our first matter, that of error detection. Is there some way in which we could know whether the message we have received is correct? Clearly, all we have to work with is the message: calling up the sender for confirmation defeats the object! Is there some way of building a check into the message itself, that will enable us to confirm it? The answer is yes, as we will shortly see.

4.2.1: Parity Checking

We first assume that the probability of an error arising in the message is very small; in fact, so small that we never have to worry about more than one error turning up. Furthermore, we will only consider errors in individual bits and not, for example, errors spread out across several neighboring, or related bits (such as "error bursts" caused by scratches on disks). Suppose the chance of an error in a symbol is one in ten thousand, and our message is ten symbols long. The chances of an error in the message are about one in a thousand. However, the odds against two errors are of the order of a million to one, and we shall consider this negligible. We will only bother trying to detect single errors, assuming doubles are too rare to worry about.

Here is a very simple scheme for checking for single errors, known as a *parity checking* scheme. Suppose we are sending the following ten-bit message:

$$1101011001$$

What we do is tag another bit onto the end of this string, which tells us the parity of the string — the number of 1's it contains, or, the same thing, the sum of its digits modulo 2. In other words, the extra digit is a 1 if the original message has an odd number of 1's: otherwise, the above message would have a 0 attached. This is an example of *coding* a message; that is, amending its basic structure in some way to allow for the possibility of error correction (or, of course, for reasons of security). When we receive the message, we look at the parity digit and if the number of 1's is wrong, then we have obviously received a faulty message. Note that this simple check actually enables us to detect any odd number of errors, but not any even number (although as we have ascribed vanishing probability to anything more than a single error, these are assumed not to occur)[2].

[2]Note that a simple machine that could do this checking would be the parity FSM discussed in the previous chapter. [RPF]

There are two main shortcomings of this procedure which we should address. Firstly, we might get an error in the parity bit! We would then be mistaken if we thought the message itself was wrong. Clearly, the longer the message, the less likely the error is to occur in the parity bit itself. Secondly, at best the check only tells us whether an error exists — it does not tell us *where* that error might be. All we can do on finding a mistake is have the message sent again. In our case, where we are using a computer, we might simply reboot the machine and go back to square one. Another minor shortcoming of this particular approach is that it leads to a certain inefficiency of communication — in this case, a ten percent inefficiency, as we had to send eleven bits to communicate a message of ten.

The obvious next question to ask is: can we construct a method for not only detecting the existence of an error, but actually locating it? Again the answer is yes, and the method is quite ingenious. It is a generalization of the simple parity check we have just examined. What we do is imagine that the data in the message can be arranged into a rectangular array, of (say) *m* rows and *n* columns (Table 4.1):

$$\begin{array}{c|c}
 & n \\ \hline
m & \begin{array}{c}
1\,1\,0\,1\ldots\ldots0\,1 \\
0\,0\,0\,1\ldots\ldots1\,1 \\
\cdots\quad\cdots\quad\cdots \\
\cdots\quad\cdots\quad\cdots \\
1\,1\,0\,0\ldots\ldots0\,1
\end{array}
\end{array}$$

Table 4.1 A Rectangular Data Array

Of course, the data would not be sent in this form: it would be sent as a binary sequence and then arranged according to some predefined rule, such as breaking the message into *m* blocks of *n* symbols and placing them one over the other. To check for errors, what we do is include at the end of each row a digit giving the parity of the row, and at the base of each column a parity digit for the column. These parity digits can be seeded into a sequential message without difficulty. An error in the array will then lead to a parity mismatch in both the row and the column in which the error appears, enabling us to pinpoint it precisely. In principle this scheme can detect any number of message errors, as long as they occur in different rows and columns.

We have to be careful, however, about errors occuring among the parity check bits. A particularly nasty instance would be a double error where a message digit and the parity digit for the row (say) both switch. We would know that there was an error, due to the column parity being wrong, but we might be inclined to think that it was a single column parity bit that was at fault – as we would have no confirming row parity error. However, we can safeguard against this ambiguity by placing another parity check in the array, this time at its lower right corner. This bit gives us the parity of the whole message (i.e. of the row and column totals), and using it we can detect – but not locate – such a double error. The end result of this double error detection may well be the same as with our single error detector – we go back to square one and send the message again – but it is still an improvement.

A useful way to quantify the efficiency of a coding method like this is by calculation of a quantity called the *redundancy R*:

$$R = \frac{\textit{no. of bits used in full}}{\textit{no. of bits in message}} \qquad (4.1)$$

The bigger R is, the less efficient our code. The quantity $R-1$ is usually known as the "excess redundancy". For our first, single-error-detecting code, the redundancy is $(n+1)/n$. For the rectangular array above we are using mn bits to send a message that is only $(m-1)(n-1)$ bits long. So we have the following result:

$$R = \frac{mn}{(m-1)(n-1)} \qquad (4.2)$$

This quantity is a minimum, and hence the code most efficient, when the array is a square, i.e. $m=n$. You might be tempted to say, "Well, I can get the redundancy down to near one by just taking m and n very large – let's just send our message in blocks and rows with not ten, but ten-thousand bits!" The problem with this is that there is a certain probability of each bit being in error, and if the number you are sending gets too big the chance of multiple errors begins to creep up.

4.2.2: Hamming Codes

I will now take a look at another single-error-correcting (and double-error-detecting) coding method based on parity-checking, which is both more efficient and a lot more subtle than the rectangular type. Actually, it is a kind of higher-dimensional generalization of the array method. In any message we send, some of the bits will be defined by the message itself, and the rest will be coding symbols — parity bits and the like. For any given message, we can ask the question: "How many check bits do I need to not only *spot* a single error, but also to *correct* it?" One clever answer to this question was discovered by Hamming, whose basic idea was as follows. The message is broken down into a number of subsets of digits, which are not independent, over each of which we run a parity check. The presence of an error will result in some of these checks failing. We use a well-defined rule to construct a binary number, called the "syndrome", which is dependent on the outcome of the parity checks in some way. If the syndrome is zero, meaning all parity checks pass, there is no error; if it is non-zero, there is an error, and furthermore the *value of the syndrome tells us the precise location of this error.* For example, if the syndrome reads 101, that is decimal 5, then the error is in bit five of the message. If on the other hand it reads 110010, then the error is in the fiftieth bit. The trick is to implement this idea.

We can straightaway make some statements about how many check bits we will need. Suppose our syndrome is m bits long so that we have m check bits. If we decide that a vanishing syndrome is to represent no error, that leaves at most (2^m-1) message error positions that can be coded. However, errors can occur in the syndrome as well as the original message we are sending. Hence, if n is the length of the original message, we must have:

$$2^m - 1 \geq (n + m) \tag{4.3}$$

or

$$n \leq 2^m - m - 1. \tag{4.4}$$

For example, if we wanted to send a message 11 bits in length, we would have to include a syndrome of at least four bits, making the full message fifteen bits long. This does not seem particularly efficient (efficiency = 11/15 or about 70%). However, if the original message was, say, 1000 bits long, we would only need ten bits in our syndrome (2^{10}= 1024) which is a considerable improvement!

Let us now see precisely how this syndrome idea works. As an example, let us continue with our problem of sending a message eleven bits long. As we saw, we will need four check bits. Each such bit will be a parity check run over a subset of the bits in the full fifteen-bit message. Just as with the simple parity check method, we will select a few specific bits in the message, calculate their overall parity, and adjust the corresponding check bit to make the total parity of the (subset plus check) zero. If there is an error in this subset, the parity check will fail. The clever thing about the Hamming code is that each message bit is in more than one subset and hence contributes to more than one parity check, but not to all of them. By seeing which parity checks fail and which pass, we can home in on the error uniquely. We assign to each parity check a one if it fails and a zero if it passes, and arrange the resulting bits into a binary number, the syndrome. This indicates directly the error position. It is pretty much arbitrary, but we will construct the syndrome by reading the parity checks from left to right in the message.

For the moment, we will assume that the parity check bits are placed in some order throughout the message, although we will not mind where for the moment. We will first identify the subsets each covers. To do this, it will help to list the four-digit binary representations of the positions within the message:

1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Let us look at the rightmost parity check, the far right digit of the syndrome. Suppose this is non-zero. Then there will be a parity failure in a position whose binary representation ends in a one: that is, one of positions 1, 3, 5, 7, 9, 11, 13 or 15. This is our first subset. To get the second, look at the second digit from the right. This can only be non-zero for numbers 2, 3, 6, 7, 10, 11, 14 and 15.

Note what is happening. Suppose we have found that both of these parity checks failed, i.e., we assign a 1 to each. The error must be in a position that is common to both sets, i.e. a binary number of the form *ab*11. This can only be 3, 7, 11 or 15; we have narrowed the possible location choices. To find out which, we have to do the remaining parity checks. The third check runs over digits 4-7, and 12-15. The final check covers digits 8 through 15. Suppose both of these are zero, that is, the parity checks out; we hence put two zeroes in our syndrome. Then there is no error in positions 4-7, 12-15, 8-15 (which obviously overlap). But there is an error somewhere in 3, 7, 11, 15. The only one of these latter four that is not excluded from the previous sets is position 3. That must be where the error lies. Of course, in binary 3 is 0011 – the syndrome calculated from the parity checks.

Let us pick a real example to illustrate these ideas in a more concrete manner. Suppose we want to send the eleven-bit message 10111011011. We first have to decide where to stick in our parity bits. There is nothing in what we have said so far that tells us whereabouts in the message these must go, and in fact we can put them anywhere. However, certain positions make the encoding easier than others, and we will use one of these. Specifically, we place our check bits at positions 1, 2, 4 and 8. We now have:

Codeword	*a*	b	1	*c*	0	1	1	*d*	1	0	1	1	0	1	1
Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Importantly, the check bits here, read from left to right, will give the *reverse* of the syndrome: that is, the first leftmost digit of the syndrome, when written out, would be read from the parity check of *d*, not *a*, and the last rightmost read from *a*, not *d*. Again, this is a matter of calculational simplicity.

We can now work out *a*, *b*, *c* and *d*. Bit *a* is the parity of the odd positions: 1, 3, 5, 7, 9... This is 1. Hence, *a* is 1. Bit *b* is found by summing the parities of positions 2, 3, 6, 7, 10, 11, 14 and 15. This gives zero. Bit *c* comes from the parity of positions 4 through 7 and 12 through 15, giving *c*=1. Finally, we get *d* by doing a check on 8 through 15, giving *d*=1. Note, incidentally, how this placing of the check bits leads to straightforward encoding, that is, calculation of *abcd*. If we had chosen an apparently more straightforward option, such as placing them all at the left end of the string occupying positions 1 to 4, we would have had to deal with a set of simultaneous equations in *a*,*b*,*c*,*d*. The important feature of our choice is that the parity positions 1, 2, 4, 8 each appear in only one subset, giving four independent equations. The completed message is thus:

$$101101111011011.$$

Let us see what happens when an error occurs in transmission. Suppose we receive the message:

$$101101011011011.$$

Where is the mistake? The sum over the odd-placed digits is 1 — a failure. We assign a 1 to this in our syndrome, this being the rightmost digit. The second parity check also gives 1, another failure, and our syndrome is now $xy11$, with x and y to be determined. Bit y comes from our next check, and is 1 again — another failure! The syndrome is now $x111$. To find x we check the parity of places 8 through fifteen, and find that this is zero — a pass. We therefore assign a zero to x, giving our syndrome as 0111. This represents position 7, and indeed if you compare the original and corrupted messages, position 7 is the only place you will find any difference.

An interesting feature of the Hamming code is that the message and code bits are on the same footing — an error in a code bit is located the same way as in a message bit. We can extend this code to detect double errors quite simply. We tag on to the end of the message yet another check bit, this time representing the parity of the whole thing. For the (uncorrupted) message we gave above, the parity is 1, so we attach a 1 at the rightmost end, to give us zero overall parity as usual. Now, if there is a single error in the message, the parity of the 15-bit message will change, and this will show up as a mismatch with the sixteenth bit. However, if there is a double error, the parity of the 15-bit message will not change, and all will look normal in the sixteenth bit; yet the parity checks within the former will fail, and this indicates a double error. Observe that if the 15-bits check out, but the overall parity does not, this indicates an error in the overall check bit. Note that the cost of these benefits is almost a 50% inefficiency — five check bits for an eleven bit message. As we pointed out earlier however, the inefficiency drops considerably as we increase the message length. For a one-thousand bit message, the inefficiency is a tiny one percent or so.

It is worth examining the practical usefulness of error-detecting codes like this by looking at how the consequences of message failure become less drastic for quite small losses of efficiency. Let us suppose that we are sending a message in separately coded batches of about a thousand bits apiece, and the probability of an error in a single one of the bits is 10^{-6}, or 10^{-3} per batch. We take these errors to be random and independent of each other. We can use Poisson's Law to get a handle on the probabilities of multiple errors occurring when we send our batches. If the mean number of errors expected is m, then the actual probability of k errors occurring is given by:

$$\frac{1}{k!}\, m^k \exp(-m). \tag{4.5}$$

The expected number of errors per batch is, as we have said, 10^{-3}. Hence, the probability of a double error in a batch is $(1/2).10^{-6}$ to high accuracy (we can ignore the exponential), and the probability of a triple error is $(1/6).10^{-9}$. Now suppose we have no error detection or correction, so we are expending no cash on insurance. If an error occurs, we get a dud message; the system fails. On average, we should only be able to send a thousand batches before this happens, which is pretty miserable. Suppose now that we have single error detection, but not correction, say a simple parity check. Now, when the system detects an error, it at least stops and tells me, and I can try to do something about it. This still happens once in every thousand or so batches, but it is an improvement that we gain at the cost of just one-tenth of a percent message inefficiency — one parity bit per thousand message bits. This system will fail whenever there are two errors, which occurs roughly once every two million batches. However, suppose we have our one percent gadget, with single error correction and double error detection. This system will take care of it itself for single errors, and only stop and let me know there is a problem if a double error occurs, once in every million or so batches. It will only fail with a triple error, which turns up in something like every six billion batches. Not a bad rate for a one percent investment!

Issues of efficiency and reliability are understandably central in computer engineering. An obvious question to ask is: "How long should our messages be?" The longer the better as regards efficiency of coding, but the more likely an error is to occur, and the longer we will have to wait to find out if an error has occurred. On the other hand, we might be prepared to sacrifice efficiency for security, sending heavily coded but brief messages so that we can examine them and regularly feel that we can trust what we receive. An example of the latter kind that is worth noting is in the field of communications with deep solar system spacecraft such as the *Voyager* series. When your spacecraft costs billions of dollars, and you have to send radio messages across millions of miles and be as certain as possible that it arrives uncorrupted — we don't want the cameras pointing at the Sun when they should be looking at Jupiter or Saturn — efficiency goes out the window. Spacecraft communications rely on a kind of voting technique, referred to as "majority logic decisions". Here, each bit in the source message is sent an odd number of times, the idea being that most of these will arrive unchanged — i.e. correct — at the receiver. The receiver takes as the message bit whichever bit appears most in each bit-batch, in best democratic fashion. How many copies you send depends on the expected error rate. Anyway, this is just a little example from communication, so I won't dwell on it. It doesn't seem to have too much relevance for computing (except perhaps for those in the habit of backing up their files a dozen times).

4.2.3: An Aside On Memory

Let me briefly discuss one interesting way in which the Hamming coding technique can be applied to computer memory systems. With the advent of parallel processing, it has become necessary to load and download information at an increasingly rapid rate, as multiple machines gobble it up and spit it out faster and faster. This information is still stored on disks, but individual disks are simply not fast enough to handle the required influx and outflow of data. Consequently, it is common practice to use "gang-disk" systems, where lots of disks share the load, simultaneously taking in and spewing out data. Such systems are obviously sensitive to errors on individual disks: if just one disk screws up, the efforts of the whole bunch can be wasted. Every manufacturer would like to build the perfect disk, one that is error-free — sadly, this is impossible. In fact, it is fair to say that the probability of any given floppy, and certainly any hard disk, on the market being free of flaws (e.g. bits of dirt, scratches) is virtually nil[3]. The reason you do not usually notice this is that machines are designed to spot flaws and work around them: if a computer locates a bad sector on a disk it will typically seal it off and go on to the next good one. This all happens so fast that we don't notice it. However, when a disk is working alongside many others in a parallel processing environment, the momentary hang up as one disk attends to a flaw can screw everything up.

The Hamming method can come to our rescue. Let's suppose we have thirty-two disks working together. We take twenty-six of these to be loaded with information, and six to be fake. We only have twenty-six worth of data, but thirty-two lines coming into the system. In each click of the clock we get one

[3]Readers might like to contact disk manufacturers and try to get some figures on the flaw rate on their products. You will have a hard time getting anywhere! [RPF]

bit fed in from each disk. What we do is run a parity check for each input of thirty-two bits coming into the system, one per disk, according to Hamming's method — hence six parity bits for twenty-six of message — and correct the single errors as they come along. Note that in this sort of set-up the odds against double errors occurring are enormous; that would need two or more disks to have errors in the same disk locations. It is a possibility, of course, but even if it happens we can soup up our system to detect these double errors and have it grind to a halt temporarily so that we can fix things. The flaws we are talking about here, in any case, are not really random: they are permanent, fixed on the disk, and hence will turn up in the same place whenever the disk is operating. We can avoid double errors of this kind by running the system, debugging it, and throwing away any disks that have coincidentally the same error spots. We then buy new ones. We do this until no more double errors are found. The use of this Hamming coding method saved the whole idea of gang disks from going down the drain.

Here are some problems for you to look at.

Problem 4.1: Devise a Hamming-type code for a message alphabet with a number (a) of elements (for binary, a=2). Show that, if the number of code symbols is r, and the total message length is N (so the original message is $N-r$) we must have

$$1 + (a-1)N = a^r$$

Work out a simple example.

Problem 4.2: This is an interesting mathematical, but not overly practical, exercise. We define a *perfect* code to be one for which:
(a) each received coded message can be decoded into some purported message, and
(b) the purported message is correct if there are less than a specific number ol errors.

Codes can correct up to e errors (thus far we have only considered e=1). Car you construct a perfect code for binary symbols (a=2) with e=3, i.e. triple erroı correction? Hint: stick to N=23, with 12 bits of message and 11 of code (oı syndrome). There is also a solution to this problem for a tertiary alphabet (a=3) For this case, try N=11, with 6 data digits and 5 of code.

4.3: Shannon's Theorem

We have asked lots of fundamental questions so far in this book. Now it is time to ask another. In principle, how far can we go with error correction? Could we make a code that corrects two, three, four, five, six, seven... errors and so on, up to the point where the error rate is so low that there is no point in going any further? Let's set the acceptable chance of us getting a failure at 10^{-30}. If you don't like that, you can try 10^{-100} : any number will do, but it must be non-zero, or you'll get into trouble with what follows. I reckon 10^{-30} will do.

Suppose we're sending a message of length M_C, which is the length of the full coded message, containing original data and coding bits. As usual, we're working in binary. The length of the data message we call M. Let's assume that the probability of any single bit going wrong is q. We want to design a coding scheme that corrects single, double, triple errors and so on, until the chances of getting more errors in M_C is less than our chosen number, 10^{-30}. How many code bits are we going to need? How much of M_C is going to be message? What's left?

Claude Shannon has shown that the following inequality holds for M and M_C:

$$M/M_C \leq f(q) = 1 - q(\log_2[1/q] + [1-q]\log_2[1/(1-q)]) \qquad (4.6)$$

Given this, says Shannon, if no limit is placed on the length of the batches M_C, *the residual error rate can be made arbitrarily close to zero.* In other words, yes, we can construct a code to correct n-tuple errors to any accuracy we choose. The only restriction in principle is the inequality (4.6). In practice, however, it might require a large batch size; and a lot of ingenuity. However, in his extraordinarily powerful theorem, Shannon has told us we can do it. Unfortunately, he hasn't told us *how* to do it. That's a different matter!

We can construct a table with a few values for q to illustrate the upper limit Shannon has placed on coding efficiencies (Table 4.2):

q	M/M_C	M_C/M
1/2	0	∞
1/3	0.082	12.2
1/4	0.19	5.3
0.1	0.53	1.9
0.01	0.919	1.09
0.001	0.988	1.012

Table 4.2 Shannon's Coding Efficiency Limits

Note that if q is 0.5 — that is, there is a fifty-fifty chance that any bit we receive might be in error — then we can get no message through. This obviously makes sense. As the error rate drops, the upper limit on the efficiency increases, meaning that we need fewer codebits per data bit. For any given q, however, it is very difficult to reach Shannon's limit.

The actual proof of this theorem is not easy. I would like to first give a hand-waving justification of it which will give you some insight into where it comes from. Later I will follow a geometrical approach, and after that prove it in another way which is completely different and fun, and involves physics, and the definition of a quantity called information. But first, our hand-waving. Let us start with the assumption that M_C is very, very large. This will enable us to make some approximations. If the probability of a single bit error is q, then the average number of errors we would expect in a batch is:

$$k = qM_C. \tag{4.7}$$

This isn't exactly true - the actual number may be more or less — but this is the average error rate we expect, and it will do as a rough guess. We have to figure out how much coding we need to dispose of this number of errors. The number of ways this number of errors could be distributed through a batch is given by simple combinatorics:

$$\frac{M_C!}{k!\,(M_C - k)!} \tag{4.8}$$

Let us assume that we have m code bits. Such a number of bits can describe 2^m things. This number of bits must be able to describe at least the M bits of the data message plus the exact locations (to give us error *correction*, not just detection) of each possible distribution of errors, of which we are saying there are $M_C!/k!(M_C-k)!$ It is clear that $m \leq M_C - M$ (since some bits could be redundant), so we have the inequality:

$$2^{M_C-M} \geq \frac{M_C!}{k!\ (M_C - k)!} \tag{4.9}$$

We now take the logarithm of both sides. The right hand side we work out approximately, using Stirling's formula:

$$n! = \sqrt{(2\pi n)}\ n^n\ e^{-n} \exp[(1/12n) - (1/360n^3) + ...] \tag{4.10}$$

for large n. Hence:

$$\log n! \approx (1/2) \log n + n \log n - n + O(1/n) \tag{4.11}$$

(Here, $\log x = \log_e x$.) The last term simply represents a lot of junk that gets smaller as n gets bigger (tending to zero in the limit), plus terms like $\log 2\pi$. We can in fact get rid of the first term, namely $(1/2)\log n$, as this is small compared to the next two when n is large. We thus use:

$$\log n! \approx n \log n - n \tag{4.12}$$

and with this the right hand side of the inequality becomes:

$$M_C \log M_C - M_C \log(M_C - k) + k\log(M_C - k) - k\log k \tag{4.13}$$

which, using $k = qM_C$ and a little algebra, is:

$$M_C\ [q\log_2(1/q) + (1-q)\log_2(1/1-q)]. \tag{4.14}$$

We have here converted the natural logarithm to base two, which simply introduces a multiplicative factor on both sides, which cancels. Taking the logarithm to base two of the left hand side of the inequality, and dividing both sides by M_C, we end up with Shannon's inequality.

This inequality tells us that, if we want to code a message M, where the bit error rate is q, so that we can correct k errors, the efficiency of the result cannot exceed the bounds in (4.6). Of course, k is not arbitrary; we have taken it to be the mean number of errors, $k=qM_C$. The question we would like to have answered is whether we can code a message to be sure that the odds against more than a certain number of errors, say k', occurring is some number of our choosing; such as 10^{-30}. Shannon's actual Theorem says that we can do this; let us take our "proof" a little further to see why this might be so.

The number of errors that can occur in the message is not always going to be k, but will be k within some range and probability. In fact, the distribution of errors will follow a binomial distribution, with mean qM_C ($=k$) and standard deviation $\sigma = \surd[M_C.q(1-q)]$. It is a standard result that, for MC large and q small, we can approximate this with a Gaussian (or Normal) distribution with mean qMC ($=k$) and standard deviation $\sigma = \surd[M_C.q(1-q)]$; that is, the same as before. Now to ask that our error rate be less than a number N (e.g. 10^{-30}) is equivalent to demanding that the number of errors we have to correct be less than k', where:

$$k' = k + g\sigma \tag{4.15}$$

for some finite number g dependent on N. For the Gaussian distribution, the probability that the number of errors lies within one standard deviation from the mean k is 68%; within two it is 96%; within three 99%. So, for example, if we wanted to be 95% certain that there would be no errors in our message, we would have to demand not that we be able to correct k errors but:

$$k + 2\sigma \tag{4.16}$$

errors. In this case $g=2$. For any level of probability we pick, we can find a value for g. As a rule, the probability of finding errors g standard deviations from k goes like:

$$\exp(-g^2/2) \tag{4.17}$$

and we can see how incredibly rare we can make errors for relatively small g. If g is twenty, for instance, this factor is exp(-200), or about 10^{-100}. A heck of a lot smaller than our 10^{-30}! For our choice of number, we get g to be about six.

To make use of this, we simply amend (4.9) by replacing k by k', the new number of errors we want to correct. If we can still find a code to do this, then we know that the odds of errors occurring in its transmission are less than our 10^{-30} or whatever. I leave it as an exercise for the reader to put:

$$k' = k + g\sigma = k + g\sqrt{k(1-q)} \tag{4.18}$$

into the inequality, and show that, in the limit, Shannon's result emerges as before.

What Shannon has given us is an upper limit on the efficiency. He hasn't told us whether or not we can find a coding method that reaches that limit. Can we? The answer is yes. I'll explain in more detail later, but the technique basically involves picking a random coding scheme and then letting M_C get larger and larger. It's a terrific mathematical problem. Provided M_C is big enough, we can reach the upper efficiency whatever the coding scheme. However, the message length might have to be enormous. A nice illustration of how big we have to take M_C in one case is from satellite communication. In sending messages from Earth to Jupiter or Saturn, it is not unusual for an error rate q of the order of a third to come through. The upper limit on the efficiency for this, from our table, is 8%; that is, we would have to send about 12 code bits for each data bit. However, to do this would require a prohibitively long M_C, so long that it is not practical. In fact, a scheme is used in which about *one hundred and fifty* code bits are sent for each data bit!

4.4: The Geometry of Message Space

I am now going to look at Shannon's Theorem from another angle, this time using geometry. In doing so I will introduce the useful idea of "message space".

Although this is primarily of importance in communications, and we are doing computing, I have found the idea interesting and useful, and you might too.

Message space, simply, is a space made up of the messages that we want to transmit. We are used to thinking of a space as something which can be many-dimensional, either continuous or discrete, and whose points can be labeled by coordinates. Message space is a multidimensional discrete space, some or all of whose points correspond to messages. To make matters a little more concrete, consider a three-bit binary code, with acceptable words:

000, 001, 010, 011, 100, 101, 110, 111.

These are just the binary numbers zero to seven. We can consider these numbers to be the coordinates of the vertices of a cube in three-dimensional space, as shown in Figure 4.1 below:

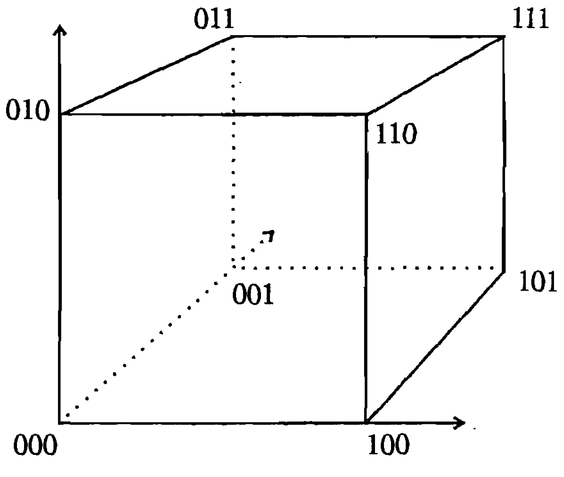


Fig. 4.1 A Simple Message Space

This cube is the message space corresponding to the three-bit messages. The only points in this space are the vertices of the cube — the space between them in the diagram, the edges, and whatnot are not part of it. This space is quite tightly packed, in that every point in it is an acceptable message; if we change one bit of a message, we end up with another. There is no wastage, everything is significant. We could easily generalize to a four-bit message, which would have a message space that was a 16-vertex "hypercube", which unfortunately our brains can't visualize! An *m*-bit message would require a 2^m-dimensional space.

What happens if there is an error in transmission? This will change the bits in the sent message, and correspond to moving us to some other point in the message space. Intuitively, it makes sense to think that the more errors there are, the "further" we move in message space; in the above diagram, (111) is "further" from (000) than is (001) or (100). This leads us to introduce a so-called "distance function" on the message space. The one we shall use is called the *Hamming distance*. The Hamming distance between two points is defined to be the number of bits in which they differ. So, the Hamming distance from 111 to 000 is 3, while from 001 to 000 it is just 1. According to this definition, in a 4-d space 1110 is as far from 1101 as is 0100, and so on. This makes sense. The notion of distance is useful for discussing errors. Clearly, a single error moves us from one point in message space to another a Hamming distance of one away; a double error puts us a Hamming distance of two away, and so on. For a given number of errors e we can draw about each point in our hypercubic message space a "sphere of error", of radius e, which is such that it encloses all of the other points in message space which could be reached from that point as a result of up to e errors occurring. This gives us a nice geometrical way of thinking about the coding process.

Whenever we code a message M, we rewrite it into a longer message M_C. We can build a message space for M_C just as we can for M; of course, the space for M_C will be bigger, having more dimensions and points. Clearly, not every point within this space can be associated one-on-one with points in the M-space; there is some redundancy. This redundancy is actually central to coding. e-Error correction involves designing a set of acceptable coded messages in M_C such that if, during the transmission process, any of them develops at most e errors, we can locate the original message with certainty. In our geometrical picture, acceptable messages correspond to certain points within the message space of M_C; errors make us move to other points, and to have error correction we must ensure that if we find ourselves at a point which does not correspond to an acceptable message, we must be able to backtrack, uniquely, to one that does. A straightforward way to ensure this is to make sure that, in M_C, all acceptable coded message points lie at least a Hamming distance of:

$$d = 2e + 1 \tag{4.19}$$

from each other. We can see why this works. Suppose we send an acceptable message M, and allow e errors to occur in transmission. The received message M' will lie at a point in M_C e units away from the original. How do we get back from M' to M? Easy. Because of the separation of $d = 2e + 1$ we have demanded, M is the closest acceptable message to M'! All other acceptable messages must lie at a Hamming distance of at least $e+1$ from M'. Note that we can have simple error *detection* more cheaply; in this case, we can allow acceptable points in M_C to be within e of one another. The demand that points be $(2e+1)$ apart enables us to either correct e errors or detect $2e$.

Pictorially, we can envisage what we have done as mapping the message space of M into a space for M_C in such a way that each element of M is associated with a point in M_C such that no other acceptable points lie within a Hamming distance of $2e+1$ units. We can envisage the space for M_C as built out of packed spheres, each of radius e units, centered on acceptable coded message points (Fig. 4.2). If we find our received message to lie anywhere within one of these spheres, we know exactly which point corresponds to the original message.

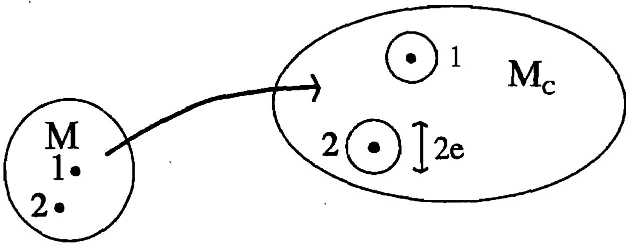
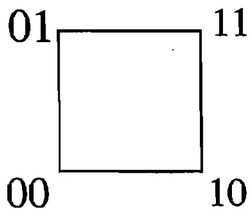


Fig. 4.2 A Message Space Mapping

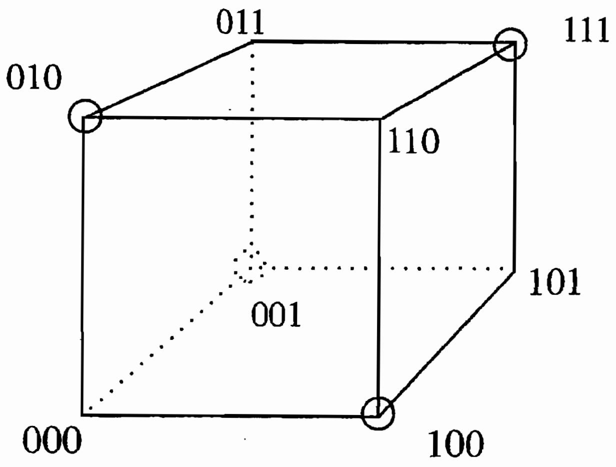
To make this idea a little more concrete, let us return to the three-bit cube we drew earlier. We can consider this the message space M_C corresponding to a parity-code that detects, but does not correct, single errors for the two-bit message system M comprising:

00, 01, 10, 11.

This system has the simple two-dimensional square message space:



The parity code simply tags an extra digit onto each message in M, clearly resulting in a 3-d cubic space for M_C. The acceptable messages are 000, 011, 101, and 110. This leaves four vertices that are redundant:



Any error in transmission will put us on one of these vertices, telling us that an error has occurred, but not where. Note that each false vertex lies within a Hamming distance of 1 from a genuine one. If we wanted single error correction for this system, we would have to use a space for M_C of four dimensions.

So if our coding system works, we should be able to move each of our message points into somewhere in the message space of M_C such that they are sufficiently separated. Every now and again we will be forced to allow some overlap between spheres of error, but this is not usually a problem[4]. We can now quickly see how this geometrical approach offers another proof of Shannon's Theorem. Use M and M_C to denote the dimensions of the original and coded message spaces respectively – this is just a fancy way of describing the lengths of the message strings. The number of points in M is 2^M, in M_C, 2^{M_C}. To correct k errors, we need to be able to pack M_C with spheres of error of radius k, one for each point in M. We do not want these to overlap. Using this, we can obtain an inequality relating the volume of M_C to that of the spheres. Now in a discrete space of the kind that is message space, the volume of a sphere is defined to be the number of points contained within it. It is possible to show

[4]"Perfect" codes, which we introduced in a problem earlier, are actually those for which the error spheres "fill" the message space without overlapping. If the spheres have radius e, then every point in the space lies within e units of one, and only one, message point. [RPF]

that, for an M_C-dimensional space, the number of points spaced a unit length apart that lie within a radius k units of a point is:

$$\frac{M_C!}{k!\ (M_C - k)!} \tag{4.20}$$

By noting that the volume of M_C must be greater than or equal to the number of points in each error sphere multiplied by the number of spheres, that is, the number of points in M, we recover the inequality (4.6). There is no need to go through the subsequent derivations again; you should be able to see how the proof works out.

Problem 4.3: Here is a nice problem you can try to solve using message space (that's how I did it). By now you are familiar with using a single parity bit at the end of a message to detect single errors. One feature of this technique is that you always need just one check bit irrespective of how long the message is; it is independent of M_C. The question is, can we also set up a method for detecting double errors that is M_C-independent? We are not interested in correcting, just detecting. We want a finite number of bits, always the same, and it would seem pretty likely that we could do it with only two bits! Recall that for the Hamming code we could correct a single error and detect doubles with a check bit for overall parity and the syndrome, but the number of check bits contributing to the syndrome depended on the message length. You should find that it is actually impossible to detect doubles without bringing in the length of the message.

4.5: Data Compression and Information

In a moment, I am going to look at Shannon's Theorem in yet another way, but first I would like you to let me wander a bit. The first direction I want to wander in is that of data compression, and I'd like to explain some of the ideas behind this. Consider a language like English. Now this has 26 letters, and if we add on commas, full stops, spaces and what-not, we have about thirty symbols for communication. So, how many things can I say in English if I have ten symbols at my disposal? You might say, well, thirty times ten. Not true. If I wrote the following string for you:

cpfajrarfw

it wouldn't be English. In real, interpretable English, you can't have everything; the number of acceptable words is limited, and the ordering of the letters within them is not random. If you have a "T", the odds are the next letter is an "H". It's not going to be an "X", and rarely a "J". Why? The letters are not being used uniformly, and there are very much fewer messages in English than seem available at first sight.

Perhaps, almost certainly, each one of you has parents back home who suffer from the fact that you never get around to writing letters. So they send you a card, all addressed and everything, which has on the back a lot of squares. And one of the squares says "I'm having a good time and enjoying CalTech. Yes/No." Next square says "I met a beautiful girl or handsome boy" or something, "Yes/No." The next message says "I have no more laundry to send", or "I'm sending it at such-and-such a time", and so on. What the poor parents are doing is converting long English sentences into single bits! They are actually trying to shame you into writing, but failing that, they are producing a method for you to communicate with them which is more efficient bitwise than your going to all that trouble. (Of course, you still have to post the card!) Another example of improving efficiency, and this is something you've all probably done, is to clip a long distance telephone company by prearranging to ring a friend and let the phone ring a set number of times if you're going to his party, another number if you're not, or maybe signal to him with three rings that you'll be at his place in your car in five minutes and he should go outside to wait, or what-have-you. You're calling long distance but getting the message through costs you nothing because he doesn't pick up the phone.

A related question is: how inefficient is English? Should I send my ten symbols directly, from the English alphabet? Or should I perhaps try a different basis? Suppose I had 32 rather than 30 symbols, then I could represent each element as a 5-bit string. Ten symbols hence becomes fifty bits, giving a possible 2^{50} messages. Of course, as I've said, most messages won't make sense. Whenever a Q appears, we expect it to be followed by a U. We therefore don't need to send the U; except for awkward words like Iraq, of course, which we could deal with by just sending a double Q. So we could exploit the structure of the language to send fewer symbols, and this packing, or compression, of messages is what we're going to look at now.

A good way to look at the notion of packing is to ask, if we receive a symbol in a message, how surprised should we be by the next symbol? I mean, if we receive a T, then we would not be surprised if we got an I next, but we would if we got an X, or a J. If you pick a T, the chances of a J next are very small; in English, you don't have the freedom to just pick any letter. What we want to guess is how much freedom we have. Here is an interesting way to do it. Try this experiment with your friends. Take an English text, and read it up to a certain point. When you stop, your friend has to guess the next letter or symbol. Then you look at the next one and tell him whether he's right. The number of guesses he has to make to get the next letter is a handy estimate of the freedom, of the possibilities for the next letter. It won't necessarily be accurate — people's guesses will not be as good as a mechanical linguistic analysis — but it'll give you an idea. But in any case, people will be able to guess the next letter at a much better rate than one in thirty! In fact, the average number of possible letters following a letter in English is not 26 but about 5. You can work this out from your experiment by doing it often and averaging the number of guesses. And that gives you an idea of how English can be compacted. It also introduces us to the notion of how much *information* is in a message. We will return to this.

Another way of considering this problem of compression is to ask: if you had N symbols of English, with 32 possibilities for each, how many messages could you send? If you like, what is the greatest amount of information you could convey? As we have discussed, you could not send the full 32^N, as most would not make sense. Suppose the number of potentially sendable messages[5] is n. We can label each of these, code them if you like, by a number. We'll take this to be in binary. Now both guys at each end of the message can have a long list, telling them which number corresponds to which message, and instead of communicating by sending a full message, they just send numbers. This is exactly analogous to you with the cards your parents send — "Yes, I need a haircut"; "No, I didn't do my homework"; and so on. You cannot get a more efficient sending method than this, compressing a whole message down to one number. We can work out how many bits we will need to send to cover all the messages. If we need I bits, then we have:

$$2^I = n, \quad or \quad I = \log_2 n. \tag{4.21}$$

This number, the number of bits we minimally need to send to convey as much as we possibly could have conveyed in the N bits of English (or whatever other

[5]Strictly speaking, we mean the number of equally likely messages that can be sent. In reality, some messages will be more likely than others. However, we are not being rigorous in what follows, and will not worry about this for the time being. [RPF]

system being used) is called the *information* content, or just the *information* being sent. It is important to stress that the meaning of the word "information" here differs from that in ordinary usage — it is not totally distinct from this, but generally "information" in our sense tells us nothing about the usefulness of or interest in a message, and is strictly an academic term. There are lots of words like this in science: the meaning of the words "work" in physics and "function" in mathematics bears little relationship to their colloquial meanings. We will return to the concept of information and define it more rigorously later. For the moment, just bear in mind the fundamental idea: that we are coding messages into a binary system and looking at the bare minimum of bits we need to send to get our messages across.

It is possible to give a crude definition of the "average information per symbol" using the notions we have developed. Suppose we have a number of symbols that is not N, but twice N. What number of possible messages will correspond to this? For figurative purposes we can split the $2N$-symbol string into two N-symbol strings:

$$2N$$

.... / ....

$$N \qquad\qquad N$$

As a rough guess, we may expect that the number of potentially sendable messages will be the number of messages in each string multiplied together, or n^2. In general, of course, the precise answer will be horribly difficult to find. For example, there will be what we might call "edge effects" — the possibility of words being formable at the join of the two strings, crossing from one into the other — since the two N-symbol strings are not actually separated. There can also be "long-range correlations" where parts of the string influence others some distance away. This is true in English, where the presence of a word at one point in a message typically affects what other words can appear near it. As we are not being rigorous yet we will not worry about such problems. In fact, you can see that if we let N get bigger and bigger, our rough guess gets more accurate. If we have $2N$ symbols, we get about n^2 messages; if we have $3N$, about n^3; and generally if we have xN symbols, the number of messages will be about n^x. If we write the information content from N symbols as $I(N)$, we have:

$$I(xN) \approx \log_2(n^x) = x\log_2 n, \tag{4.22}$$

and we see that the ratio:

$$r = \frac{I(xN)}{xN} \approx \frac{\log_2 n}{N} = \frac{I(N)}{N} \tag{4.23}$$

is independent of x. So for large enough N, as long as our approximation gets better, it tends to a constant. We call this ratio the *information per symbol* of our system, an interpretation that seems clear from the right hand side of (4.23).

Let us return to the notion of information and try to get a better idea of what it means. In a sense, the amount of information in a message reflects how much surprise we feel at receiving it. Consider, say, receiving a printed communication from a bookshop, such as: "We are pleased to tell you that the book you ordered is in stock"; or its opposite: "We are sorry to inform you that ... is not in stock." These long messages contain many more symbols but no more information than the simple "Yes" or "No" you could elicit from a shopworker if you called the bookshop direct. Most of the symbols in the printed communications are redundant in any case: you only have to spot the words "pleased" and "sorry" to figure out what they are saying. In this respect, information is as much a property of your own knowledge as anything in the message.

To clarify this point, consider someone sending you two duplicate messages: a message, then a copy. Every time you receive a communication from him, you get it twice. (This is not for purposes of error detection; it's just a bad habit!) We might say, well, the information in the two messages must be the sum of that in each (remember, $I(n_1n_2) = \log_2(n_1n_2) = \log_2 n_1 + \log_2 n_2$). But this would be wrong. There is still only one message, the first, and the information only comes from this first half. This illustrates how "information" is not simply a physical property of a message: it is a property of the message and your knowledge about it.

Perhaps the best way to demonstrate the difference between our definition of information and the everyday term is to consider a *random* message, that is, an N-bit binary string with random bits. If all possible strings are allowable messages, and all are equally likely (which will happen if each bit is equally likely to be 0 or 1), then the information in such a message will be:

$$I = \log_2(2^N) = N \tag{4.24}$$

This is actually the most information you can get with this particular choice of symbols. No other type of message will reach $I{=}N$. Now surely this doesn't make sense – how can a random string contain *any* information, let alone the maximum amount? Surely we must be using the wrong definition of "information"? But if you think about it, the N-bit strings could each label a message, as we discussed earlier, and receiving a particular string singles out which of the 2^N possible messages we could get that we are actually getting. In this sense, the string contains a lot of "information". Receiving the message changes your circumstance from not knowing what it was to now knowing what it is; and the more possible messages you could have received, the more "surprised", or enlightened, you are when you get a specific one. If you like, the difference between your initial uncertainty and final certainty is very great, and this is what matters.

4.6: Information Theory

We have defined the information in a message to be:

$$I = \log_2 n, \tag{4.25}$$

where n is the number of *equally likely* messages we might receive. Each message contains this same amount of information. In the general case, some messages will be more likely than others, and in this case, the greater the likelihood, the less information contained. This makes sense, given our claim that the information in a message represents the "surprise" we experience at receiving it. In this section, we come on to the topic of *information theory* proper, which will enable us to both generalize and make more rigorous our previous considerations.

We'll take a simple example first. Suppose our message is built from an alphabet of symbols. There could be any number of these, such as the four bases of DNA, or whatever: we certainly do not want to restrict ourselves to the letters of English. Let the number of symbols be i, and label them:

$$a_1,\ a_2,\ \ldots\ldots\ a_i.$$

Messages in this language are long strings of these symbols, say of length N. Now before we go any further, we have to make some assumptions about the way these symbols are distributed throughout messages. We assume firstly, that we can assign a probability, p_i, to each symbol, which is the probability that any given symbol in the message is the symbol a_i. The quantity p_i tells us the frequency of occurrence of a_i. We also assume that each symbol in the message is independent of every other; that is, which symbol appears at a given position does not depend on symbols at other positions, such as the one before. This is actually quite an unrealistic assumption for most languages. We will consider cases for which it is not true shortly.

How much information is carried by a given message? A simple way in which we can work this out is as follows. Suppose the message we have is length N. Then we would expect to find symbol a_1 turn up Np_1 times on average, a_2 Np_2 times, ... a_i Np_i times. The bigger N is, the better these guesses are. How many different messages do we have? Combinatorics comes to our rescue, through a standard formula. If we have N objects, m of one type, n of another, p of another, ..., and $m+n+p+... = N$, then the number of possible arrangements of the m, n, p... is given by:

$$\frac{N!}{m!n!p!...}. \tag{4.26}$$

On average, then, we can say that the number of different messages in N symbols is

$$\frac{N!}{(Np_1)!(Np_2)!...}. \tag{4.27}$$

We earlier defined information to be the base two logarithm of the number of possible messages in a string. That definition was based on the case where all messages were equally likely, but you can see that it is a good definition in the unequal probability case too. We therefore find the expected information in a message, which we write $<I>$, by taking the $\log_2$ of (4.27). Assuming N to be very large, and using Stirling's approximation, with which you should be familiar by now, we find:

$$<I> = N \sum_{i=1}^{M} (-p_i \log_2 p_i). \tag{4.28}$$

We can therefore obtain the *average information per symbol*:

$$<I>/N = \sum_{i=1}^{M} (-p_i \log_2 p_i). \tag{4.29}$$

This derivation appeals to intuition but it is possible to make it more rigorous. Shannon defined the information in a message to be the base two logarithm of the probability of that message appearing. Note how this ties in with our notion of information as "surprise": the less likely the message to appear, the greater the information it carries. Clearly, the information contained in one particular symbol a_n is:

$$-\log_2 p_n, \tag{4.30}$$

and if a message contains n_1 a_1's, n_2 a_2's, and so on, its information will be:

$$I = -\log_2[(p_1^{n_1})(p_2^{n_2}) \ldots (p_M^{n_M})] \tag{4.31}$$

which is:

$$-(n_1 \log_2 p_1 + n_2 \log_2 p_2 + \ldots n_M \log_2 p_M). \tag{4.32}$$

Incidentally, this shows that if we place two messages end to end, the total information they convey is twice that in the messages individually, which is satisfying. Check this for yourselves. Now the average information in a message is calculated in standard probabilistic fashion; it is just:

$$\begin{aligned} \textit{Average information} &= \textstyle\sum \textit{information in symbol } a_i \\ &\quad \cdot (\textit{expected number of appearances of } a_i) \qquad (4.33) \\ &= -\textstyle\sum (\log_2 p_i) \times (Np_i) \end{aligned}$$

which is our previous result. Incidentally, Shannon called this average information the "entropy", which some think was a big mistake, as it led many to overemphasize the link between information theory and thermodynamics[6].

Here is a nice, and slightly different, illustration of these ideas. Suppose we work for a telegraph company, and we send, with unequal probabilities, a range of messages — such as "Happy birthday", "Isn't Christmas wonderful", and so on. Each one of these messages has a probability P_m of being requested by a customer (m=1 to M, say). We can define two types of information here. There is that calculated by the person who receives the telegram — since it's my birthday it is not very surprising I get a "happy birthday" message, so there is not much information there. There is also the information as seen by the telegraphist who gets requested to send the message. It's interesting to look at the second type. To work this out, we would have to look at the operation of the telegraphy business for some time, and calculate the proportions of each type of message that are sent. This gives the probability P_m of message m being sent and we can treat each message as the symbol of some alphabet, similar to labeling each one, rather like the parent-student card we looked at earlier. We can hence calculate the information from the viewpoint of the telegraphist.

4.7: Further Coding Techniques

Let me now return to the topic of coding and describe a couple of popular techniques for coding messages to show you some of the wonderful and ingenious ways in which this can be done. These codes are unlike those we've considered so far in that they are designed for messages in which the symbol probabilities vary.

[6]Legend has it that Shannon adopted this term on the advice of the mathematician John Von Neumann, who declared that it would give him ". . . a great edge in debates because nobody really knows what entropy is anyway." [RPF]

4.7.1: Huffman Coding

Consider the following system of eight symbols, where by each I have written the probability of its occurrence (I have arranged the probabilities in descending order of magnitude):

E	0.50
THE	0.15
AN	0.12
O	0.10
IN	0.04
R	0.04
S	0.03
PS	0.02

The probabilities add to one, as they should. A sample message in this system might be:

ANOTHER

which has probability 0.12 x 0.10 x 0.15 x 0.04. Now we notice that the symbol E appears much more often than the others: it turns up 25 times as often as the symbol PS, which takes twice as much effort to write. This symbol system doesn't look very efficient. Can we write a new code that improves it? Naively, we might think: "Well, we have eight symbols, so let's just use a three-bit binary code." But since the E occurs so often, would it not be better to describe it, if we can, by just *one* bit instead of three? We might have to use more bits to describe the other symbols, but as they're pretty rare maybe we might still gain something. In fact, it is possible to invent a *non-uniform* code that is much more efficient, as regards the space taken up by a message, than the one we have. This will be an example of compression of a code. Morse had this idea in mind when he assigned a single "dot" to the common E but "dash dash dot dash" to the much rarer Q.

The idea is that the symbols will vary in their lengths, roughly inversely according to their probability of appearance, with the most common being represented by a single symbol, and with the upshot that the typical overall message length is shortened. We will actually replace our symbols by binary strings. The technique I will outline for you — I will let you figure out for yourselves why it works — is due to Huffman. It is quite a popular method, although I believe frequently costly to implement. It is a two-stage process, and is best understood by considering the following tree diagram, where initially the symbols are arranged in ascending order of probabilities (Fig. 4.3):

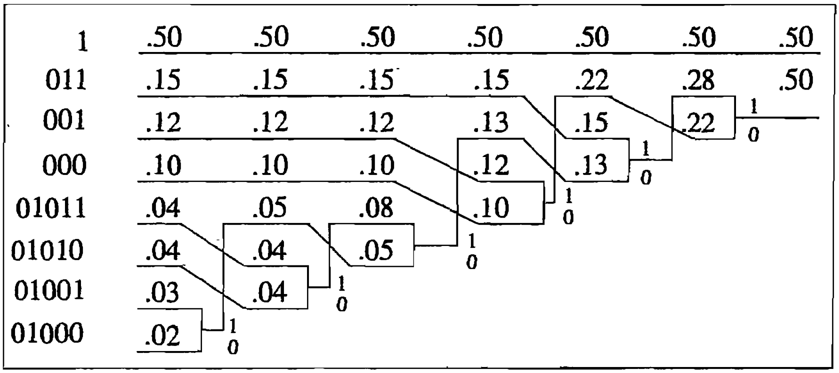
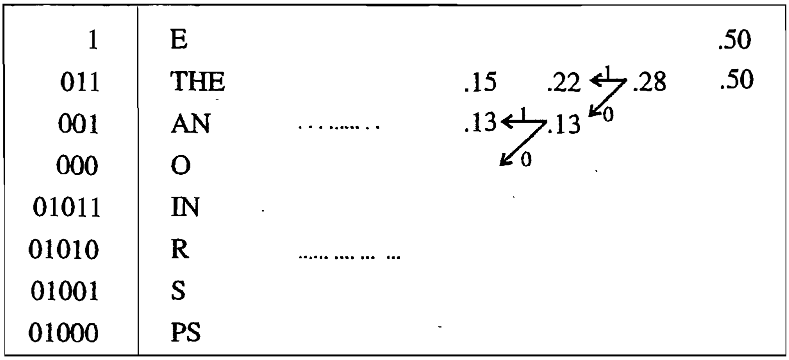


Fig. 4.3 Huffman Coding Tree

Begin by coalescing the two lowest probability symbols into one, adding the probabilities. We can now pretend that we have a source alphabet consisting of the original symbols, less the lower two, plus a new "joint" symbol with probability of occurrence (in this case) .05. Redraw the column, placing the joint symbol at its appropriate point in the probability hierarchy, as shown in Figure 4.3. Now iterate. Coalesce the next two to shrink the list further. Continue in this vein until we reach the right hand of the tree, where we have an "alphabet" of two symbols, the original maximally probable one, plus a summed "joint" symbol, built from all the others.

To figure out the actual assignment of coding symbols, we now retrace a path back through the tree. The rule is straightforward: at each branch in the path required to get back to the original symbol, you add a digit to its code. If you follow the upper path at the branch, you add a one; a lower branch gives you a zero (this is purely a matter of convention). You move from right to left across the tree, but the code you write out from left to right as you go. What is happening is shown in Figure 4.4:



1	E
011	THE
001	AN
000	O
01011	IN
01010	R
01001	S
01000	PS

Fig. 4.4 "Trellis" for Huffman Coding

Let us look at the code for "THE". To get to it, we have to start with a 0. We follow the upper path from the first branch, giving us 01 so far. Then, again, we have to follow the upper path from the next branch. We end up with 011, which is the code for THE. The other codes are as shown above. It is worth pointing out that other Huffman codes can be developed by exploiting the ambiguity that occasionally arises when a joint probability at some point equals one of the original probabilities. Do we put the joint above its equal in the table, or beneath? You might like to think about this.

We can easily calculate the length saving of this code in comparison with a straight three-bit code. With three bits, the average length of a symbol is obviously three! With this Huffman code, the average symbol length is:

$$(1 \times 0.5) + 3 \times (0.15 + 0.12 + 0.10) + 5 \times (.09+.04+.03+.02)$$
$$= 2.06.$$

which is a saving of nearly a third!

There is a nice subtlety in constructing non-uniform codes that the Huffman method takes care of nicely. It has the property that *no code word is the prefix of the beginning of any other code word.* A little thought shows that a code for which this is not true is potentially disastrous. Suppose we had the following set of symbols:

1, 01, 10, 101, 010, 011.

Try and decode this message: 011010110. You can't do it! At least, not uniquely. You do not know whether it is 01-1-01-01-10 or 011-01-01-10 or 01-101-01-10 or another possibility. There is an ambiguity due to the fact that the symbols can run into each other. A good, uniquely decodable symbol choice is necessary to avoid this, and Huffman coding is one way forward. You can check that the code choice we have found for our symbols leads to unique decoding.

Problem 4.4: Huffman coding differs from our previous coding methods in that it was developed for compression, not error correction. The technique gives us nicely-packed codes, but they are quite sensitive to errors. If we have the following message:

00100001101010 (= ANOTHER)

then a single glitch can easily result in swapped symbols. For example, an error in position 2 would give us THEOTHER[7]. This throws up an interesting question that you might like to address. For general non-uniform coding, what is the greatest number of symbol errors that can be generated by a one-bit mistake? You are used to thinking of one error — one bit, but with non-uniform coding that might not be true. For example, might it not be possible that a single error might change one symbol to one of a different length, and that this will affect the next symbol, and the next, and so on, so that the error propagates throughout the entire message string? It turns out not. Can you figure out why?

4.7.2: Predictive Coding

Thus far, I have only considered situations in which the probabilities of symbols occurring in a message are independent: symbols exert no influence across the message. However, as I have stressed by the example of English, such dependence is extremely common. The full mathematical treatment of source alphabets comprising varying probabilities of appearance and intersymbol influence is quite complex and I will not address it here. However, I would like to give you some flavor of the issues such influence raises, and I will do this by considering *predictive coding*. This is another way of compressing codes, rather than correcting them.

[7]It could be said that too many glitches would drive us 01011010010011. [RPF]

Let us suppose that we have a source alphabet which is such that, if we know the contents of a message up to a point, we can predict what the next symbol will be. This prediction will not typically be certain — it will be probabilistic — and the details of how we make it do not matter. The method we use might require the knowledge of only the previous symbol, or the previous four, or even the whole message. It does not matter. We just have some rule that enables us to predict the next symbol.

Imagine now that we do our predicting with a predictor, a black box that we stick next to the message to be sent, which contains some formula for making predictions. Here is how things work. The predictor is fully aware of the message that has been sent so far (which we illustrate by feeding a message line into it), and on the basis of this it makes a prediction of what symbol is to be sent next. This prediction is then compared with the actual source symbol that comes up. If the prediction is right, then we send a zero along the transmission channel. If the prediction is wrong, we send a one. The easiest way to implement this is to bitwise add the source symbol and the prediction and ignore any carry. Schematically, we have, at the transmission end (Fig. 4.5):

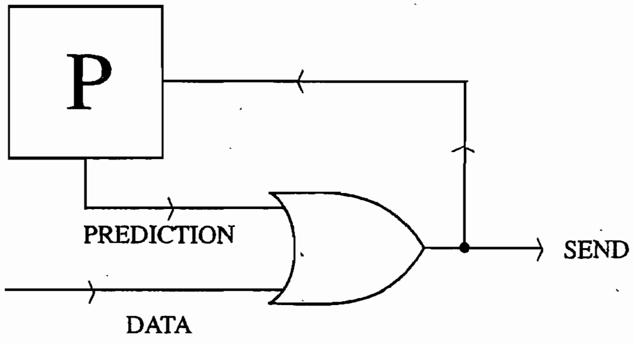


Fig. 4.5 A Predictive Encoder

Note that we have incorporated a feedback loop to let the predictor know whether its prediction was correct or not. A good predictor will produce a long string of zeroes, with the occasional one interspersed where it made a mistake: a random predictor, one that just guesses, will have ones and zeroes split fifty-fifty, if that is the base-rate in the source. It's not difficult to see how, if we send this string, we can reconstruct the original message at the other end by using an identical predictor as a decoder. It simply works backwards. This is all very nice, of course, but what is the point of this rather exotic procedure? Well, if the first predictor is good, making pretty accurate predictions, then it will generate far more zeroes than ones. Interspersed between the ones will be long runs of zeroes. The key is this — when sending the message, we do not send these runs: *instead we send a number telling us how many zeroes it contained.* We do this in binary, of course. If there is a run of twenty two zeroes before the next one digit, we don't send out:

0000000000000000000000

but rather its binary equivalent:

10110.

That's some saving of transmission space! All we have to do is get the guy at the receiving end to break the binary numbers down into strings of zeroes, and use his predictor to figure out what we were saying. Predictive coding enables us to compress messages to quite a remarkable degree.

Problem 4.5: An interesting problem with which you can entertain yourself is how to compress things still further. The average length of the runs of zeroes is dependent on how good the predictor is. Once we know how good it is, we can work out the probability that a run will have a given length. We can then use a Huffman technique to get an even tighter code! Work out the details if we are sending an equally likely binary code, and the probability of the predictor being wrong in its prediction is q. You can get pretty close to Shannon's limit using compression of this sort.

4.8: Analogue Signal Transmission

I would like to discuss one more coding problem before leaving the subject. This is the question of how we can send information that is not naturally in the form of bits; that is, an analogue signal. Ordinarily, information like the oil pressure in a car, the torque in a drive shaft, the temperature variation on the Venusian surface, is *continuous*: the quantities can take any value. If we only have the capacity to transmit bits, how do we send information of this kind? This is not a matter of fundamental principle; it is actually a practical matter. I will say a few words about it despite the fact it is somewhat peripheral. You could say that the whole course is somewhat peripheral. You just wait!

Let us suppose for starters that our continuous quantity — S, say — is restricted to lie between 0 and 1:

$$0 \le S \le 1 \tag{4.34}$$

The secret of sending the value of S is to approximate it. The most important question to ask is with what accuracy we want to send data. Suppose we want S to within 1%. Then, all we need do is split the interval [0,1] up into one hundred slices (usually referred to as "bins"), and transmit information about which slice the value of S is in; in other words, a number between 0 and 100. This is easy. However, as we prefer to use binary, it is better to split the range of S into 128 slices ($=2^7$), and send the S value as a 7-bit number. Similarly, if we want to send S to an accuracy of one part in a thousand, we would send a 10-bit number, having split [0,1] into 1024 bins.

What happens if the variable S is unbounded? This is not uncommon. Usually, such a variable will have values that are not evenly distributed. In other words, it will be more likely to have some values rather than others (very, very few physical quantities have flat probability distributions). We might have a variable with a probability distribution such as that shown in Figure 4.6:

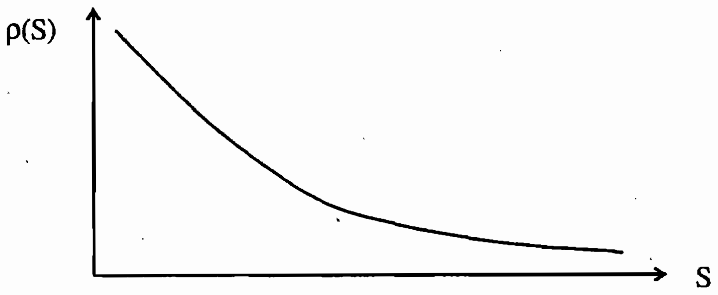


Fig. 4.6 A Sample Probability Distribution for a Physical Variable

The probability density $\rho(S)$ has the usual definition: if we make a measurement of S, the probability of finding its value to lie between S_1 and S_2 is:

$$\int_{S_1}^{S_2} \rho(S)\ dS \qquad (4.35)$$

or, if S_1 and S_2 lie infinitesimally close to one another, $S_2 = S_1 + \delta s$:

$$\rho(S_1).\delta s \qquad (4.36)$$

The basic idea for transmitting S in this general case is the same. We divide the effective range of S into a number of bins with the important difference that we size these bins so that they are all of equal probability (Fig. 4.7):

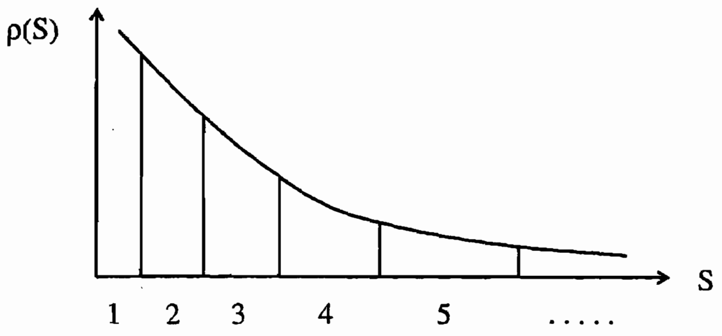


Fig. 4.7 Division of ρ(S) into Equal Volume Bins

Clearly the bins are of different width, but they are chosen to have the same area when viewed probabilistically. They are defined by the formula:

$$(1/128)\int_{S_i}^{S_{i+1}} \rho(S)\ dS \qquad (4.37)$$

where i runs from 0 to 127, and the ith bin corresponds to the S-values S_i to S_{i+1}.

Alternatively, we can make a change of variables. For each value s of S we can define the function $P(s)$ by:

$$P(s) = \int_0^s \rho(S)\, dS \tag{4.38}$$

$P(s)$ is just the cumulative probability function of S, the probability that $S \leq s$. It clearly satisfies the inequality ($0 \leq P \leq 1$). One well-known statistical property of this function (as you can check) is that its own distribution is *flat*: that is, if we were to plot the probability distribution of $P(s)$ as a function of s in Figure 4.6, we would see just a horizontal line. A consequence of this is that if we make equal volume bins in P, they will automatically be of equal width. That takes us back to the first case.

A different, but related, problem is that of transmitting a function of time (Fig. 4.8):

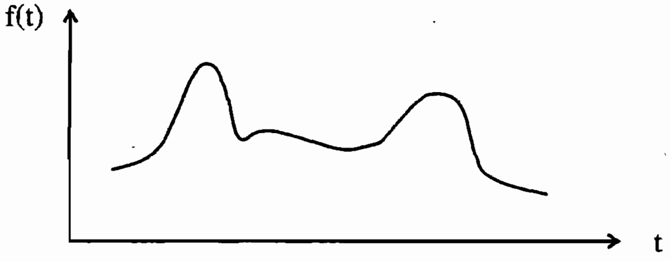


Fig. 4.8 A Typical Function of Time

Consideration of such a problem will bring us on to consider the famous *Sampling Theorem*, another baby of Claude Shannon. The basic idea here would be to sample the function at certain regular time intervals, say τ, and send the value of the function at each time digitally. The receiver would then attempt to reconstruct the original signal from this discrete set of numbers. Of course, for a general function, the receiver will have to smooth out the set, to make up for the "gaps". However, for certain types of continuous function, it is actually possible to sample in such a way as to encode *completely* the information about the function: that is, to enable the receiver to reconstruct the source function exactly! To understand how it is possible to describe a continuous function with a finite number of numbers, we have to take a quick look at the mathematical subject of Fourier analysis. I will cover this because I think it is interesting;

those without the mathematical background might wish to skip it!

It turns out that if the "Fourier transform" of the function $g(\omega) = 0$ for all $|\omega| \geq \upsilon$, and we sample at intervals of $\tau = \pi/\upsilon$, then these samples will completely describe the function. What does this mean? Recall that, according to Fourier theory, any periodic function $f(t)$ can be written as a sum of trigonometric terms. For a general function of time, $f(t)$, we have:

$$f(t) = (1/2\pi)\int_{-\infty}^{\infty} g(\omega)e^{-2\pi i\omega t}\, d\omega \qquad (4.39)$$

where $g(\omega)$ is the *Fourier Transform* of $f(t)$. What we have effectively done here is split $f(t)$ up into component frequencies, suitably weighted. Now the typical function (signal) that is encountered in communication theory has a limited *bandwidth*; that is, there is an upper limit to the frequencies that may be found in it (for example, the channel through which the signal is sent might not be able to carry frequencies above a certain value). In such a case, the limits of integration in (4.39) become finite:

$$f(t) = (1/2\pi)\int_{-W(\upsilon)}^{W(\upsilon)} g(\omega)e^{-2\pi i\omega t}\, d\omega, \qquad (4.40)$$

where W is the bandwidth, and υ is now the highest frequency in the Fourier expansion of $f(t)$[8].

It is possible (the math is a bit tough) to show that this expression reduces to the infinite sum over the integers:

$$f(t) = \sum_{n=-\infty}^{\infty} f(n\pi/\upsilon).[\sin\pi(\upsilon t - n\pi)]/(\upsilon t - n\pi) \qquad (4.41)$$

This is the Sampling Theorem. If you look at this expression, you will see that as long as we know the values of the function $f(t)$ at the times:

[8]Conventionally, the bandwidth W is given by $W=\upsilon/2\pi$. [RPF]

$$t = n\pi/\upsilon, \tag{4.42}$$

where n is an integer, then we can work it out at all other times, as a superposition of terms weighted by the signal samples. This is a subtle consequence of using the well-known relation:

$$(\sin x)/x \to 1 \textit{ as } x \to 0 \tag{4.43}$$

in (4.41): setting $t = n\pi/\upsilon$ in the summand, we find that all terms except the n^{th} vanish: the n^{th} is just unity multiplied by the value of f at $t = n\pi/\upsilon$. In other words, if we sampled the function at times spaced (π/υ) time units apart, we could reconstruct the entire thing from the sample! This finding is of most interest in the physically meaningful case when the function $f(t)$ is defined only over a finite interval (0,T). Then, the sum (4.41) is no longer infinite and we only need to take a finite number of sample points to enable us to reconstruct $f(t)$. This number is $(T\upsilon/\pi)$.

Although I have skated over the mathematical proof of the Sampling Theorem, it is worth pausing to give you at least some feel for where it comes from. We are sampling a function $f(t)$ at regular intervals, τ. The graph for the sampled function arises from multiplying that of the continuous $f(t)$ by that of a spikey "comb" function, $C(t)$, which is unity at the sample points and zero elsewhere (Fig. 4.9):

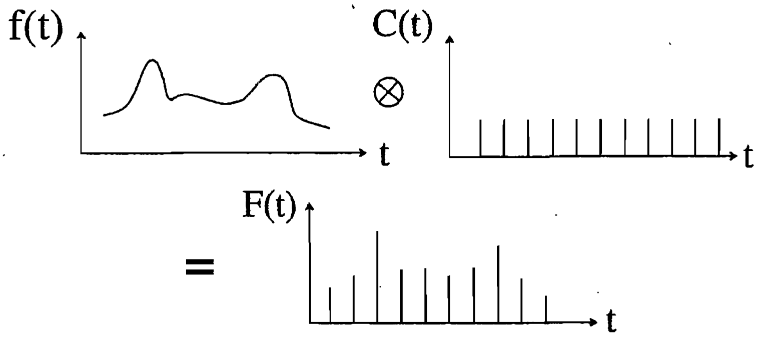


Fig. 4.9 The Sampled Function

Now, corresponding to $f(t)$ is a Fourier Transform $\phi(\omega)$. $C(t)$ also has an associated transform, $\chi(\omega)$ another comb function (Fig. 4.10):

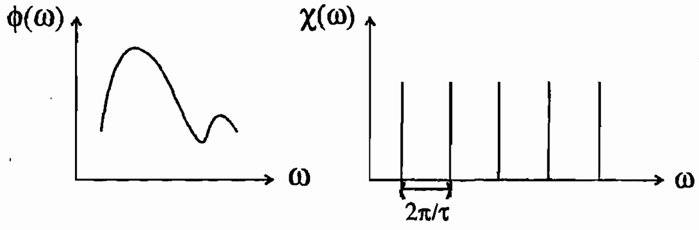


Fig. 4.10 Fourier Transforms of f(t) and C(t)

The transform χ is actually a set of equally-spaced delta functions ($2\pi/\tau$ apart). The Fourier transform of the sampled function, $F(t)$, is obtained by the process of "convolution", which in crude graphical terms involves superposing the graph of ϕ with that of χ. We find that the transform of $F(t)$ comprises copies of the transform of $f(t)$, equally-spaced along the horizontal axis, but scaled in height according to the trigonometric ratio in (4.41) as in Figure 4.11:

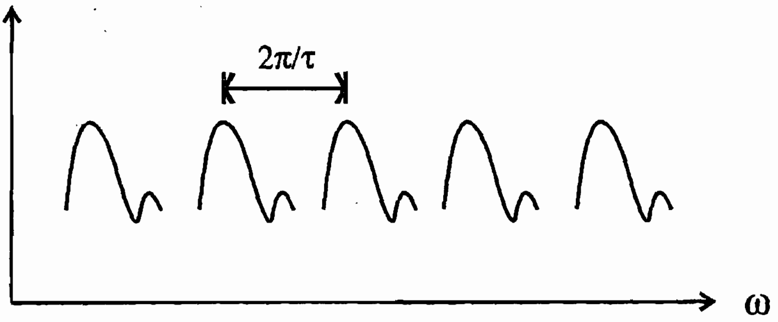


Fig. 4.11 The Fourier Transform of the Sampled Function

Look closely at this graph. What it is telling us is that information about the whole of $f(t)$ could, in principle, be extracted from $F(t)$ alone. There is as much information in one of the Fourier-transformed bumps in Figure 4.11 as there is in the whole of Figure 4.9! As the former transform comes solely from the sampled function $F(t)$, we can see the basic idea of the Sampling Theorem emerging.

An interesting subtlety occasionally arises in the sampling process. The Sampling Theorem tells us that, if a signal has an upper frequency limit of ν (i.e. a bandwidth of $\nu/2\pi$) then we need at least $(T\nu/\pi)$ sample points to enable us to reconstruct the signal. If we take more points than this, all well and good. However, if we take fewer (and this can arise by accident if the function $f(t)$ has "tails" that lie outside the interval (0,T)), our sampling will be insufficient. Under such circumstances we get what is known as *aliasing*. The sampling interval will be too coarse to resolve high frequency components in $f(t)$, instead mapping them into low frequency components — their "aliases". A familiar example of this phenomenon can be found in movies. Movies, of course, are samples — 24 times a second, we take a snapshot of the world, creating the illusion of seamless movement to our eyes and brains. However, evidence that sampling has occurred often shows up. Maybe the best known is the behavior of wagon wheels in old westerns. As kids we all noticed that, when a wagon started moving, at first the spokes in the wheels seemed to go around the right way. Then, as things sped up, they appeared to stop rotating altogether. Finally, as things sped up still further, the wheels appeared to be going the wrong way around! The explanation for this phenomenon lies in inadequate sampling. Another example of aliasing occurs when we inadequately sample audio data, and end up with frequencies that we cannot ordinarily hear being aliased into ones we can. To avoid aliasing, we would need to filter out of the signal any unduly high frequencies before we sampled. In the case of the movies, this would mean taking pictures with a wider shutter, so that the picture is just a blur, or smoothed out.

It is now possible to send sound *digitally* — sixteen bits, 44.1 kHz reproduces perfectly, and is pretty resistant to noise; and such a method is far superior to any analog technique. Such developments will transform the future. Movies will be cleaned up, too — optical fibers, for example, are now giving us *over*capacity. The soap ad will appear with absolute clarity. It seems that the technological world progresses, but real humanistic culture slides in the mud!

FIVE

REVERSIBLE COMPUTATION AND THE THERMODYNAMICS OF COMPUTING

I would now like to take a look at a subject which is extremely interesting, but almost entirely academic in nature. This is the subject of the energetics of computing. We want to address the question: *how much energy must be used in carrying out a computation?* This doesn't sound all that academic. After all, a feature of most modern machines is that their energy consumption when they run very fast is quite considerable, and one of the limitations of the fastest machines is the speed at which we can drain off the heat generated in their components, such as transistors, during operation. The reason I have described our subject as "academic" is because we are actually going to ask another of our fundamental questions: what is the *minimum* energy required to carry out a computation?

To introduce these more physical aspects of our subject I will return to the field covered in the last chapter, namely the theory of information. It is possible to treat this subject from a strictly physical viewpoint, and it is this that will make the link with the energy of computation.

5.1: The Physics of Information

To begin with, I would like to try to give you an understanding of the physical definition of the information content of a message. That physics should get involved in this area is hardly surprising. Remember, Shannon was initially interested in sending messages down real wires, and we cannot send messages of any kind without some interference from the physical world. I am going to illustrate things by concentrating on a particular, very basic physical model of a message being sent.

I want you to visualize the message coming in as a sequence of boxes, each of which contains a single atom. In each box the atom can be in one of two places, on the left or the right side. If it's on the left, that counts as a 0 bit, if it's on the right, it's a 1. So the stream of boxes comes past me, and by looking to see where each atom is I can work out the corresponding bit (Fig. 5.1):

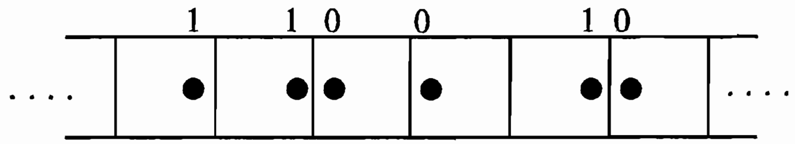


Fig. 5.1 A Basic Atomic Message

To see how this model can help us understand information, we have to look at the physics of jiggling atoms around. This requires us to consider the physics of gases, so I will begin by taking a few things I need from that. Let us begin by supposing we have a gas, containing N atoms (or molecules), occupying a volume V_1. We will take this gas to be an exceptionally simple one; each atom, or molecule, within it (we take the terms to be interchangeable here) is essentially free — there are no forces of attraction or repulsion between each constituent (this is actually a good approximation at moderately low pressures). I am now going to shrink the gas, pushing against its volume with a piston, compressing it to volume V_2. I do all this isothermally: that is, I immerse the whole system in a thermal "bath" at a fixed temperature T, so that the temperature of my apparatus remains constant. Isn't it wonderful that this has anything to do with what we're talking about? I'm going to show you how. First we want to know how much work, W, it takes to compress the gas (see Fig. 5.2):

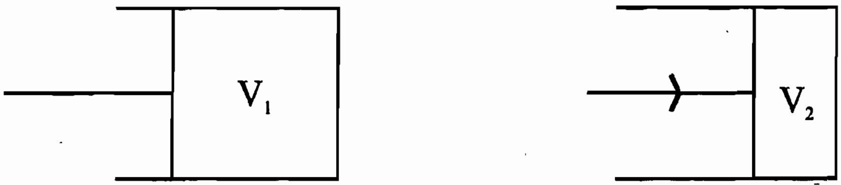


Fig. 5.2 Gas Compression

Now a standard result in mechanics has it that if a force F moves through a small distance δx, the work[1] done δW is:

$$\delta W = F\ \delta x \tag{5.1}$$

If the pressure of the gas is p, and the cross-sectional area of the piston is A, we can rewrite this using $F = pA$ and letting the volume change of the gas $\delta V = A\delta x$ so that:

$$\delta W = p\ \delta V. \tag{5.2}$$

Now we draw on a standard result from gas theory. For an ideal gas at pressure p, volume V and temperature T, we have the relation:

$$pV = NkT \tag{5.3}$$

where N is the number of molecules in the gas and k is Boltzmann's constant (approximately 1.381×10^{-23} J K^{-1}). As T is constant — our isothermal assumption — we can perform a simple integration to find W:

$$W = \int_{V_1}^{V_2} \frac{NkT}{V}\, dV = NkT \log \frac{V_2}{V_1}. \tag{5.4}$$

(Here, $\log x = \log_e x$.) Since V_2 is smaller than V_1, this quantity is negative, and this is just a result of the convention that work done on a gas, rather than by it, has a minus sign. Now, ordinarily when we compress a gas, we heat it up. This is a result of its constituent atoms speeding up and gaining kinetic energy. However, in our case, if we examine the molecules of the gas before and after compression, we find no difference. There are the same number, and they are jiggling about no more or less energetically than they were before. There is no difference between the two at the molecular level. So where did the work go? We put some in to compress the gas, and conservation of energy says it had to go somewhere. In fact, it *was* converted into internal gas heat, but was promptly

[1]Another one of those awkward words, like "information". Note that, with this definition, a force must move through a distance to perform work; so it does not take any of this kind of "work" to hold up a suitcase — only to lift it! [RPF]

drained off into the thermal bath, keeping the gas at the same temperature. This is actually what we mean by isothermal compression: we do the compression slowly, ensuring that at all times the gas and the surrounding bath are in thermal equilibrium.

From the viewpoint of thermodynamics, what we have effected is a "change of state", from a gas occupying volume V_1 to one occupying volume V_2. In the process, the total energy of the gas, U, which is the sum of the energies of its constituent parts, remains unchanged. The natural thermodynamical quantities with which such changes of state are discussed are the *free energy* F and the *entropy* S, which are related by:

$$F = U - TS. \tag{5.5}$$

The concept of free energy was invented to enable us to discuss the differences between two states even though there might be no actual mechanical differences between them. To get a better feel for its meaning, look at how expression (5.5) relates small variations at constant temperature:

$$\delta F = \delta U - T\delta S. \tag{5.6}$$

For the change under consideration, the total gas energy remains constant, so $\delta U=0$ and $\delta F = -T\,\delta S$. δF is just the "missing" heat energy siphoned off into the heat bath, $NkT\,\log(V_1/V_2)$, and we use this to write (5.6) as an entropy change:

$$\Delta S = Nk \log \frac{V_2}{V_1}. \tag{5.7}$$

Note that as we are dealing with a finite change here, we have replaced the infinitesimal δ with a finite Δ.

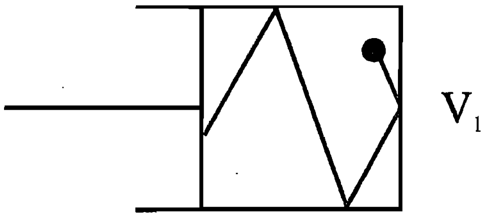
Entropy is a rather bizarre and counter-intuitive quantity, and I am never sure whether to focus on it or on the free energy! For those who know a little thermodynamics, the general equation $\delta S = -\delta F/T$ is a variant of the standard formula $\delta S = \delta Q/T$ for the infinitesimal change in entropy resulting from a thermodynamically reversible change of state where, at each stage, an amount of heat δQ enters or leaves the system at absolute temperature T. For an irreversible process, the equality is replaced by an inequality, ensuring that the entropy of an isolated system can only remain constant or increase — this is the Second Law of Thermodynamics. I'll say a little more about entropy in a moment.

Now we take a bit of a leap, and it is not obvious that we can do this, but we can. We consider the case where our gas contains only one molecule. That is, we put N=1 into our formulae. Now it's difficult to get a feeling for concepts like temperature, pressure and volume, never mind free energy and entropy, when you only have one molecule! However, these concepts make sense as long as we consider them to be time averaged, smoothing out the irregularities of this one particle as it bounces back and forth. Indeed, our formulae actually work with N=1, as long as there is this hidden smoothing. The situation is more fun, too!

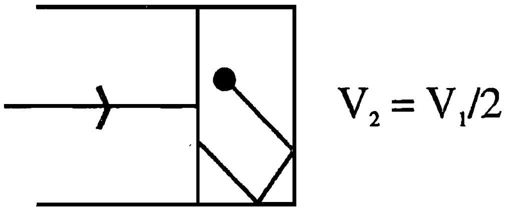
Let us suppose that we are halving the volume occupied by the molecule: $V_2 = V_1/2$. We then find that the free energy and the entropy of the particle change by:

$$+ kT \log 2 \textit{ and } -k \log 2 \qquad (5.8)$$

respectively. What does this mean? Pictorially, the situation has changed from:



to:



The physical state of the molecule before and after the compression appears to be the same — its actual (kinetic) energy has not changed, for example — yet for some reason we have a change in these quantities F and S. What has happened, and this is very subtle, is that my *knowledge of the possible locations of the molecule has changed.* In the initial state, it could be hiding anywhere in volume V_1: after the compression, it must be somewhere within V_2. In other words, there are fewer places it can be in.

This concept of "knowledge" is extremely important, and central to the concept of entropy, so I will dwell on it awhile. It arises from the deeply statistical nature of thermodynamics. When doing the mathematics of vast numbers of particles that make up gases, we cannot practically follow the paths and momenta of every molecule in the gas, so we are forced to turn to probability theory. Concepts such as temperature and pressure of a gas are essentially defined to be statistical averages. We assign certain physical properties to each molecule, assume particular distributions for these molecules, and calculate the average by a weighting process: so many molecules will move this fast, corresponding to one temperature; so many will move that fast, giving another temperature; and we just average over everything. The entropy of a gas is defined statistically, indeed this is its core definition, but in a different way to quantities such as temperature and energy. Unlike these, it is not a macroscopic property that arises from a sum of microscopic properties. Rather, it is directly related to the *probability that the gas be in the configuration in which it is found.* By "configuration" I mean a particular arrangement, or cluster of arrangements, of positions and momenta for each of the N constituent molecules (or, if you want to be fancy, a particular point or region in "phase space"). The existence of such a probability should not come as too much of a surprise: if you look at any given gas it is far less likely at the outset that you will find all the molecules moving in the same direction or paired up and dancing than you will find them shooting all over the place at random. Entropy quantifies this notion. Loosely speaking, if the probability of a particular gas configuration is W, we have:

$$S \approx k \log W. \tag{5.9}$$

The bigger W, the bigger the entropy, and, like all probabilities, the W's add, so we can straightforwardly calculate the chances of being in some range of configurations. The gas with molecules going all one way has a W much less than that of the one with a more random — or *more disordered* — structure, and hence has a lower entropy. What has all this got to do with our knowledge of a system? Simply, the less we know about the configuration of a gas, the more states it could be in, and the greater the overall W – and the greater the entropy. This gives us a nice intuitive feel of what is happening when we compress a gas into a smaller volume. Working isothermally, the momenta of the molecules within the container remains the same ($\delta U=0$), but each molecule has access to fewer possible spatial positions. The gas has therefore adopted a configuration with smaller W, and its entropy has decreased. As an aside, the Second Law of Thermodynamics tells us that in any isolated system:

$$\delta S \approx k \, \delta W/W \geq 0, \tag{5.10}$$

i.e. the entropy never decreases. The fact that the entropy of our compressed gas has dropped is a reminder that the system is not isolated – we have been draining heat into a heat bath. The heat flow into the bath increases its entropy, preserving the Second Law. Generally speaking, *the less information we have about a state, the higher the entropy.*

As the definition of entropy is essentially statistical, it is perfectly all right to define it for a gas with a single molecule, such as the one we have been considering, although there are a few subtleties (which we will avoid). You can see that if we compress the volume by a factor of 2, then we halve the number of spatial positions, and hence the number of configurations that the molecule can occupy. Before, it could be in either half of the box: now, it can only be in one half. You should be able to see in our probabilistic picture how this leads to a decrease in entropy by an amount:

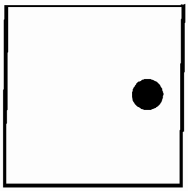
$$\delta S = k \log 2 \tag{5.11}$$

This is the same as we obtained with our work and free energy considerations.

We can now return to the topic of information and see where all this weird physics fits in. Recall the atomic tape with which we opened this section, in which the position of atoms in boxes tells us the binary bits in the message. Now if the message is a typical one, for some of these bits we will have no prior knowledge, whereas for others we will – either because we know them in advance, or because we can work them out from correlations with earlier bits that we have examined. We will *define* the information in the message to be *proportional to the amount of free energy required to reset the entire tape to zero*. By "reset to zero", we mean compress each cell of the tape to ensure that its constituent atom is in the "zero" position.

Straightaway, we note what seems to be an obvious problem with this definition, namely, that it introduces an unnatural asymmetry between 0 and 1. If an atom is already in the zero part of the compartment, then surely the reset operation amounts to doing nothing, which costs no free energy. Yet if it is in the one position in the compartment, we have to do work to move it into the zero position! This doesn't seem to make sense. One would expect to be able to introduce an alternative definition of information for which the tape is reset to one — but then we would only seem to get the same answer if the message contained an equal number of ones and zeroes! But there is a subtlety here. Only if we *do not know* which side of the compartment the atom is in do we expend free energy. It is only in this circumstance that the phase space for the atom is halved, and the entropy increases. If we know the atom's position, then we expend *no* energy in resetting, irrespective of where the atom starts out. In other words, as one would hope, the information in the message is contained in the surprise bits. Understanding why this is so is worth dwelling on, as it involves a style of argument often seen in the reversible computing world. It seems a bit counter-intuitive to claim that the energy required to reset a one to a zero is no more than leaving a zero alone — in other words, nothing.

To clear this point up, I first have to stress the idealized nature of the set-up we are considering. Although I have talked freely about atoms in "boxes", these boxes are not real boxes made of cardboard and strung together, with mass and kinetic and potential energy. Moreover, when I talk about "energy", I certainly don't mean that of the tape! We are only interested in the content of the message, which is specified by the positions of the atoms. Let us suppose we have a message bit that we know is a one — the atom is on the right hand side — so we have the following picture:



We can show that to reset this to zero costs no energy in several ways. One pretty abstract way is to first slip in a little partition to keep the atom in place. All I have to do now is *turn the box over*. The end result is that we now have a zero on the right hand side (Fig. 5.3):

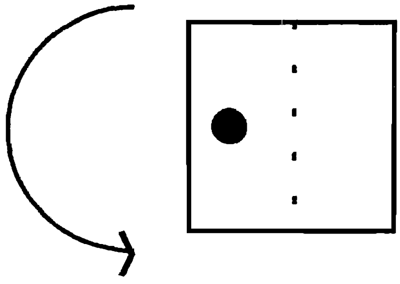


Fig. 5.3 A Simple Reset Procedure

This is abstract because it might seem odd to be able to insert pistons and turn boxes without expending energy. In the real world, of course, you can't — but we are dealing with abstractions here and, as I have said, we are not interested in the kinetic energy or weight of the "boxes". Given our assumptions, it is possible to do so, although the downside is that we would have to take an eternity to do it! (We will return to this sort of argument in §5.2.) Another way, perhaps a little less abstract, would be to introduce two pistons, one on each side of the box, and push the atom over with one, while drawing the other out (Fig. 5.4):

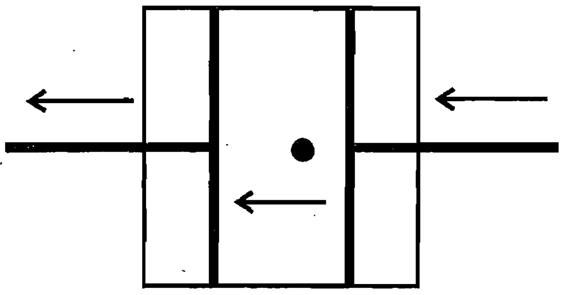


Fig. 5.4 A More "Realistic" Reset

Now the bombardment on the left is equal to that on the right, and any work put in at one end will be taken out at the other, and so is recovered. One could even join the pistons by an up-and-over rod, and you should be able to see that the tiniest touch on one piston will send the whole thing coasting over to its final position. So, if you do it slowly enough — "infinitesimal in the limit" — no work is done in resetting. Clearing, or resetting, the tape is what occurs when we don't know what compartment the atom is in. Then we must perform a compression, and this will take free energy, as we discussed earlier, as we are lessening our ignorance of the atom's position.

Another way of looking at these ideas is due to Bennett, who suggests using a message tape as *fuel*, and relates the information in the tape to its fuel value — that is, to the amount of energy we can get from it. His idea, which is quite subtle, goes as follows. We suppose we have a machine, in contact with some kind of heat bath, which takes in tapes at one end, and spits them out at the other. We assume to begin with, that the tape the machine eats is blank, i.e. all of its atoms are in the zero state. We will show how such a tape can be used to provide us with useful work, which we can use to power our machine.

What we do is incorporate a piston into the system. As each cell comes in, we bring the piston into it, up to the halfway position in each box (Fig. 5.5):

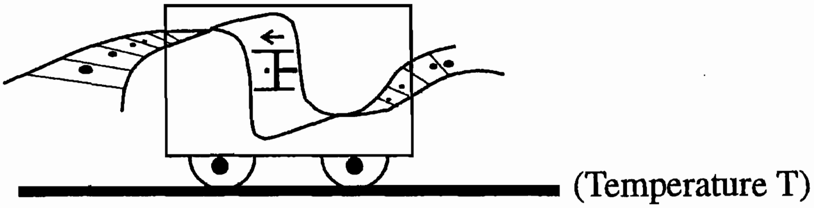


Fig. 5.5 An Information-driven Engine

We now let the heat bath warm the cell up. This will cause the atom in the cell to jiggle against the piston, isothermally pushing it outwards as in Figure 5.6:

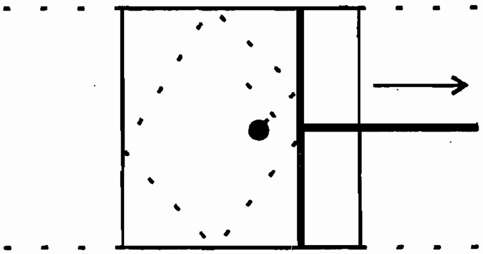


Fig. 5.6 Work Generation Mechanism in the Engine

This is just the opposite process to the compression of a gas we considered at the beginning of this section. The net result is that work is done on the piston which we can subsequently extract: in other words, we can get our tape to do work for us. You should be able to see that for a tape of n bits this work is equal to $nkT\log2$, the free energy, where T is the temperature of the heat bath. An important consequence of our procedure is that the tape that the machine spits out has been *randomized*: after the piston has been pushed out, the atom that did the pushing can be anywhere in that cell, and we have no way of knowing where, short of performing a measurement.

We now generalize the argument by assuming that our piston is maneuverable. This allows us to extract work from tapes which have a 1 in them. If we get a 1, we switch the piston to the other side of the cell, bring it up to the edge of the 1 half, and proceed as before. Again we get $kT\log2$ of useful work given out, and again the tape that emerges from the machine is randomized. What is crucial here is that we *know* what bit is about to enter the machine. Only then can we ready the piston to ensure that it does work for us. Obviously, if we left the piston in the 0 position, and we got a 1 in, we would actually have to do work to shift the atom into the 0 cell, and when the atom expands back into the full cell we would get that work back: that is, no useful work would be done. Clearly, *a random tape has zero fuel value*. If we do not know what bit is coming in next, we do not know how to set our piston. So we would leave it in one position, and just push it in and hope, push it in and hope, boom, boom, boom. Sometimes we would get lucky, and find an atom pushing our piston out again, giving us work; but equally likely, for a truly random message, we have to do work on the atom. The net result is zero work to power our machine.

Clearly, Bennett's tape machine seems to do the opposite to our reset process. He uses a message tape to extract work, ending up with a random tape: we took a random tape and did work on it, to end up with a tape of standard zeroes. This inverse relationship is reflected in the definition of information within Bennett's framework. Suppose we have a tape with N bits. We *define* the information, I, in the tape by the formula:

$$\textit{Fuel value of tape} = (N-I).kT \log 2. \qquad (5.12)$$

From this we see that a tape giving us a full fuel-load — that is, $kT\log2$ per bit — carries zero information. This is what we would expect since such a tape must have completely predictable contents. There is a nice physical symmetry between these two approaches. If we run a message tape through the machine, we will be able to extract a certain energy E from it: this energy E will be precisely what we need to reset the newly randomized tape to its original form. It is, of course, up to you which picture you prefer to adopt when thinking about these things. I opt for the erasure picture partly because I do not like having to subtract from N all the time to get my information!

You might like to contemplate some problems on Dr. Bennett's machine.

Problem 5.1: Suppose we have two tapes: an N-bit random tape, and an exact copy. It can be shown that the fuel value of the two tapes combined is $NkT\log 2$. See if you can design a machine that will be able to extract this amount of energy from the two tapes. (Hint: you have to expand one tape "relative" to the other.)

Problem 5.2: We have a tape in which three bits are repeated in succession, say 110110110110... For a $3N$-bit tape, what is the fuel value? How do you get it out?

5.1.1: Maxwell's Demon and the Thermodynamics of Measurement

Those of you who wish to take your study of the physics of information further could do no better than check out many of the references to a nineteenth century paradox discovered by the great Scottish physicist James Clerk Maxwell. *Maxwell's Demon*, as it is known, resulted in a controversy that raged among physicists for a century, and the matter has only recently been resolved. In fact, it was contemplation of Maxwell's demon that partly led workers such as Charles Bennett and Rolf Landauer to their conclusions about reversible computing, the energy of computation, and clarified the link between information and entropy. Importantly, such research has also shed light on the role of *measurement* in all this. I will not go into the matter in great detail here, but supply you with enough tidbits to at least arouse your interest. A full discussion of the demon and of the attempts to understand it can be found in *Maxwell's Demon: Entropy, Information, Computing*, by H.S. Leff and A.F. Rex (Adam Hilger, 1990).

With Maxwell, we will imagine that we have a small demon sitting on a partitioned box, each half of which is filled by a gas of molecules with a random distribution of positions and velocities (Fig. 5.7):

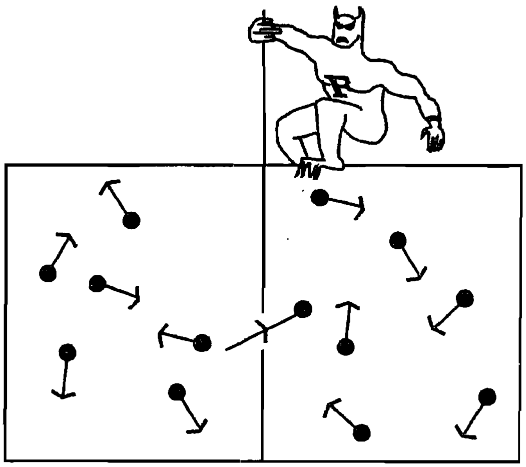


Fig. 5.7 Maxwell's Demon at Work

The demon has a very simple task. Set into the partition is a flap, which he can open and shut at will. He looks in one half of the box (say, the left) and waits until he sees a fast-moving molecule approaching the flap. When he does, he opens the flap momentarily, letting the molecule through into the right side, and then shuts the flap again. Similarly, if the demon sees a slow-moving molecule approaching from the right side of the flap, he lets that through into the side the fast one came from. After a period of such activity, our little friend will have separated the fast- and slow-moving molecules into the two compartments. In other words, he will have separated the hot from the cold, and hence created a temperature difference between the two sides of the box. This means that the entropy of the system has decreased, in clear violation of the Second Law!

This seeming paradox, as I have said, caused tremendous controversy among physicists. The Second Law of Thermodynamics is a well-established principle in physics, and if Maxwell's demon appears to be able to violate it, there is probably something fishy about him. Since Maxwell came up with his idea in 1867, many people have tried to spot the flaw in his argument. Somehow, somewhere, in the process of looking for molecules of a given type and letting them through the flap, there had to be some entropy generated.

Until recently, it was generally accepted that this entropy arose as a result of the demon's *measurement* of the position of the molecules. This did not seem unreasonable. For example, one way in which the demon could detect fast-moving molecules would be to shine a demonic torch at them; but such a process would involve dispersing at least one photon, which would cost energy. More generally, before looking at a particular molecule, the demon could not know whether it was moving left or right. Upon observing it, however this was done, his uncertainty, and hence entropy, would have reduced by half, surely accompanied by the corresponding generation of entropy in the environment.

In fact, and surprisingly, Bennett has shown that Maxwell's demon can actually make its measurements with zero energy expenditure, providing it follows certain rules for recording and erasing whatever information it obtains. The demon must be in a standard state of some kind before measurement, which we will call S: this is the state of uncertainty. After it measures the direction of motion of a molecule, it enters one of two other states — say L for "left-moving", or R for "right-moving". It overwrites the S with whichever is appropriate. Bennett has demonstrated that this procedure can be performed for no energy cost. The cost comes in the next step, which is the *erasure* of the L or R to reset the demon in the S state in preparation for the next measurement. This realization, that it is the erasure of information, and not measurement, that is the source of entropy generation in the computational process, was a major breakthrough in the study of reversible computation.

5.1.2: Energy and Shannon's Theorem

Before leaving physics and information, I would like to return to something we studied in the previous chapter, namely, the limits on sending information down a channel. It will come as no surprise to you that we can revisit Shannon's Theorem with our physical tools too! Let us combine our study of the physics of information with our earlier work on errors. An interesting question is: How does the occurrence of an error in a message affect its information content? Let's start off with a message with all its M bits perfectly known, containing information N, and suppose that we want to send it somewhere. We're going to send it through a noisy channel: the effect of this is that, in transit, each bit of the message has a probability q of coming through wrong. Let us ask a familiar question: what is the minimum number of bits we have to send to get the information in the M bits across? We will have to code up the message, and in keeping with our earlier look at this question, we'll say the coded message has length M_C. This is the number of bits we actually send. Now we have said that to clear the tape, assuming we know nothing about its contents, we need to expend the following amount of free energy:

$$M_C kT \log 2. \tag{5.13}$$

However, some of this energy is taken up in clearing errors. On average, using our earlier derivations, this amount will be:

$$M_C kT \log 2[-q\log_2 q-(1-q)\log_2(1-q)]=[1-f(q)]M_C kT \log 2. \tag{5.14}$$

This energy we consider to be wasted. This leaves us with the free energy:

$$M_C kT \log 2 - [1-f(q)]M_C kT \log 2 = f(q)M_C kT \log 2 \tag{5.15}$$

to expend in clearing the message. By conservation of energy, then, and using our relationship between free energy and information, the greatest amount of information I can send through this channel will be:

$$M_C\ [q\log_2(1/q) + (1-q)\log_2(1/1-q)] \tag{5.16}$$

You can see how this kind of physical argument now leads us on to Shannon's result.

5.2: Reversible Computation and the Thermodynamics of Computing

It has always been assumed that any computational step required energy[2]. The first guess, and one that was a common belief for years, was that there was a minimum amount of energy required for each logical step taken by a machine. From what we have looked at so far, you should be able to appreciate the argument. The idea is that every logical state of a device must correspond to some physical state of the device, and whenever the device had to choose between 0 and 1 for its output — such as a transistor in an AND gate — there would be a compression of the available phase-space of the object from two

[2]Detailed accounts of the history of this subject can be found in the papers "Zig-zag Path to Understanding", R. Landauer, Proceedings of the Workshop on Physics and Computation Physcomp '94, and "Notes on the History of Reversible Computation", C.H. Bennett, IBM J. Res. Dev. 32(1), pp. 16-23 (1988). [Editors]

options to one, halving the phase-space volume. Therefore, the argument went, a minimum free energy of $kT\log 2$ would be required per logical step[3]. There have been other suggestions. One focused on the reliability of the computational step. The probability of an error, say q, was involved and the minimum energy was supposed to be $kT\log q$. However, recently this question has been straightened out. The energy required per step is less than $kT\log q$, less than $kT\log 2$, in fact less than any other number you might want to set – provided you carry out the computation carefully and slowly enough. Ideally, the computation can actually be done with *no* minimal loss of energy. Perhaps a good analogy is with friction. In practice, there is always friction, and if you take a look at a typical real-world engine you will see heat energy dissipated all over the place as various moving parts rub against one another. This loss of energy is ordinarily large. However, physicists are very fond of studying certain types of idealized engines, so-called Carnot heat engines in which heat energy is converted into work and back again, for which it is possible to calculate a certain maximum efficiency of operation. Such engines operate over a *reversible* closed cycle: that is, they start off in a particular state and, after one cycle of operation, return to it. The Second Law ensures that this cannot be done for zero energy cost but it is theoretically possible to operate such machines in such a way as to achieve the maximum efficiency, making the losses due to friction, for example, as small as possible. Unfortunately, they have to be run infinitesimally slowly to do this! You might, for example, want to drain heat from the engine into a surrounding reservoir to keep everything at thermal equilibrium, but if you operate the machine too quickly you will not be able to do this smoothly and will lose heat to parts of the engine that will simply dissipate it. But the point is that, in principle, such engines could be made, and physicists have learned much about thermodynamics from studying them. The crucial requirement is reversibility. Now it turns out that a similar idea works in computers. If your computer is reversible, and I'll say what I mean by that in a moment, then the energy loss could be made as small as you want, provided you work with care and slowly – as a rule, infinitesimally slowly. Just as with Carnot's engines, if you work too fast, you will dissipate energy. Now you can see why I think of this as an academic subject. You might even think the question is a bit dopey – after all, as I've said, modern transistors dissipate something like 10^8 kT per switch – but as with our discussion of the limits of what is computable, such questions are of interest. When we come to design the Ultimate Computers of the far future, which might have "transistors" that are

[3] This is actually a lower limit far beneath anything practically realizable at present. Conventional transistors dissipate on the order of 10^8 kT per step! [RPF]

atom-sized, we will want to know how the fundamental physical laws will limit us. When you get down to that sort of scale, you really have to ask about the energies involved in computation, and the answer is that there is no reason why you shouldn't operate below *kT*. We shall look later at problems of more immediacy, such as how to reduce the energy dissipation of modern computers, involving present-day transistors.

5.2.1: Reversible Computers

Let me return to the matter of "reversible computing". Consider the following special kind of computation, which we draw as a black box with a set of input and output lines (Fig. 5.8):

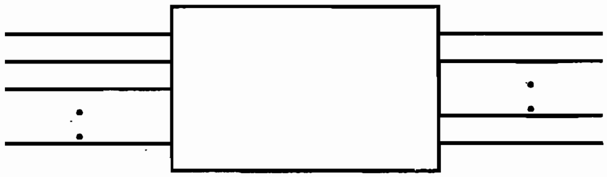


Fig. 5.8 A Reversible Computation

Suppose that for every input line there is one, and only one, output, and that this is determined by the input. (In the most trivial case, the signals simply propagate through the box unchanged.) In such circumstances, the output carries no more information than the input — if we know the input, we can calculate the output and, moreover, the computation is "reversible". This is in sharp contrast to a conventional logic gate, such as an AND (Fig. 5.9):

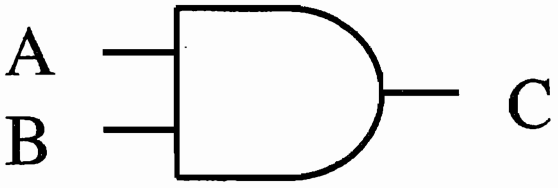


Fig. 5.9 The AND Gate

In this case we have two lines going in but only one coming out. If the output is found to be zero, then any one of three possible states could have led to it. I have irretrievably lost information about the input so the AND gate is irreversible. So too, is the OR gate (but not the NOT!). In other words, the phase space of the inputs has shrunk to that of the output, with an unavoidable decrease in entropy. This must be compensated by heat generation somewhere. The mistake everyone was making about energy dissipation in computers was based on the assumption that logical steps were necessarily like AND and OR — irreversible. What Bennett and others showed was that this is not necessarily the case. The fact that there is no gain in information in our abstract "computation" above is the first clue that maybe there's no loss of entropy involved in a reversible computation. This is actually correct: reversible computers are rather like Carnot engines, where the reversible ones are the most efficient. It will turn out that the only entropy loss resulting from operating our abstract machine comes in resetting it for its next operation.

We can consider a "higher" kind of computer which is reversible in a more direct sense: it gives as its output the actual result of a computation plus the original input. That is, it appends the input data to the output data printed on its tape (say). This is the most direct way of making a computation reversible. We will later show that, in principle, such a calculation can be performed for zero energy cost. The only cost is incurred in resetting the machine to restart, and the nice thing is that this does not depend on the complexity of the computation itself but only on the number of bits in the answer. You might have billions of components whirring away in the machine, but if the answer you get out is just one bit, then *kT*log2 is all the energy you need to run things.

We actually studied some reversible gates earlier in the course. NOT is one, as I've said. A more complicated example we looked at was Fredkin's CONTROLLED CONTROLLED NOT gate (Fig. 5.10):

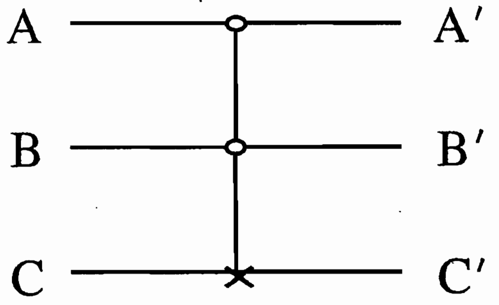


Fig. 5.10 The CCN Gate

in which the lines *A* and *B* act as control lines, leaving *C* as it is unless both are one, in which case *C* becomes NOT *C*. This is reversible in the sense that we can regain our input data by running the output through another CCN gate (see section 2.3).

I would now like to take a look in more detail at some reversible computations and demonstrate the absence of a minimum energy requirement. I'll start with a computation that you might not ordinarily think of as a computation: the act of copying (recall our discussion of Turing copying machines, §3.5). This seems like a dumb sort of computation, as you're not getting anywhere, but it is a useful introduction to some of the ideas underlying issues of energy dissipation. It's not at all obvious that you can copy information down from one place to another without expending at least some energy, even in principle. Having said this, it is easy to suggest why it might not cost any energy. We can consider a set of data and its copy as two messages on tape, both identical. Either we know what the original message is, or we don't. In the first case, no free energy is expended in clearing the tape, and none need be for the copy tape: we just turn it over when necessary, as we discussed previously. In the second case, clearing the tape will cost free energy, but not for the copy: knowing what the first tape says, we can use this information to clear the copy by turning bits over again. Simply, there is no more information in the (data plus copy) set than is in just the single data set. Clearing the system should not, therefore, require more free energy in the first case than the second. This is a common type of argument in the reversible computing world.

5.2.2: The Copy Computation

Let us make these ideas a little more concrete. In a moment, I will examine a copying machine found in Nature, namely the RNA molecule found in living cells. But first, I will take a look at two rather artificial examples of copying machines. Our discussion follows Bennett.

We start with a very general copy process. We will have an original object, which we'll call the model, which can somehow hold a zero or one. It's some kind of bistable physical device. We want another object, which we'll call the copier, which can also hold a zero or one. An example of a bistable device would be one which could be modeled by the following potential well (Fig. 5.11):

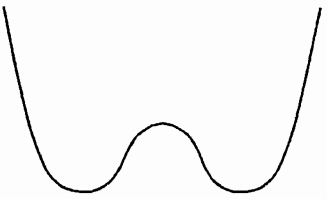


Fig. 5.11 A Potential Well

I will give one possible physical realization of this shortly. What this rather abstract diagram means is that some part of the device, which we will represent by a dot, can be in either of two stable states – here, in the left or the right trough, meaning one or zero, say. The curve displays the potential energy of the dot according to its position in the device. The troughs are the minima of this energy, and are favored by the dot: they are of equal depth, and are hence equally likely to be occupied at the outset. A useful way to think of this operation is to have the dot as a ball, and the curve an actual shape constraining it. Putting energy into the ball makes it move up and down the sides of its trough; enough energy and the ball will go over the hill and into the next trough – equivalent to our model changing its bit-state. The height of the hill, the amount of energy needed for the transition to occur, is called the barrier potential. In actual operation, we would want the typical thermal fluctuations of whatever it is the dot represents to be much less than this, to keep the device stable. Another way of visualizing this is to imagine the dot to be in a box separated into two halves by a partition. The barrier potential will be the energy required to get the dot from one half into the other.

We suppose both model and copier to be modeled by such a potential, and the model to be in some state. This can be random – we need not know what it is, but for sake of illustration let's say it is as shown in Figure 5.12 (where we have used an X for the model's dot):

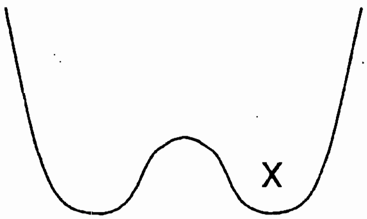


Fig. 5.12 Initial State of the Model

How does the copier start out? It must be in some standard state. It cannot be in a random state, because copying will involve getting it into a definite state, and to do this we must do work (compressing, if we use the box and partition analogy). Alternatively, you can use phase space considerations, comparing the number of possible model-copier options before copying (four, if the model is randomly set) and after (just two): this would be a logically irreversible step. Let's say the copier starts out in the state opposite to the model (Fig. 5.13):

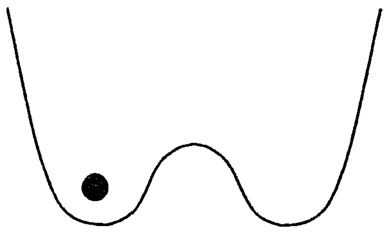


Fig. 5.13 Initial State of the Copier

Clearly, copying is going to involve somehow getting the dot from one trough to the other. To do this, we need to be able to manipulate the potential curve; we have to make the other trough energetically more favorable to the dot. We shall assume that there are two parameters associated with the copier that we can adjust: the barrier height, and the relative depths of the troughs. Furthermore, we assume that the depths of the troughs can be altered by some force of interaction between the copier and the model. (Don't worry if this is all horribly confusing and abstract! All will become clear.) We'll call this a "tilt" force, since it tilts the graph. We will combine these two operations to move the copier dot, but we will combine them in such a way — and this is important — that there will always be a unique minimum accessible to the dot at all times.

What we do is this. We start with the model some way away from the copier. Even at a distance it will exert a slight tilt force on the copier. We take this force to have the consequence of increasing the depth of whichever trough of the copier corresponds to that occupied in the model. The copier potential will hence be slightly distorted at the outset, as shown overleaf:

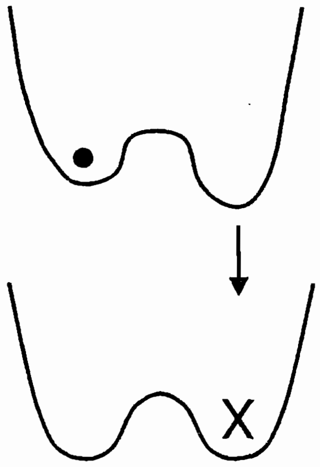


Fig. 5.14 Initial Copier Distortion

The first step in the copy process involves gently lowering the copier's potential barrier. This removes the obstacle to the dot switching positions: it can now wander over to the other bit state. What will make it do this? This is where the "tilt" from the model comes in. In step two, we slowly bring the model up closer to the copier, and in the process its tilt force increases. This gradually distorts the copier's potential even more, lowering the energy of the appropriate trough as shown in Figure 5.15:

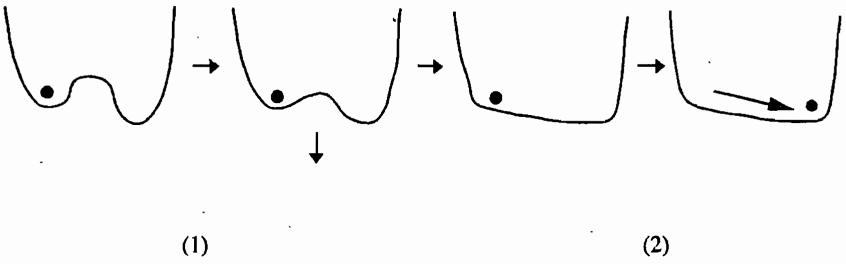


Fig. 5.15 Lowering the Potential Barrier and Tilting

The dot now slides smoothly down the potential curve, occupying the new, energetically more favorable trough. In step three, we replace the potential barrier to secure the dot in its new position, and finally, step four, we take the model away, restoring the copier's potential to its normal state (Fig. 5.16):

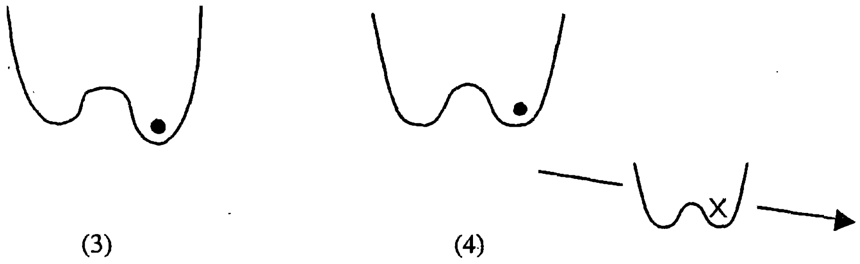


Fig. 5.16 Final State of the System

That is the basic idea of this copy machine. It's possible to play around with it further. For example, for appropriate physical systems, we can envisage bringing the model up to the copier in step one in such a way as that the tilt force lowers the state the dot is already in so that the dot is held steady while we lower the potential barrier, if this is a concern. The model is then moved over to the other side to provide the new tilt. This is one variation, but it does not significantly alter the basic idea.

The crucial thing about this process is that it needs to be carried out slowly and carefully. There are no jumps or sudden changes. The easiest way to get the dot from one trough to the other would be to bring the model up rapidly to bias the troughs in the desired way, then to rip away the potential barrier. The dot would then slosh over into its new trough, but the whole process, while nice and quick, would invariably involve dissipation in a real system. However, if the procedure is graceful enough, the lowering of the barrier, the tilting of the trough and the copying can be done for nothing. This is basically because the physical quantities that contribute to the energy dissipation — such as the kinetic energy of the dot moving to its new state, the work done in raising and lowering the barrier — are negligible under such circumstances. You should be able to see, incidentally, that this procedure will work even if we don't know what state the model is in.

When Bennett discovered all this, no one knew it could be done, although much of the preliminary groundwork had been carried out by his IBM colleague, Landauer, as far back as 1961. There was a lot of prejudice around that had to be argued against. I see nothing wrong with his arguments. I was asked by Carver Mead at CalTech to look into the energy consumption of computers, so I looked at all this stuff and gradually concluded that there was no minimum energy. This was something of a surprise to me! Bennett's result was four years old by then but there were still people fighting over it. Also it's nice to work this sort of thing out for yourself: as I said in Chapter One – OK, you're not the first, but at least you understand it!

5.2.3: A Physical Implementation

Let me return to the preceding example and give you something that is essentially a physical realization of it. It is also fun to think about! We need some kind of bistable physical device, and here it is: two compass needles – just two magnetic dipoles on pivots. One end is North and the other South, and as we all know North attracts South and vice versa; otherwise we have repulsion. Now suppose that both the model and the copier are made up of such a pair. To make the analysis easier, we insist that the each member of a pair is linked to the other, in such a way that both members must point in the same direction. This means that we can analyze each system in terms of just one variable, the angle ϕ the needles make with the horizontal. So we have the allowed and disallowed situations shown below:

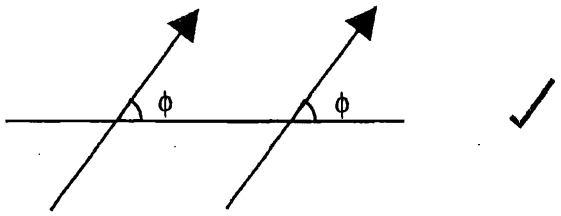


Fig. 5.17(a) Allowed Angular Configuration

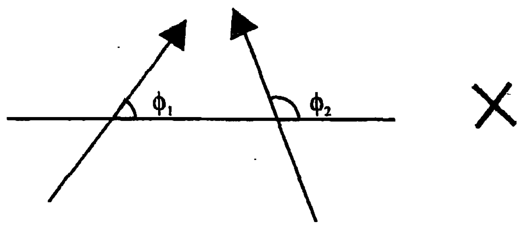


Fig. 5.17(b) Disallowed Angular Configuration

The disallowed case would, in any case, clearly be unstable. Now, not all alignments of the needles within a pair have the same potential energy. This is obvious by comparing the states shown in Figure 5.18:

(horizontal) $S \longrightarrow N \quad S \longrightarrow N$ with (vertical) $\begin{array}{cc} N & N \\ \uparrow & \uparrow \\ S & S \end{array}$

Fig. 5.18 Stable and Unstable States

The first is evidently quite stable, with the tip of one needle attracting the base of the other. The second, with both arrows vertical, is quite unstable: the North poles will repel, and the needles will seek to occupy the first state or its mirror image. We can actually calculate the potential energy for a state with angle ϕ. It is approximately (close enough for us) given by:

$$\textit{Potential energy} \approx \sin^2 \phi \qquad (5.17)$$

This potential energy function looks like the graph of Figure 5.19:

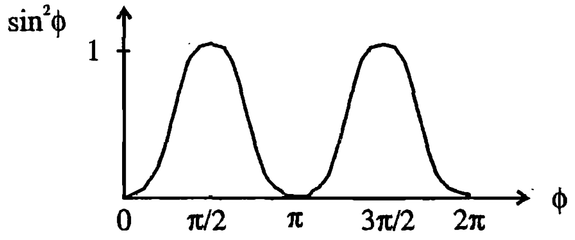


Fig. 5.19 Potential Energy as a function of ϕ

Note how similar this is to our abstract potential well. The minima are at $\phi = 0$ and $\phi = \pi$, corresponding to the stable "horizontal" states, whilst the maxima correspond to the vertical states at $\pi/2$ and $3\pi/2$. (Remember that the graph wraps around at $\phi = 0$ and 2π.) The system is clearly bistable and we can see that once the needles are in one of the two minima, energy would have to be expended to push them to the other.

To manipulate the barrier in this case, we introduce a vertical magnetic field B. It can be shown that this adds a term:

$$-B \sin \phi \tag{5.18}$$

to the potential energy. As we increase B, the effect is to lower the barrier between the 0 and π states as shown in Figure 5.20:

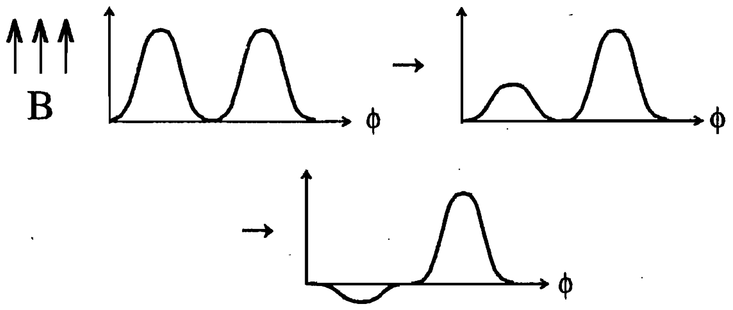


Fig. 5.20 Barrier Manipulation in the Dipole Copier

(You can play with numbers to gauge the exact effect of this.) The tilt force, as before, results from bringing the model closer to the copier; this time, we can see what it is about the model that causes this force — it is the magnetic field from the data bit. The force is perpendicular to B, and in the direction of the needles in the model. If we call it b, then it contributes:

$$-b \cos \phi \tag{5.19}$$

to the potential energy. This clearly removes the symmetry about $\pi/2$ and $3\pi/2$ and represents a tilting. We can now see how the copying process works. We start with the copier in a standard state, which we take to be the $\phi = 0$ state ($\rightarrow\rightarrow$). We gently turn up the field B — or alternatively slowly move the copier from a region of weak B to one of high B — until the barrier is removed. At this stage, the dipole is vertical (Fig. 5.21):

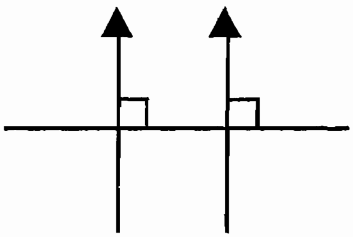


Fig. 5.21 Initial (Unstable) Copier State

Now we bring in the model. This has already been slightly perturbing the copier pair, but not enough to have a noticeable effect so far. Now, as it gets closer, its field biases the copier needles to flip over — but not suddenly! — into a new state. (This is if a new state is appropriate: if the standard state and the model state coincide, the needles will simply return to their original position.) The model is removed, the copier taken out of the field B to restore the barrier, and the copying is finished.

Once again, you can check that this copying method will work if we do not know what the model state is. It is not difficult to see that, if performed slowly, it will cost no energy — no current, no nothing. My previous discussion was to show you the principles; this specific example is probably easier to understand.

5.2.4: A Living Computer

The foregoing example of two dipoles has a certain physical basis, but is undeniably artificial. However, here's a copying process that really is found in

Nature and is one that involves thermodynamical, rather than mechanical, forces[4]. It occurs as one of the steps in the synthesis of proteins in a living cell. Now you probably know what proteins are — long, twisted molecular chains of amino acids (such as tryptophan or alanine) — and you may know how central they are both to the structure and functioning of living things. However, a proper understanding of the complex business that is their manufacture would require an understanding of biochemistry lying way beyond this course! I can't make up for that here, so I'll just try to give you enough background to let you see how the copying "machine" I have in mind behaves.

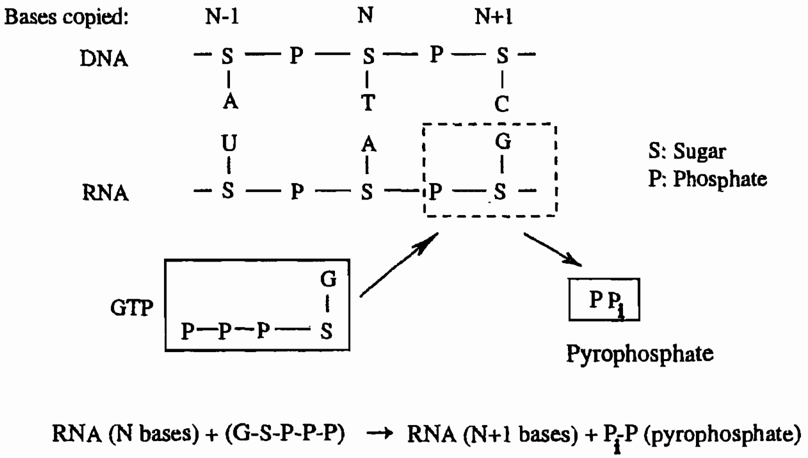
A living creature typically contains a huge number of different types of protein, each uniquely defined by some combination of specific amino acids. If the cell is to manufacture these molecules, then clearly a set of "design rules" for each protein-type must be available somewhere. This information actually resides in the DNA (Deoxyribonucleic Acid) molecule, the famous "double-helix" structure which resides in the cell nucleus. DNA comprises a double chain, each strand of which is made up of alternating phosphate and pentose sugar groups. To each sugar group is attached one of four bases, A (adenine), T (thymine), C (cytosine) and G (guanine) (a base-sugar-phosphate group is called a nucleotide). It is a certain sequence of bases that provides the code for protein synthesis.

We can break down the synthesis of proteins into two stages. The first stage, and it is only this which interests us, requires the formation of another, linear, strand of sugar phosphates with bases attached, called messenger RNA (or m-RNA). The code on the DNA is copied onto the RNA strand base by base (according to a strict matching rule, which I shall come on to), and the m-RNA, once completed, leaves the nucleus and travels elsewhere to assist in the making of the protein. The machine that does the copying is an enzyme called RNA polymerase. What happens is this. The DNA and enzyme are floating around in a crazy biological soup which contains, among other things, lots of triphosphates — such as ATP (adenosine triphosphate), CTP, GTP and UTP (U is another base, Uracil). These are essentially nucleotides with two extra phosphates attached.The polymerase attaches itself to whichever part of one of the DNA strands it is to duplicate and then moves along it, building its RNA copy base by base by reacting the growing RNA strand with one of the four nucleotides present in the soup. (A crucial proviso here is that RNA is built out of the four bases A, G,

[4] For a discussion of this topic in the literature, see C. Bennett, Int. J. Theor. Phys 21, pp. 905-940 [1982].[RPF]

C and U (not T), and that the RNA strand must be complementary to that on the DNA; the complementarity relationships are that As on the DNA must match with Us on the RNA, Ts with As, Cs with Gs and Gs with Cs). The nucleotides are provided in the triphosphate form, and during the addition process two of the phosphates are released back into the soup, still bound together (as a pyrophosphate). The nucleotide chosen must be the correct one, that is, complementary to the base on the DNA strand that is being copied. For example: suppose the enzyme, traveling along the DNA strand, hits a C base. Now at this particular stage of its operation a bonding between the polymerase and a GTP molecule from the soup happens to be more energetically favorable than one between it and UTP or ATP: complementarity is actually enforced by energy considerations. Preferentially, then, it will pick up a GTP molecule. It releases a pyrophosphate back out into the soup, moves along the DNA and looks for the next complementary nucleotide.

Schematically, we have the following picture (Fig. 5.22):



Bases copied:
N-1
N
N+1
DNA
RNA
S: Sugar
P: Phosphate
GTP
PPi
Pyrophosphate
RNA (N bases) + (G-S-P-P-P) → RNA (N+1 bases) + Pi-P (pyrophosphate)

Fig. 5.22 Formation of m-RNA

Now the role of enzymes in biochemical processes is as catalysts: they influence the rate at which reactions occur, but not the direction in which they proceed. Chemical reactions are reversible, and it would be just as possible for the polymerase reaction to go the other way — that is, for the enzyme to undo the m-RNA chain it is working on. In such an event, it would extract a pyrophosphate from the surrounding soup, attach it to a base on the m-RNA, and then release the whole lot back into the environment as one of our triphosphates. The enzyme could just move along the wrong way, eat a G, move along, eat a C, move along,.., undoing everything[5]. Which way the reaction goes depends on the relative concentrations of pyrophosphates and triphosphates in the soup. If there is a lot of ATP, GTP, and so on, but not much free pyrophosphate, then the rate at which the enzyme can run the reaction backwards is lowered, because it can't find much pyrophosphate with which to pull off the m-RNA nucleotides. On the other hand, if there is an excess of free pyrophosphates over triphosphates, the reaction will tend to run the wrong way, and we'll be uncopying and ruining our copy.

We can actually interpret these relative concentrations in terms of the number of possible states available to our system at any given computational point. If there are plenty of triphosphates around, then there are plenty of forward-moving, and comparatively few backward-moving, states available: the RNA polymerase will tend to enter the former state, in the process lowering its entropy. The difference in free energies, measured by the differing concentrations, determines the way it goes. If we get the concentrations just right, the copier will oscillate forever, and we will never get around to making copies. In an actual cell, the pyrophosphate concentration is kept low by hydrolysis, ensuring that only the copying process occurs, not its inverse. The whole RNA polymerase system is not particularly efficient as far as energy use goes: it dissipates about $100kT$ per bit. Less could be wasted if the enzyme moved a little more slowly (and of course, the reaction rate does vary with concentration gradient), but there has to be a certain speed for the sake of life! Still, $100kT$ per bit is considerably more efficient than the 10^8kT thrown away by a typical transistor!

To reiterate: The lesson of this section is that there is no absolute minimum amount of energy required to copy. There *is* a limit, however, if you want to copy at a certain speed.

[5]Bennett has nicely christened machines like this "Brownian computers" to capture the manner in which their behavior is essentially random but in which they nevertheless progress due to some weak direction of drift imposed on their operation. [RPF]

5.3: Computation: Energy Cost versus Speed

The question of speed is important and I would like to write down a formula for the amount of free energy it takes to run a computation in a finite time. This at least makes our discussion a bit more practical. There is little room for reversible computing in the computer world at the moment, although one can foresee applications that are a little more immediately useful than getting from $10^8 kT$ to under kT. (You can actually get to 2 or 3 kT irreversibly, but you can't get under this.) For example, we can look at the problem of errors arising in parallel processing architectures where we might have thousands of processors working together. The question of error correction through coding in such a situation has arisen and is unsolved. It occurs to me that maybe the devices in the machine could all be made reversible, and then we could notice the errors as we go. What would be the cost of such reversible devices? Maybe these things will find application soon. That would make this discussion more practical to you and since computing is engineering you might value this! In any case, I shouldn't make any more apologies for my wild academic interest in the far future.

An example we gave of reversible computing was that of the chemical process of copying DNA. This involved a machine (if you like) that progressed in fits and starts, going forward a bit, then backwards, but more one than the other because of some driving force, and so ended up doing some computation (in this case, copying). We can take this as a model for more general considerations and will use this "Brownian" concept to derive a formula for the energy dissipation in such processes. This will not be a general formula for energy dissipation during computation but it should show you how we go about calculating these things. However, we will precede this discussion by first giving the general formula[6], and then what follows can be viewed as illustration.

Let us suppose we have a reversible computer. Ordinarily, the free energy expended in running it reversibly will be zero, when the process is infinitesimally slow, but let us suppose that we are actually driving it forward at a rate r. In other words, at any given stage, it is r times more likely to make a forward calculational step than a backwards one. Then, the general result is that minimum energy that must be expended per computational step in the process is:

[6] This rule is pretty general, but there will be exceptions, requiring slight corrections. We will [illegible] "ballistic" computer in §5.5. [RPF]

$$kT \log r. \tag{5.20}$$

Note that the smaller r is, the lower the energy.

Let us illustrate this rule by looking at a Brownian-type computer. Imagine we have a system, or device, in a particular state, which has a particular energy associated with it. It can go forwards or backwards into a new state, each transition corresponding either to doing a computation (forward) or undoing it (backward). We can model this situation using the energy level diagram of Figure 5.23:

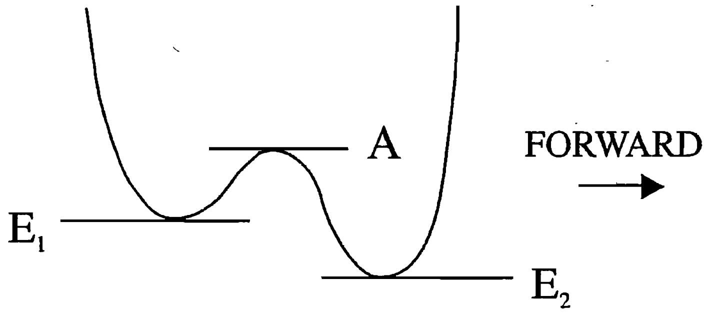


Fig. 5.23 The General Transition

We assume our computer to be sitting in one of the two states, with energy E_I or E_2. These energies will not generally be equal. Now our device can go from E_I to E_2, a forward step – the idea is that the energies are lower in the direction of computation – or from E_2 to E_I, a backward step. The energies of the two states might be equal, but one of them could be effectively lowered by the imposition of an external driving force. We have introduced into this diagram the "activation energy" A, which is the energy that must be supplied to the system to cause a transition of any kind. We will focus on the effects of thermal fluctuations which will, quite randomly, cause the computer to move between states, whenever the energy of these fluctuations exceeds A. Such fluctuations can make the device go either way, and we can calculate the rate at which it goes in either direction. These will not be equal. Roughly, the chance of the system going into the state with energy E_i is the chance that by accident it acquires enough energy to get past the barrier (that is, A) and into E_i. Clearly, the energy needed to get from E_I to E_2, a forward step, is $(A\text{-}E_I)$, while to get from E_2 to E_I it is $(A\text{-}E_2)$. It is a standard result in statistical mechanics that the probability of a transition from one state to another differing in (positive) energy δE is:

$$C \exp(-\delta E/kT), \tag{5.21}$$

where C is a factor that carries information about the thermal fluctuations in the environment. This can be calculated through a phase-space (entropy-type) analysis, examining the probabilities of ensemble transitions between states. However, we are interested in the transition rates between states and this is describable by a similar formula. We simply have to insert another factor, say X, giving us:

$$\textit{forward rate} = CX \exp[-(A-E_1)/kT] \tag{5.22}$$

and

$$\textit{backward rate} = CX \exp[-(A-E_2)/kT]. \tag{5.23}$$

The factor X depends on a variety of molecular properties of the particular substance (the mean free path, the speed, and so on), but the property that interests us is that it does not depend on E (consider the transition rates for the case $E_1 = E_2$). We can therefore write for the ratio of the forward to backward rates:

$$\exp[(E_1-E_2)/kT]. \tag{5.24}$$

This depends only on the energy difference between successive states. This gives us some insight into the rate at which our computation (= reaction) proceeds, and the energy difference between each step required to drive it. The bigger the energy difference E_1 - E_2, the quicker the machine hops from E_1 to E_2, and the faster the computation.

We can tie this result into our earlier general formula by setting the above rate equal to r. We then have, for the energy expended per step:

$$kT \log r = E_1-E_2 \tag{5.25}$$

which makes sense.

Let me give you one more illustration of driving a computer in a particular direction. This time we will look at computational states that do not differ in their energy, but in their availability. That is, our computer selects which state of a certain kind to go into next, not on the basis of the energy of the state, but on the number of equivalent states of that kind available for it to go into. We have an example of such a situation in our DNA copier. A calculational step there involved the RNA enzyme attaching bases to the RNA chain and liberating pyrophosphates into the surroundings. The inverse step involved taking up phosphates from the solution and breaking off bases. Each step is energetically equivalent but one can be favored over the other, depending on the relative concentrations of chemicals in the soup. Suppose there is a dearth of phosphates but a wealth of bases available. Then, the number of (forward) states of the system in which a base is attached to the RNA strand and a phosphate is released — and we consider all such states equivalent — exceeds the number of states in which a phosphate is grabbed and a base released (again, all such states we take to be the same). So we can envisage a computer designed so that it proceeds by diffusion, in the sense that it is more likely to move into a state with greater, rather than lower, availability. Schematically, we have the situation shown in Figure 5.24:

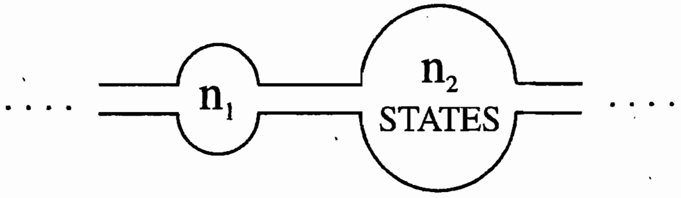


Fig. 5.24 The Availability of States

where n_i is the number of available states. It is possible to show in this situation (although it takes a litle thought) that the ratio of the forward rate to the backward rate is:

$$r = n_2/n_1. \tag{5.26}$$

If you recall, we defined the entropy of a configuration of a system to be:

$$S \approx k \log W \tag{5.27}$$

where W is the probability of finding the system in that configuration. Hence we may write:

$$kT \log r = kT (\log n_2 - \log n_1) = (S_2 - S_1)T \tag{5.28}$$

(with various constant factors canceling to leave the equality). In other words, for this process the energy loss per step is equal to the entropy generated in that step, up to the usual temperature factor. Again, this makes sense.

So we can see that our general formula reduces to the specific formulae we have obtained in these instances. An interesting question that arises is: in a real world situation, can we minimize the energy taken per computational step? We know that if we have an effectively reversible computer, the chances of forward and backward movement are equal, and we have no energy loss. The price we pay for this is that a computation will take an infinite time. We will never know when we're finished. So as we've said, to get it going we want to give things a tug, lower the energies of successive steps, make them more available, or whatever. Let us suppose that we have the forward rate, f, just a little bigger than the backward rate, b, so the computation just goes. We write:

$$f = b + \Theta \tag{5.29}$$

where Θ is small. Our general formula now gives:

$$\textit{energy per step} = kT \log [1+(\Theta/b)] \approx kT\Theta/b = kT(f-b)/b \tag{5.30}$$

for small Θ. We can provide a nice physical interpretation of this expression, although at the cost of mathematical inaccuracy. We replace the formula above by one that is nearly equal to it:

$$\textit{energy per step} = kT \frac{(f-b)}{(f+b)/2}. \tag{5.31}$$

This differs from the original formula by terms of order Θ^2. Now the numerator of this fraction is the speed at which we go forward and do the calculation. It is a bit like a velocity, in that it represents the rate at which the computer drifts through its calculation, measured in steps per second. The denominator is the average rate of transition — it is a measure of the degree to which our computer is oscillating back and forth. We can interpret this roughly as the fastest speed at which you could possibly go, backwards or forwards, which would be the speed found if the computer underwent a series of steps in one direction with no reverses: it is the greatest possible drift. So we can write, approximately:

$$\textit{energy loss per step} = kT\,\frac{v_{drift}}{v_{max}}. \tag{5.32}$$

Alternatively, we can emphasize time as our variable and write:

$$\textit{energy loss/step} = kT\frac{\textit{minimum time taken/step}}{\textit{time/step actually taken}} \tag{5.33}$$

Let us now take a look at more general issues in reversible computing.

5.4: The General Reversible Computer

We have repeatedly stated that, if a computation is to be reversible, then we have to store a lot of information that we would ordinarily lose or throw away in order that we can subsequently use it to undo something. The logic gates of such a machine give us not just the answer to the logical calculation we want, but a whole lot of extra bits. A simple illustration of this for a realistic gate is a simple adder built out of reversible gates. In §2.3 I set as a problem for you, the construction of a full three-bit adder from reversible gates — specifically, using CN and CCN gates (or alternatively, just CCN gates, out of which all others can be built). An easier example, the simple two-bit adder, is built as follows:

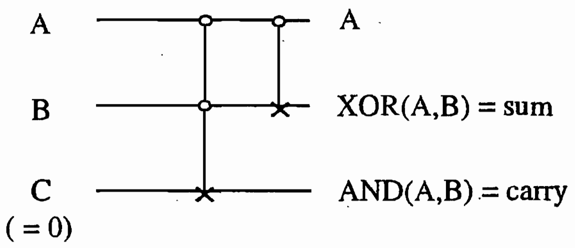


Fig. 5.25 A Two-bit Adder From Reversible Gates

The C-input is kept set at zero (the full three-bit adder requires the addition of a fourth input line, kept set at zero). As well as the sum and carry of A+B, we find this gate feeds the A-line through. We can see that this bit is necessary if we are to be able to reconstruct the input (A,B) from the output. If you look at the three-bit adder, you will find two spare bits at the output. Generally, then, we will always need a certain amount of junk to remind us of the history of the logical operation. We can summarize the main constraint on reversible gates as follows: it is obvious that, when running a computer forward, there must be no ambiguity in the forward step — if you have a "goto", you have to know where to go to. With a reversible machine, there cannot be any ambiguity in *backward* steps either. You should never have a situation where you do not know where to go back to. It is this latter feature that makes reversible computing radically different from ordinary, irreversible computing.

We can, following Bennett, consider the most general computational process, and also answer a criticism leveled at advocates of reversible computing. Let us suppose we have a system of (reversible) logic units tied together, and we put into it some input data. We also have to feed in a set of "standard" zeroes, the bits that are kept set at particular values to control the reversible gates. (If we want a "standard" one instead of a zero, we can just NOT one of the zeroes: this is reversible, of course!) The logic unit will do its business — dup, dup, dup — and at the end we will find an output — the answer we want plus a pile of garbage bits, forming the history tape. This is shown in Figure 5.26 below:

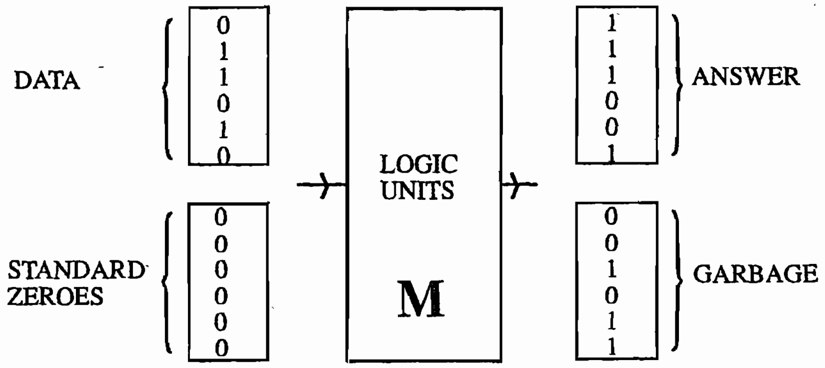


Fig. 5.26 The General Reversible Computation

Now this picture makes it look like you start up with a blank tape (or a preset one) and end up with a lot of chaos. Not surprisingly, everyone said that was where the entropy was going: "This randomization of zeroes is (in Bennett's picture) fueling the running of your machine. How can keeping this data make your computation practically reversible? It's rather like claiming that you can make an irreversible heat engine reversible by keeping the water that all the heat has gone into, rather than throwing it away. If you don't throw the water away, sure you have all the information you need about the history of the system, but that hardly means the engine is going to be able to run backwards, reversing the motions of water molecules!" In the thermodynamic case, that would indeed be silly. But it isn't so for computing. By adding one more tape to the system, and feeding the results through another machine, we can bypass this difficulty (Fig. 5.27):

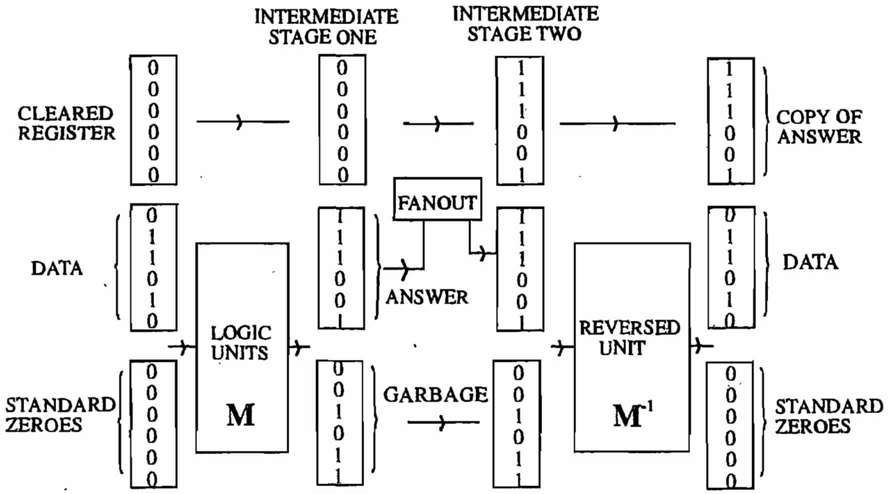


Fig. 5.27 A Zero Entropy Loss Reversible Computer

Let us try to make sense of this! The new logic unit that we have added is the reverse of the original (hence we have labeled it $\mathbf{M}^{-1}$) and is also reversible. $\mathbf{M}^{-1}$

is such that if we feed the output of **M** through this, it undoes all the work on it and feeds us back the original inputs to **M**. The new tape is a cleared register which we will use to copy the answer to our computation. We begin as before, feeding into **M** the input data and the standard bits for control. **M** gives us an output and a history tape (marked garbage in the diagram). The history tape we feed directly into $\mathbf{M}^{-1}$. We also feed the data output tape in. However, before we do this we make a copy of it onto the cleared register. We have shown this schematically as a fanout, but this actually represents a copy process (which is, of course, a reversible operation).

The reverse machine $\mathbf{M}^{-1}$ now undoes all the work done by **M**, producing as its output the standard bits and the input data. At the end of the whole process we are left with the answer to the computation, plus an exact copy of the inputs we started with. So our grand machine has done a calculation for no entropy loss (ideally — in practice we would have to drive the system a little as discussed) and reversible computing really can save us work. Of course, there will be an energy loss when we wipe our tapes clean to do another calculation.

Reversible computing is quite a strange concept for those used to thinking in classical Boolean terms, so let me suggest a few problems for you to work on to help you become more comfortable with the ideas.

Problem 5.3: Suppose a reversible computer is carrying out a calculation and it needs to execute a subroutine. So it gets sent off to some other place to execute a compact set of instructions. Now these instructions must be reversible, as are the basic computing elements, and so there is a chance that once we are into our subroutine we might find ourself running backward. It might even happen that we get back to the start of the routine — and then have to re-enter the main body of the program where we left it! The question is: Given that this same subroutine might be used several times throughout the computation, how does the machine know where to return to when this reverse happens? You might like to think about this. Somehow you have to have a number of memory stacks to keep track of where you have to go to find the subroutine, but also where to go back to should you reverse. This is your first problem in reversible computing — how to handle subroutines.

Problem 5.4: A related problem concerns how to get "if" clauses to work. What if, after having followed an "if... then..." command, the machine starts to reverse? How can the machine get back to the original condition that dictated which way the "if" branched? Of course, a set of initial conditions can result in a single "if" output ("if x = 2, 3, 4 or 6.159 let F= d"), so this condition may not be uniquely specified. Here is a nice way to analyze things. Simply bring in a new variable at each branch, and assign a unique value to this variable for each choice at a branch point. You might like to work this through in detail.

Problem 5.5: A simple question to ask about a general reversible computer is: How big a history tape do we need? The gates we have considered so far have had the number of outputs equal to the number of inputs. Is this always necessary for reversibility? As far as I know, this question hasn't even been asked by theorists. See if you can work it out. Certainly the minimum has something to do with the number of possible inputs that the output could represent, and we'll apparently need a number of bits to keep track of that (on top of the actual outputted answer). So the questions are: firstly, what is the minimum number of bits needed to keep a gate reversible in principle, and secondly, could we actually accomplish it?

5.5: The Billiard Ball Computer

To give you a demonstration of a reversible computer that can actually do calculations, I am now going to discuss an ingeniously simple machine invented by Fredkin, Toffoli and others. In this device, the movement of billiard balls on a plane is used to simulate the movement of electronic signals (bits) through logic gates. We fire balls into the machine to represent the input, and the distribution of balls coming out gives us our output. The balls all move diagonally across a planar grid and obey the laws of idealized classical mechanics (i.e zero friction and perfectly elastic collisions). To introduce you to the basic idea, examine the following diagram (Fig. 5.28):

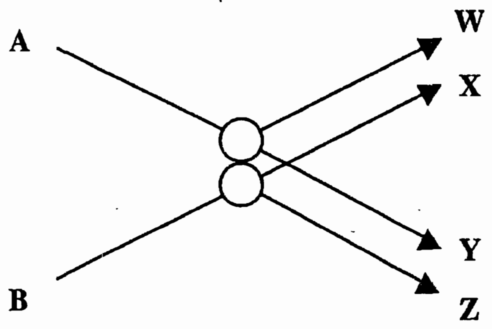


Fig. 5.28 The Basic Two-ball Collision Computation

This illustrates how a two-ball collision realizes a two-input four-output logic function. The data to this gate is represented by the presence of a ball at a particular position (1) or its absence (0). For example, the gate has two input channels, A and B. If we fire a ball in at A, then the input at A is binary 1. If there is no ball, it is zero. Similarly with B. If we find a ball coming out at point X, this means output X is 1, and so on. There are four possible input states, and for each we use basic mechanics to calculate the configuration of balls coming out of the device. There are four possible outputs, two corresponding to one input ball being absent and the other going straight through, and two corresponding to a collision.

Let us suppose there is no ball at A. If there is a ball at B, it will continue on through the "machine", coming out at X. We can see that we will only get a ball at X if there is no ball at A *and* one ball at B. In logic terms, X is 1 if and only if B is 1 and A is 0, so we have:

$$X = B \textit{ AND NOT } A \tag{5.34}$$

Similarly, we find that:

$$Y = A \textit{ AND NOT } B \tag{5.35}$$

Output W is a little trickier. We will find a ball there only if there is a ball at both A and B. Likewise for output Z. Hence, both W and Z realize the same AND function:

$$W, Z = A \textit{ AND } B \tag{5.36}$$

Let us summarize this with some fancy notation (Fig. 5.29):

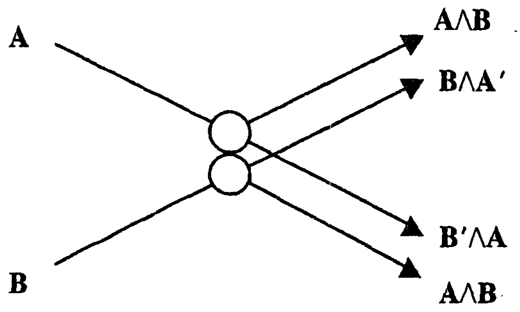


Fig. 5.29 Logical Structure of the Basic Collision Computation

This is the fundamental collision of this billiard ball computer and you can see how neatly the logic element drops out of it. We can build other logic functions besides AND with this gate. For example, we can use it to make a FANOUT. If we set $A = 1$ (the billiard equivalent of a control line set to "on") and take our output from W and Z, then clearly this has the effect of branching our B input: a ball at B produces one at each of W and Z; no ball at B leaves both outputs blank. You can also make a CN gate with this unit (try it). However, by itself, the basic collision gate will not make enough elements to build a whole computer — we'd be stuck with pairs of balls going along two lines, and we could never change anything! How do we reroute balls? We have to introduce two fundamental mechanical devices. The first, which you would never invent if you were a logician, as it seems a damn silly thing to do, I'll call a collision gate; in this device, two balls go in, but four come out (Fig. 5.30):

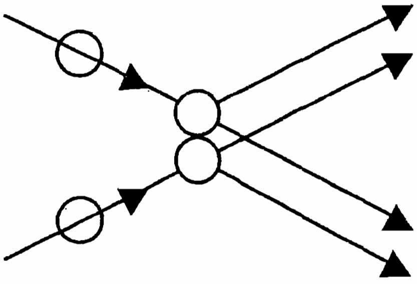


Fig. 5.30 The Collision Gate

This is a sort of all-in "double-FANOUT" process, which we achieve by letting the two incoming balls collide with two stationary ones. (You might find it an interesting exercise to consider the energy and momentum properties of this gate.) The second and more important device is a redirection gate. This is just a mirror to reflect a ball. It can be oriented any way you wish, although we restrict ourselves to four possibilities (Fig. 5.31):

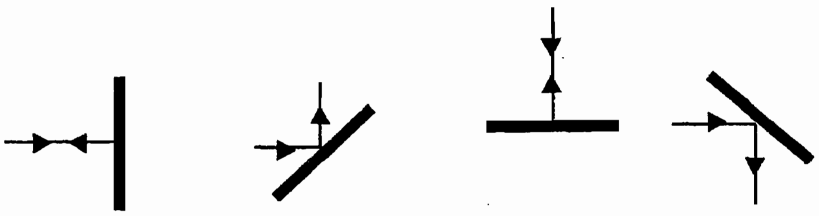


Fig. 5.31 Four Redirection Gates

Mirrors enable us to do a lot of things. For example, we can use mirrors to construct a "crossover" device (Fig. 5.32):

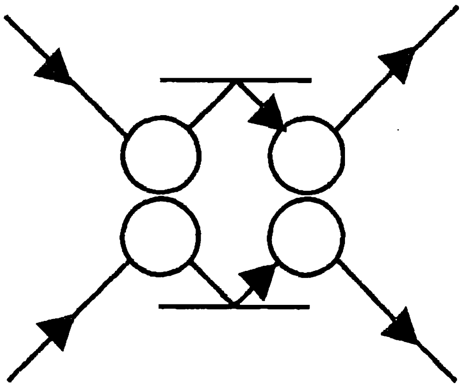


Fig. 5.32 A Crossover Device

Incidentally, this device tells us something important about the balls, namely, that they are indistinguishable. We do not tell them apart, and are interested only in their presence or absence. The above crossover device actually switches the incoming balls, but as we can't tell them apart, it looks as if they just pass each other by. Note that if one ball is missing, the other just sails right through.

To show you the sorts of thing you can build with these basic structures, I will first give you a unit that acts as a switch (Fig. 5.33):

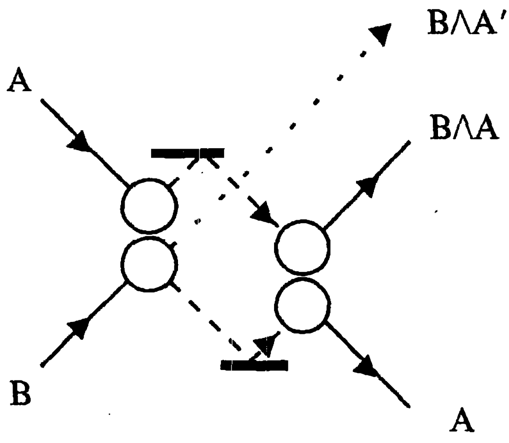


Fig. 5.33 A Switching Device

This is a sort of offset crossover. Note that, irrespective of whether or not there is a *B* input, the lower right output is always the same as *A*. This is a "debris" bit, corresponding to the control line fed through the gate. Of course, we are used to such outputs by now.

A question that arises in the context of this chapter is obviously: "OK, show me how to make reversible gates with all these mirrors and balls." Specifically, can we build, say, a CN gate? The answer is that we can, and a CCN gate too if we like. However, it is more enlightening to build a Fredkin, or controlled exchange gate. This is because it is possible to build everything we could want, just out of Fredkin gates! I'll remind you of what such a gate is (Fig. 5.34):

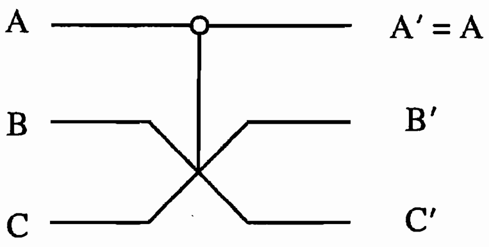


Fig. 5.34 The Fredkin Gate

Line *A* goes through unchanged. This is true of *B* and *C* also, if *A*=0; but if *A*=1, *B* and *C* switch. I won't leave building a Fredkin gate as an exercise. It is constructed from four switching devices of the kind depicted in Figure 5.33, put together with considerable ingenuity as shown in Figure 5.35:

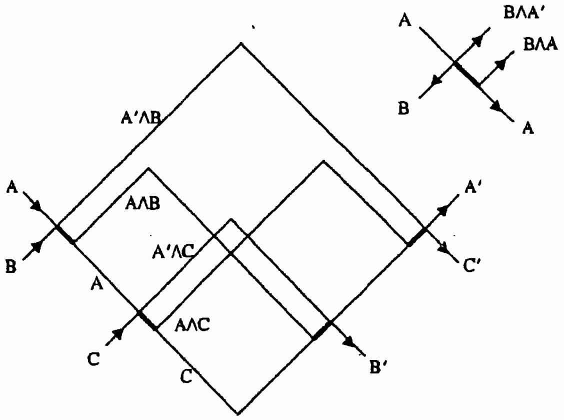


Fig. 5.35 The Fredkin Gate Realized by Billiard Ball Gates

Obviously, there is no point in making a computer like this except for fun. However, it does show how profoundly simple the basic structure of a machine can be.

Now anybody who is familiar with bouncing balls knows that if there's a slight error, it is rapidly magnified. Suppose you have a ball on a table and you drop another onto it from above, right in the middle. You might think: "Oh, it'll go straight down, then straight up, and so on." Everybody has an intuition about this but if you played with balls as a baby you know that you can't bounce one ball on another. It doesn't work! What happens is that as soon as the ball bounces ever so slightly wrongly, the next bounce is further out and the ball comes down slightly more cock-eyed. When it comes down next time, it is further out still and hits the lower ball in an even more glancing fashion. Next time, the balls will probably miss altogether.

The reasons for this are not hard to fathom. Although at the macroscopic level, balls seem stable and solid, at the microscopic level, they are a seething mass of jiggling molecules. Thermal oscillations, statistical mechanical fluctuations and whatnot, all contribute corrections to the naive collision of ideal balls. In fact, even the tiniest effects of quantum mechanics get in the way. According to the Uncertainty Principle, we cannot know both the precise location and momentum of a ball, so we cannot drop one perfectly straight. Suppose we have two ideal 1cm balls, and we drop one onto the other from a height of 10cm. How many bounces can we get away with before, according to quantum mechanics, things *have* to go awry? We can actually calculate this and the answer is about seventeen bounces. Of course, in reality the disturbances from more classical phenomena are far more significant and we would never get anywhere near this quantum limit. Don't forget, even your hand will be shaking from Brownian motion!

So surely the billiard ball machine idea is nonsense? We may not be dropping balls from a height, but we are colliding them and we would therefore expect errors to accrue just as inevitably. So how can we claim to have a physically implementable reversible computer? After all, all you have to do is give me an error per collision, and I will tell you how long you have before the machine falls apart. 10^{-3}? Five minutes. 10^{-6}? OK, ten minutes. It looks completely hopeless. In order to get this system to work, we have to find some way to keep straightening out the balls. Perhaps we could put them in troughs, guiding them in some way. But if you put a ball in a trough it'll slosh back and forth, getting worse and worse, unless there are losses — absorption, resistance, dissipation. Even if we design our troughs to cope with these difficulties, inevitably energy will be lost because of friction in the trough. We would have to pull the balls through to drive the machine. Now if you drive it just a little, you can find that the energy required to drive it is a minimum of the ratio:

$$kT \frac{\textit{time to make collision}}{\textit{speed at which it happens}}. \tag{5.37}$$

This expression has not been analyzed in any great detail for the billiard ball machine.

5.6: Quantum Computation

The billiard ball computer operates chiefly according to the laws of classical mechanics. However, inspired by the questions it brings up, people have asked me (and others have thought about this too[7]): "What would the situation be if our computer is operating according to the laws of *quantum* mechanics?" Suppose we wanted to make extremely small computers, say the size of a few atoms. Then we would have to use the laws of quantum mechanics, not classical mechanics. Wouldn't the Uncertainty Principle screw things up? Not necessarily. I will wind up this chapter by briefly considering what may become the computers of the future – quantum computers.

We are asking yet another question about absolute limitations! This time, it is: "How *small* can you make a computer?" This is one area where, I think, I've made a contribution. Unlike an airplane, it turns out that we can make it pretty much as small as we want. There will be engineering details about wires[8], and we will have to find a way of magnifying outputs and whatnot, but we are here discussing questions of principle, not practicality. We cannot get any smaller than atoms[9] because we will always need something to write on,

[7]Notably the physicist Paul Benioff (see, for example, "Quantum Mechanical Models of Turing Machines that Dissipate No Energy", Phys. Rev. Lett. 48, pp. 1581-1585 [1982]). [Editors]

[8] It is interesting to note that most computer theorists treat wire as idealized thin string that doesn't take up any room. However, real computer engineers frequently discover that they just can't get enough wires in! (We'll return to this in Chapter Seven.) [RPF]

[9]I am not allowing for the possibility that some smart soul will build a computer out of more fundamental particles! [RPF]

but all we actually need are bits which communicate. An atom, or a nucleus will do since they are natural "spin systems", i.e, they have measurable physical attributes that we can put numbers to and we can consider each different number to represent a state. We can make magnets the size of atoms. (It'll put some chemists out of a job, but that's progress). But the point is that there are no further limitations on size imposed by quantum mechanics, over and above those due to statistical and classical mechanics.

I won't go into too much detail here: I will return to the subject, and all its lovely math — in the next chapter. For now, I'll just give you the gist of the ideas. Let us begin with some idealized quantum mechanical system (anything very small) and suppose that it can be in one of two states — say "up", which might correspond to an excited state, and "down", corresponding to a de-excited state. Alternatively, the two states might refer to the spin of the quantum system (spin is a crude classical analogy). We can actually allow it to be in other states as well, but for our purposes it just needs at least two states to represent a binary number: up is one, down is zero. I'll call this quantum mechanical system an atom, so that you can get a grip on its basic nature, but bear in mind that it could be something more complex, or even something simpler, like an electron (which has two spin states). Now the idea is that we build our computing device out of such atoms by stringing them together in a particular way. We start with part or all of the system — a string of atoms in one or other of their two states — representing a number, our input. We then let the whole system evolve over time according to the laws of quantum theory, interacting with itself — the atoms change states, the ones and zeroes move around — until at some point we have a bunch of atoms somewhere which will be in certain states, and these will represent our answer.

We could set the machine running with a single input bit — say firing an atom into the system — and design things such that the machine itself tells us when the calculation is complete, say by firing an atom out of the system. Nothing would be trustable until the output bit was one. You would measure this bit, then change it to zero and freeze the answer for examination. Putting the information in and out is not, incidentally, a particularly quantum mechanical process — it is a matter of amplification. Interestingly, as a rule one cannot predict the time the computer will take to complete its calculation. It turns out to be ballistic, like Fredkin's, but at the end you only get a wave packet for the arrival of the answer. We test to see whether or not the answer is in the machine or not. For the simple machine I have designed (see next chapter), there exist several quantum mechanical "amplitudes" (certain physical properties of the system) which, upon measurement, tell us how far through the calculation we have gone, but ultimately we have to wait for the machine to let us know it's finished.

So, in 2050, or before, we may have computers that we can't even see! I will return to these strange beasts in the next chapter.