Codes

Chris Sawer

University of Liverpool Maths Club
http://www.maths.liv.ac.uk/~mathsclub/
29 March 2003

You will almost certainly have used a code today without even realising it. Have you had a conversation on a mobile 'phone? Sent an e-mail? Typed a letter on a computer and saved it to disc? Watched digital TV or listened to digital radio? Listened to a CD? Bought an item in a supermarket which has had a bar code 'scanned' at the checkout? All of these things involve digital data being transferred from one place to another. There are many different ways of doing this, but none are perfect - there is always the possibility that errors may creep in.

How come, then, when you reload that word-processed letter, that nothing has changed? Why isn't there horrendous interference on your mobile 'phone when the signal isn't very strong? The answer to these questions is the same - the information has been *encoded* so that errors can be detected and, if possible, corrected.

One example of a mathematical code that you might have encountered when ordering a book is the ISBN number. This is a ten digit number hidden away on the back of almost every book published nowadays. The first nine digits identify the book uniquely, but the tenth is a 'check digit', calculated from the previous nine.

The check digit is calculated as follows: If the digits are numbered *a-bcd-efghi-x*, then:
Find $y = 10a + 9b + 8c + 7d + 6e + 5f + 4g + 3h + 2i$
Then $x = 11 -$ (the remainder when y is divided by 11)

This method is used because it can detect two different types of error:

- If any one of the digits is mis-typed, then x will be different
- If any two of the digits are transposed (swopped over), then x will be different

These are the two mistakes most likely to be made by someone typing in the ISBN number into a computer. If the calculated x does not equal the x typed in, then the computer will flag up an error knowing that a mistake has been made somewhere. It does not know where, but this is not too important since the person typing can simply enter the number again.

We're going to look at codes used for digital data, that is binary messages which consist entirely of 0s and 1s. In the real world, this is the language used by computers. Various combinations of 0s and 1s represent different things, for example characters in a word-processed document, music on a Compact Disc, images on the Internet, or even your voice during a mobile 'phone call.

In many practical applications of digital data, there are opportunities for errors to creep in to the messages. For example, the laser in a CD player might misread a 0 for a 1. Interference or poor reception may cause your mobile 'phone to receive errors. Clearly, these could cause serious problems, and want to be minimised if at all possible.

In practical applications of digital data, therefore, extra information is added to the message which in a perfect world wouldn't be necessary, but is there to help detect and correct errors. The check digit at the end of an ISBN number is a good example of a code which can detect errors.

In many systems error detection is all that is needed, for example when transmitting data over the internet, if an error is detected, the receiving computer can ask the computer sending the information to repeat that particular piece of data. In other systems, however, such as transmission of data from remote space probes, this is not possible, so sufficient information has to be included with the data so that a small number of errors can be successfully corrected.

Some terminology I'm going to use:

Message:	The original binary data
Codeword:	The data together with coding information
Coding function:	The rule which enables you to make the codeword from any given message

To start with, let's consider a simple binary code. Each message consists of three digits, which can either be 0 or 1. This means there are a total of eight messages in the code: 000, 001, 010, 100, 110, 101, 011, 111. To make the codeword from each message, add an extra digit on the end so that the number of 1s in the resulting codeword is even.

Message	000	001	010	100	110	101	011	111
Codeword	0000	0011	0101	1001	1100	1010	0110	1111

This code is called a 'parity check'. It can detect one error in any given codeword. This is because if a 0 is changed to a 1 (or vice versa), the number of 1s in the word is no longer even.

The code cannot correct any errors, however. If the word '0010' is received, for example, we know there is an error, but we don't know if the intended codeword was '0000', '0011', '1010' or '0110' as all differ from the received word in just one digit.

How then could we make a code that can correct errors? One simple example is to repeat the message three times. Assuming the same messages as above, the resulting codewords are:

Message	000	001	010	100	110	101	011	111
Codeword	000000000	001001001	010010010	100100100	110110110	101101101	011011011	111111111

In this case if the word '101101111' is received, we can guess that it should have been '101101101' since this differs from '101101101' in just one place. It differs from all of the others in more than one place.

Of course, the word might have been intended to be '111111111' and two errors have crept in – but we always assume that the codeword closest to the received word was the one intended. This is known as the 'nearest neighbour' principle.

Mathematically, the number of places in which two words differ is called the **distance** between those words. For example, the distance between 100101 and 101111 is 2 since they differ in two places.

The distance between two words can also be calculated by 'adding' the words together but adding in a rather strange way. The method is called adding modulo two. What this means in practice is that you line the numbers to be added on top of each other, and apply the following rules to each column:

$0 + 0 = 0$
$1 + 0 = 1$
$0 + 1 = 1$
$1 + 1 = 0$ ← watch out for this!

You do not carry any digits. For example:

```
  100101
+ 101111
  001010
```

The resulting number in this case has two 1s in it, so we say it has **weight** two. The distance between two codewords can be calculated, as here, by finding the weight of their sum.

If you think about it, the distance between codewords has a lot to do with the number of errors which can be detected or corrected. If all of the codewords are very similar, then the distance between them will be small, and if a codeword is received with a few errors, it might be the same as another codeword. If you want to try and correct errors, the codewords need to have an even greater distance between them so that you can judge which is 'closest' to the received codeword, which may have errors.

In fact, you can work out how many errors a code will detect and correct by calculating the **minimum distance** between any two codewords.

The code will **detect** up to k errors when:
the minimum distance between any two codewords is $\geq k + 1$

The code will **correct** up to k errors when:
the minimum distance between any two codewords is $\geq 2k + 1$

But how can we calculate the minimum distance between any two codewords. Well, for a general code, unfortunately we have to calculate the distance between every pair of codewords, and look to see which is the minimum. This is extremely tedious, and for the code above ('tripling' the message) involves doing $7 + 6 + 5 + 4 + 3 + 2 + 1 = 28$ calculations. The answer turns out to be 3, proving that this code detects up to two errors (since $3 \geq 2 + 1$ but $3 < 3 + 1$) and can correct up to one error (since $3 \geq 2\times1 + 1$ but $3 < 2\times2 + 1$).

Is there an easier way? Well, not for all codes, but for certain codes known as **group codes**, the minimum distance can be calculated much more easily. A group code is a code in which whenever you add two codewords (using the method of addition shown above), the result is always another codeword.

To check that any given code is a group code can be a tedious process - you would have to check that adding every pair of codewords results in another codeword. However, there is a process for generating codes which guarantees the result is a group code. It's called using a **generator matrix**. A generator matrix is a rectangular array of 0s and 1s similar to the following:

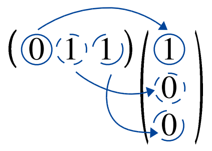
$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

But how do you find the code using this? Firstly, the number of digits in the original message has to equal the number of rows in the matrix. The number of columns then determines the number of digits in the codewords. To find the codeword corresponding to any particular message, you 'multiply' it by the above matrix as follows. Place the message to the left of the matrix. The codeword then consists of the 'dot products' of the message and each individual column of the matrix. This means multiplying the first digit of the message by the top entry in the column, the second digit of the message by the second entry in the column, and so on, down to the bottom of the column. Adding all of these results together gives the dot product. Be careful - we are still working modulo two which means that 1 + 1 = 0, so 1 + 1 + 1 = 1, 1 + 1 + 1 + 1 = 0, etc.

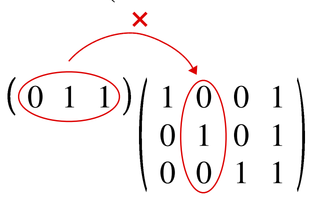
The following example might help: multiplying the message '011' by the above matrix:

$$\begin{pmatrix} 0 & 1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} = ?$$

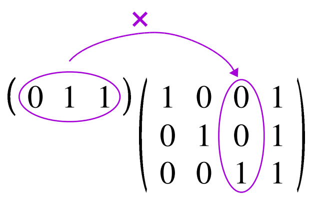
The first diagram below shows the dot product of the message and the first column of the matrix. The others show the dot products with the other columns (the relevant one is circled each time):



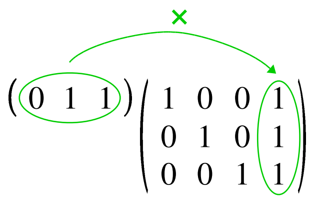
$= 0\times1 + 1\times0 + 1\times0 = 0$



$= 0\times0 + 1\times1 + 1\times0 = 1$



$= 0\times0 + 1\times0 + 1\times1 = 1$



$= 0\times1 + 1\times1 + 1\times1 = 0$

Therefore:

$$\begin{pmatrix} 0 & 1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 0 \end{pmatrix}$$

So the resulting codeword is '0110'. In fact, this matrix produces the 'parity check' code we saw earlier. Check this by multiplying some other messages by it and seeing that the result always has an even number of 1s. When you've done this, see if you can have a go at making a matrix for the other code we've looked at (the one which 'triples' the message).

As I stated earlier, the minimum distance between codewords is much easier to calculate for a group code. In fact:

The minimum distance between codewords = *the lowest weight of a non-zero codeword*

Remember that this is how you work out how many errors the code can detect and correct.

Now, suppose we have received an encoded message with possible errors. We want to decipher the original message by seeing which codeword each received word is 'closest' to. For each received word we could find the minimum distance between it and each of the actual codewords, but this is tedious and repetitive. A better way is to form a table called a **coset decoding table**. The top row of the table consists of all the codewords that can be generated using the generator matrix, and the rest of the table consists of **all** the other possible combinations of 0s and 1s which might be received, each of which is positioned underneath the codeword that it's closest to. Constructing this table might seem like a difficult task, but once done, it enables you to find the corrected version of any received codeword is extremely quickly – it is simply the codeword at the top of the column containing the received word.

It's probably easiest to explain this with an example, so, let's look at the following code:

$$G = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

000	001	010	011	100	101	110	111
000000	001110	010011	011101	100101	101011	110110	111000

First, write the codewords as the top row of the table:

000000	001110	010011	011101	100101	101011	110110	111000

For the next row, pick a word with the smallest weight possible that is not already in the table. We'll start with '000001'. Write this under the '000000' column, and calculate the rest of the row by adding it to the other codewords:

000000	001110	010011	011101	100101	101011	110110	111000
000001	001111	010010	011100	100100	101010	110111	111001

Continue picking words with the smallest weight possible and repeating the above until all words ($2^6 = 64$ in this case) appear in the table:

000000	001110	010011	011101	100101	101011	110110	111000
000001	001111	010010	011100	100100	101010	110111	111001
000010	001100	010001	011111	100111	101001	110100	111010
000100	001010	010111	011001	100001	101111	110010	111100
001000	000110	011011	010101	101101	100011	111110	110000
010000	011110	000011	001101	110101	111011	100110	101000
100000	101110	110011	111101	000101	001011	010110	011000
001001	000111	011010	010100	101100	100010	111111	110001

Note that there were three choices for the first word in this final row. This is because all the words in this row contain two errors, and the code can only correct one. Therefore each word in this row is equally close to several codewords, so there are several choices each time.

Final task: Suppose that a message is encrypted using the following number-to-letter equivalents:

000:	E	001:	G	010:	M	011:	O
100:	Q	101:	R	110:	T	111:	Y

The code used is as above. The message received is:

101110 100000 011001 011011 001000 110110 111011 110000

There are many errors in this message. Just taking the first three digits of each of the received words, ie. ignoring the error detection/correction gives the following word: RQOOGTYT

This is clearly wrong. Use the table above to work out what the message should have been.