Checksums and error control

Peter M. Fenwick

March 22, 2004

1 Error Control Codes

Computing has always had to live with errors, especially in data transmission and data recording. Sometimes these errors are only a nuisance and a simple retry can obtain satisfactory, accurate, data. But sometimes an error can be serious, and perhaps even disastrous if an accurate original copy is inaccessible.

Two related, but somewhat parallel disciplines, have developed to deal with the handling of erroneous data, both part of the general theme of "Coding Theory" and collected under the generic title of "Error Control".

Error Detection Error Detection extends the ideas of parity to provide powerful and reliable detection of errors, usually by appending a "checksum" of 8, 16 or 32 bits to the data. The checksum is carefully designed to be sensitive to the probable errors: a checksum for manual data entry must be sensitive to digit transposition and repetition, while one for data transmission must detect long bursts of errors.

A detected error invariably leads to an alarm of some sort and request for data re-entry or retransmission.

Error Correction Error correction is required if the original data is remote either in space (such as telemetry) or in time (such as data recording). In both cases the data must carry sufficient redundant information to allow the original to be reconstructed in the presence of an error. While methods for handling single-bit errors have been known for many years and errors of just a few bits for nearly as long, few data errors are that simple. The methods for coding on physical transmission or recording media mean that many single errors at the *physical* level become bursts of errors at the *data* level. Burst error correction is then important, but is unfortunately a very difficult topic.

Despite the division of error control into two fields, many of the techniques in one can be applied to the other. In particular, the better error detection codes are based on polynomial generators and Galois field arithmetic. Exactly the same techniques can be applied to some of the simpler error correcting codes, perhaps just be choosing a different generator polynomial. A consequence of

this convergence is that a suitable long checksum can often provide some degree of error correction over a short message. An example is found in the ATM cell header which is protected by an 8 bit checksum (or "Header Error Control" field – HEC), far longer than is usually needed for the 32 bits of the header. Although it is designed for *error detection*, the HEC can provide some *error correction* as well.

This chapter emphasises codes for error detection where it is possible to repeat the entry or transmission. Codes for error correction are touched on only briefly, describing Hamming codes one of the older and simpler error correcting codes. A full discussion of error correction codes is far beyond the intended scope of this book.

2 More on Parity

The simplest form of error control adds a single parity bit to a byte, word, or other simple data unit. Simple parity is fine for detecting very occasional errors, but becomes less satisfactory for higher error probabilities and for longer data.

With a bit error probability of p and assuming independent errors, the probability P_k of an n-bit message having k errors is

$$\begin{aligned} P_0 &= (1-p)^n \\ P_1 &= n(1-p)^{n-1}p \\ P_2 &= \frac{n(n-1)}{2}(1-p)^{n-2}p^2 \\ &\dots \\ P_k &= \frac{n(n-1)\dots(n-k+1)}{k!}(1-p)^{n-k}p^k \\ \text{if } p \approx 0, \text{ then } P_2 &\approx \frac{(np)^2}{2} \quad \text{the probability of an undetected error} \end{aligned}$$

Unfortunately a single parity bit detects only *odd* numbers of errors and does not detect even numbers of errors. For a message of 12500 octets ($n = 100000$ bits) and a bit error probability $p = 10^{-6}$, the probabilities are

P_0	90.5%	probability of no errors
P_1	9.05%	probability of one error
P_2	0.45%	probability of two errors (undetected)
$P_{k>0}$	10.5%	probability of at least one error
P_{odd}	10.0%	probability of any odd errors (detected)
P_{even}	0.50%	probability of any even errors (undetected)

Thus even though 10.5% of messages have detected errors and should be retransmitted, 0.5% of the errors remain undetected and are falsely reported as "correct".

Except where errors are *very* infrequent, practical error control uses much more powerful and complex checking functions, with the checking spread over several inter-related checking bits, so that even a single error affects several parity bits and multiple errors are unlikely to cancel out and give a "false positive".

char	P	a	r	i	t	y		c	h	e	c	k	s	
hex	50	61	72	69	74	79	20	63	68	65	63	6B	73	HP
VP	0	1	0	0	0	1	1	0	1	0	0	1	1	0
MSB	1	1	1	1	1	1	0	1	1	1	1	1	1	0
	0	1	1	1	1	1	1	1	1	1	1	1	1	0
	1	0	1	0	1	1	0	0	0	0	0	0	1	1
	0	0	0	1	0	1	0	0	1	0	0	1	0	0
	0	0	0	0	1	0	0	0	0	1	0	0	0	0
	0	0	1	0	0	0	0	1	0	0	1	1	1	1
LSB	0	1	0	1	0	1	0	1	0	1	1	1	1	0

Figure 1: Horizontal and Vertical Parity on a Message

Many of the checks described here are for strings of decimal digits, while others apply to sequences of bytes or octets.

As a simple extension of parity, Figure 1 shows an ASCII message with both character (vertical) parity and message (horizontal) parity as was used on some early ASCII terminals with block-mode transmission. The top row shows the characters of the message and the line below that their encoding in (7-bit) ASCII. Below that row VP shows the even *vertical parity* of each character based on the bits as shown in the remaining rows. At the extreme right, the column headed HP shows the *horizontal parity* for the entire message, where each bit is the Exclusive-OR of all preceding bits in that row (or bit-position within the characters). It is again even parity, though odd parity can be used if desired in either case. An error is usually signalled if the vertical parity fails for *any* character or if the overall horizontal parity fails.

The 2-dimensional parity is much better than the simple 1-dimensional parity in detecting errors; only errors which occur in fours, in positions on the corner of a rectangle will escape detection. (If it is known that only one error has occurred, then that error can be corrected at the intersection of the failing row and column parities.) It was soon superseded by the much more powerful CRC-16 checks, described in Section 5.3.

3 Hamming Codes

The Hamming code [8] is one of the oldest and simplest of the error-correcting codes and is a good example of a Single Error Correcting (SEC) code. For the simplest non-trivial case (and the one which is the usual example) take 4 data bits and 3 parity bits and arrange them in a 7-bit word as $d_7 d_6 d_5 p_4 d_3 p_2 p_1$, with the bits numbered from 7 on the left to 1 on the right. The bits whose numbers are of the form 2^k are used as parity bits, with the other bits used as data.

On transmission set the parity bits as below, transmitting the entire 7-bit word as the codeword.

$$
\begin{aligned}
p_1 &= d_3 \oplus d_5 \oplus d_7 \quad \text{(the bits with a “1” in the bit number)} \\
p_2 &= d_3 \oplus d_6 \oplus d_7 \quad \text{(the bits with a “2” in the bit number)} \\
p_4 &= d_5 \oplus d_6 \oplus d_7 \quad \text{(the bits with a “4” in the bit number)}
\end{aligned}
$$

On reception, calculate the *syndrome* $S = \{s_4 s_2 s_1\}$ from the equations

The raw data	1	1	0	1						
Bit numbers	7	6	5	4	3	2	1			
position data for Hamming	1	1	0	.	1	.	.			
generate even parities	.	.	.	0	.	1	0			
combine for codeword	1	1	0	0	1	1	0			
bit#6) corrupted	1	0	0	0	1	1	0	syndrome		
generate syndrome	1	0	0	0	1	1	0	1	1	0
syndrome = 6, bit error at		↑								

Figure 2: Example of Hamming correction

$$s_1 = p_1 \oplus d_3 \oplus d_5 \oplus d_7$$
$$s_2 = p_2 \oplus d_3 \oplus d_6 \oplus d_7$$
$$s_4 = p_4 \oplus d_5 \oplus d_6 \oplus d_7$$

If $S = 0$, then the bits are all correct. If $S \neq 0$, then it gives the number of the bit in error (assuming just one error).

The operation of a Hamming Code is shown in Figure 2

The Hamming code is easily extended to longer words, by using each bit number 2^k as a parity bit, but does not extend to correcting more than one error. The example used here is often written as a (7,4) code; each codeword has 7 bits, with 4 of those for user data. A general SEC Hamming Code is described as a $(2^k - 1, 2^k - 1 - k)$ code. Examples are the (15,11) and (31,26) codes.

Adding a single parity bit gives a code which is able to detect two errors if an internal parity fails, but the overall parity still succeeds, giving a Single Error Correcting, Double Error Detecting (SEC-DEC) code.

4 Modular Checkdigits and checksums

Most of the checks described here use some form of modular arithmetic, deriving some relatively large value from the data and reducing that value to a smaller one by taking its remainder on division by some modulus. In very simple cases we may just use a modulus of 10 (which delivers the units decimal digit), or 256 (to give the least significant 8 bits). Generally the modulus is chosen using some less obvious criterion which maximises the ability to detect errors. The simpler techniques use ordinary numerical division and are more suitable for software calculation, while others use polynomial division and are better for hardware implementation. For example, many checking algorithms work best if the modulus is a prime number. Ordinary parity is the simplest example of modular arithmetic, taking the sum of the bits modulo 2.

4.1 Modular Arithmetic

When adding or subtracting mod p, it is necessary only to divide every value by p and take the positive remainder. Multiplication and division require more care.

We say that a "number a is congruent to a' modulo m" if both a and a' give the same remainder on division by m. This relation is written

$$\begin{aligned} a &\equiv a' \bmod m \\ b &\equiv b' \bmod m \end{aligned}$$

Consider the particular case $m = 12$, $a = 21$ and $b = 20$. Then

$$a' = 9 \text{ and } b' = 8$$

and

$$ab \equiv a'b' \equiv 0 \bmod 12$$

despite neither a nor b being congruent to zero modulo 12. Only for a prime modulus do we have that if a product is zero, then at least one factor is zero. Given that many checksums work by forcing an overall value to be congruent to zero, this is a very important requirement.

4.2 Parity and Arithmetic Checksums

In the following descriptions we will use the generic term *digit* for the basic data. Depending on the context it may be a decimal digit (range $0 \ldots 9$), a byte (range $0 \ldots 255$), a 32-bit word (range $0 \ldots 2^{31} - 1$), etc. In many respects the algorithms are similar for all cases.

For simplicity the term *checksum* will include terms such as *parity* and *check-digit*. *Digit* will include *byte*, *character* and *word* as whatever is the major data unit entering into the calculation. More usually though, the term *check digit* is used where the entities being checked are decimal digits (human readable) and the check is itself a decimal digit (or is usually decimal).

logical sum A checksum is formed by bit-wise Exclusive-ORing together all the bytes or words of the message.

$$\{0010, 1010, 1001, 0001, 0110\} \rightarrow 0110$$

Each bit of the checksum is the Exclusive-OR of the corresponding bits of each data word, as shown in the horizontal parity of Figure 1. Problems with this approach are that errors have limited effect on the checksum, and that it does not detect transpositions (unimportant for data transmission, but crucial for data entry).

arithmetic sum This resembles the logical sum, except that the Exclusive-OR is changed to a conventional arithmetic addition. With this change, the carries give some inter-dependence between bits of the checksum, but it is still insensitive to data transpositions.

With most computers the obvious addition uses 2s complement, reducing the sum modulo 2^N for an N bit word. However, with the carries propagating from least- to most-significant bits, the more-significant bits are much more sensitive to errors than are the less-significant bits. Changing to 1s-complement addition (adding modulo $2^N - 1$ rather than modulo

2^N), allows the *end-around* carry to give an overall symmetry to the operation with low-order checksum bits affected by changes in high-order data bits.

This is the checksum used in TCP/IP. While it is computationally simple and better than a simple Exclusive-OR, it is not as good as the Fletcher or Adler checksums described later.

Tests on real data by Stone et al[12] show that the TCP/IP checksum is not good for many types of real-world data, such as character strings and even real numbers where there may be high correlations between adjacent words. They show that checksum values are far from uniformly distributed, and that the 16-bit TCP/IP arithmetic checksum may be no better than a 10-bit CRC.

4.3 Digit Checksums

The checksums of this section are all designed to check decimal numbers, and especially ones which are manually entered. Frequent errors during manual data entry are duplication or deletion of digits, transposition of adjacent digits and substitutions such as "667" for "677". Simple parity and arithmetic sums have the disadvantage that all digits are treated identically; to handle transposition errors the digits must be treated differently so that the checksum is also dependent in some way on the position of each digit. Most of the examples for digit checksums use systematically varying weights for successive digits. Wagner and Putter discuss using decimal check digits for a particular application[13].

The examples will assume a string of decimal digits

$$\ldots d_6\ d_5\ d_4\ d_3\ d_2\ d_1\ d_0$$

where the subscript corresponds to the power of 10. The digits $d_6 \ldots d_1$ will be the supplied data and d_0 the checksum digit.

4.3.1 IBM Check

This check, discussed briefly in [4] [p 49], is given as an example of a very simple checksum which involves minimal computation and is appropriate to electromechanical equipment, such as the IBM 026 Card Punch. It will detect adjacent transpositions (but not $09 \leftrightarrow 90$) but because of the simple repeating pattern of weights is insensitive to many other errors.

Form the sum of the *even* digits plus twice the *odd* digits.

$$s = \sum d_{2i} + 2 \times \sum d_{2i+1}$$

and the check digit is the 10s complement of the last digit of s. Thus the check digit c is

$$c = 10 - \left(\sum d_{2i} + 2 \times \sum d_{2i+1} \right) \bmod 10$$

Wagner and Putter[13] describe a similar algorithm (possibly the same one given that they provide more details) which is used for some account numbers and is known as the *IBM check*. The possible difference is that when a digit is doubled and exceeds 10, the two digits of the sum are added. Thus $2 * 3 \rightarrow 6$, while $2 * 7 \rightarrow 14 \rightarrow 5$.

5 Modular checkdigits

This method is described Hamming [9][p 28 ff] but is widely used in many other contexts as well. All digits are weighted by their position in the input number. While it might seem natural to have the weights increasing from left to right, the usual *sum of sums* algorithm assigns weights increasing from right to left.

The value (including checkdigit), must have $\sum(i+1)d_i \bmod m \equiv 0$.

To generate the sum without multiplication (or even prior knowledge of the number of digits), progressively form the sum of the digits, in order left to right, and at each stage add the sum into a running *sum of the sums*. To illustrate with the successive digits $p\ q\ r\ s\ t$.

Message	Sum	Sum of Sums
p	p	p
q	$p+q$	$2p+q$
r	$p+q+r$	$3p+2q+r$
s	$p+q+r+s$	$4p+3q+2r+s$
t	$p+q+r+s+t$	$5p+4q+3r+2s+t$

Hamming [9] gives an example of checksumming a combination of the letters "A"... "Z", digits "0"... "9" and space "␣". This gives an alphabet of 37 symbols, conveniently a prime number.

A better example is the ISBN (International Standard Book Number). The ISBN is a sequence of 9 decimal digits indicating the country, the publisher and a sequence number for the book. The digits are combined with a *sum of sums* as above and reduced modulo 11 to give a check digit, written as the final, 10th, digit of the ISBN. With a modulus of 11, the check digits can range from 0 to 10. While we could just ignore values with a checkdigit of 10, that wastes $^1/_{11}$ of the available numbers. Instead, a check digit of 10 is represented by "X", giving an ISBN such as 0 7112 0232 X.

To compute the checksum, calculate $\ldots 8d_7+7d_6+6d_5+5d_4+4d_3+3d_2+2d_1$, reduce the sum modulo 11 and take the 11s complement of the result.

$$c = 11 - (\sum(i+1)d_i) \bmod 11$$

To illustrate the verification of an ISBN, consider the example above –

digit	sum		sum of sums	
0	$0+0$	$=0$	$0+0$	$=0$
7	$0+7$	$=7$	$0+0$	$=7$
1	$7+1$	$=8$	$7+8$	$=15$
1	$8+1$	$=9$	$15+9$	$=24$
2	$9+2$	$=11$	$24+11$	$=35$
0	$11+0$	$=11$	$35+11$	$=46$
2	$11+2$	$=13$	$46+13$	$=59$
3	$13+2$	$=16$	$59+16$	$=75$
2	$16+2$	$=18$	$75+18$	$=93$
X	$18+10$	$=28$	$93+28$	$=121 \equiv 0 \bmod 11$

The final sum of sums is a multiple of 11, showing that this is a valid ISBN.

5.0.2 ID checksum

This is a variation of the modular checksum which is often used for ID number checksums and the like. In this case the digit weights are successive powers of 2 and a check digit again makes the result 0 modulo 11. More formally, the checkdigit is calculated as

$$d_0 = c = 11 - \left(\sum 2^{i+1} d_i\right) \bmod 11$$

and then, for a 6-digit number with checkdigit,

$$2(d_0 + 2(d_1 + 2(d_2 + 2(d_3 + 2(d_4 + 2(d_5 + 2d_6)))))) \bmod 11 \equiv 0$$

(Taking digits from the left, double the *sum-so-far* and add in the next digit.) The polynomial may be written in a more familiar form as

$$\left(\sum_{i=0}^{n} 2^i d_i\right) \bmod 11 \equiv 0$$

To confirm that 6051001 is indeed a valid checked number in this system

$$6 \times 64 + 0 \times 32 + 5 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 = 473$$

As $473 = 43 \times 11$ the result is (0 mod 11) and is correct.

In contrast to the ISBN, values with a checkdigit of 10 are rejected.

A possible variation using a modulus of 7 allows all numbers to be handled (no rejects) but at the cost of decreased error detection. This possibility has not been investigated. However Wagner and Putter[13] describe a similar modulo 97 code which appends *two* check digits.

5.0.3 Dihedral Group Checksum

This method is discussed in detail by Wagner and Putter[13], who cite both Verhoeff's original paper[14] and its rediscovery by Gumm[7]. The algorithm uses operations in the dihedral group D_5, which is related to symmetries of a pentagon. In particular, multiplication in D_5 is not commutative, so that $a * b \neq b * a$, where $*$ denotes multiplication in D_5. Instead of addition using a simple pattern of weights to differentiate the incoming digits, the digits are first subjected to a *Permutation*, (perhaps more correctly a *substitution*) where each is replaced by another, the permutation function depending on the digit position. The permuted digits are then multiplied in D_5 to give the checksum.

The algorithm is most easily implemented with supporting tables –

Multiplication table This is a 10 by 10 matrix where each element corresponds to the product of its indices (0-origin) in D_5.

$$M = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 2 & 3 & 4 & 0 & 6 & 7 & 8 & 9 & 5 \\ 2 & 3 & 4 & 0 & 1 & 7 & 8 & 9 & 5 & 6 \\ 3 & 4 & 0 & 1 & 2 & 8 & 9 & 5 & 6 & 7 \\ 4 & 0 & 1 & 2 & 3 & 9 & 5 & 6 & 7 & 8 \\ 5 & 9 & 8 & 7 & 6 & 0 & 4 & 3 & 2 & 1 \\ 6 & 5 & 9 & 8 & 7 & 1 & 0 & 4 & 3 & 2 \\ 7 & 6 & 5 & 9 & 8 & 2 & 1 & 0 & 4 & 3 \\ 8 & 7 & 6 & 5 & 9 & 3 & 2 & 1 & 0 & 4 \\ 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \end{pmatrix}$$

Inverse This gives the multiplicative inverse of i, so that $i * \mathrm{Inv}[i] \equiv 0$ in D_5.

$$I = \begin{pmatrix} 0 & 4 & 3 & 2 & 1 & 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$

Permutation Function The successive digits are combined by an equation $f_1 a_1 * f_2 a_2 * \ldots * f_n a_n$, where the successive f_i are permutation functions, defined by successive applications of an initial function. Note that $F[i, j] \equiv F[i \bmod 8, j]$.

$$F[1, j] = [\ 1 \quad 5 \quad 7 \quad 6 \quad 2 \quad 8 \quad 3 \quad 0 \quad 9 \quad 4\ ]$$

and

$$F[i, j] = F[i - 1, F[1, j]]$$

giving

$$F = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 5 & 7 & 6 & 2 & 8 & 3 & 0 & 9 & 4 \\ 5 & 8 & 0 & 3 & 7 & 9 & 6 & 1 & 4 & 2 \\ 8 & 9 & 1 & 6 & 0 & 4 & 3 & 5 & 2 & 7 \\ 9 & 4 & 5 & 3 & 1 & 2 & 6 & 8 & 7 & 0 \\ 4 & 2 & 8 & 6 & 5 & 7 & 3 & 9 & 0 & 1 \\ 2 & 7 & 9 & 3 & 8 & 0 & 6 & 4 & 1 & 5 \\ 7 & 0 & 4 & 6 & 9 & 1 & 3 & 2 & 5 & 8 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 5 & 7 & 6 & 2 & 8 & 3 & 0 & 9 & 4 \end{pmatrix}$$

Assuming that the digits to be checked are in positions $1 \ldots n$ of an array `dig` and that `dig[0]` is the check digit, the two checking functions are

```
Boolean checkDihedral()   // TRUE if valid checksum
  {
  int check = 0;          // the running check digit
  for (int i = 0; i <= n; i++)
    check = M[check, F[i % 8][dig[i]]];
  return check == 0;      // TRUE if valid check
  }

void computeDihedral()
  {
  int check = 0;          // the running check digit
  for (int i = 0; i <= n; i++)
    check = M[check,F[i % 8][dig[i]]];
  dig[0] = I[check];      // set the check digit
  }
```

In contrast to the simpler mod-11 algorithms, the dihedral check has the advantage that *any* combination of data digits can be checksummed. There is no need to reject those with an unrepresentable check digit. This advantage comes at the cost of a much less comprehensible algorithm, which depends on rather inaccessible mathematics.

Despite the undoubted quality of the dihedral algorithm, Wagner and Putter caution against its use, especially in commercial applications which may be maintained by less-skilled programmers who do not understand the mathematics[1]. In their paper, they describe a system where the customer wanted 4 check digits for 8 data digits; their solution involves three nested checks. First is a mod-11 check on the 8 data digits, expanding to 9 digits. Next is a mod-97 check on those 9 digits, to a total of 11 digits. Finally, the 11 digits are subjected to a mod-10 IBM check. The resulting code may be inferior to one using 4 check digits and based on advanced mathematics, but its three stages are comprehensible to users with modest mathematical ability. Comprehensibility is often preferred to intellectual excellence.

5.1 Fletcher Checksum

The Fletcher checksum[6] [11] was developed for the Transport Layer (Level 4) of the OSI communication model. It is fundamentally a sum of sums method, with all additions done modulo 255. (But note that 255 is *not* prime!) Thus to add in the digit d_i we calculate

$$\begin{aligned} s1 &= s1 + d_i \bmod 255 \\ s2 &= s2 + s1 \bmod 255 \end{aligned}$$

If the checksum is at the end of the message (the usual case) the two check-bytes are set to $B_1 = s1 - s2$ and $B_2 = -2s1 + s2$ to make the checksum including the two check bytes sum to zero. Testing for correct transmission is a little different from many checksums, because the result is correct if *either* $s1 = 0$ or $s2 = 0$. An error is signalled only if *both* sums are non-zero.

If the checksum bytes are at position n and $n + 1$ of an L-octet message (numbering $0 \ldots L - 1$), then

$$\begin{aligned} b_n &= (L - n) \times s1 - s2 \text{ and} \\ b_{n+1} &= s2 - (L - n + 1) \times s1 \end{aligned}$$

A Java fragment to calculate the checksum given an array `c[ ]` of `nChar` characters follows. Note that the characters should be limited to ASCII characters, or at least to values < 256. The `while` loops are equivalent to division with remainder, but should be faster here, where more than one correction is seldom needed.

```
int s1 = 0,  s2 = 0;              // initialise checksums
for (int i = 0; i < nChars; i++)  // scan the characters
  {
  s1 += c[i];                     // add in the character
  while (s1 >= 255)               // reduce modulo 255
    s1 -= 255;
  s2 += s1;                       // get the sum of sums
```

[1]Similar comments were made by Knuth when describing what is now called the *Knuth-Morris-Pratt* pattern matching algorithm. An earlier version of the algorithm, carefully designed according to finite-state machine theory was, within a few months, "hacked" beyond recognition by well meaning but ignorant programmers.

```
while (s2 >= 255)               // modulo 255
  s2 -= 255;
}
```

The Fletcher checksum is stated to give checking nearly as powerful as the CRC-16 checksum described below, detecting –

- all single-bit errors,
- all double-bit errors,
- all but 0.000019% of burst errors up to length 16, and
- all but 0.0015% of longer burst errors.

5.2 Adler Checksum

The Adler checksum [5] is a development of the Fletcher checksum which generates 16-bit sums and a 32-bit checksum. It was devised particularly for the GZIP text compressor. For each digit (or byte)

$$\begin{aligned} s1 &= s1 + d_i \bmod 65521 \\ s2 &= s2 + s1 \bmod 65521 \end{aligned}$$

The checksum is the 32-bit value $65536 * s1 + s2$, transmitted most- significant byte first. The values are initialised with $s1 = 1, \quad s2 = 0$ to give a length-dependent checksum for all-zero data.

Note that the modulus 65521 *is* prime, removing one doubtful feature about the design of the Fletcher checksum.

5.3 Cyclic Redundancy Checks

These are the most important and widespread of the error-detecting codes. They are especially suitable for hardware implementation at very high operating speeds and are used in most data communications systems. They are still based on modular arithmetic, but with some major changes from the earlier examples –

1. The "number system" is changed from the conventional and familiar integers to one of *finite fields*, specifically $GF(2)$. All of the arithmetic is performed modulo 2, as in point 3 below.

2. The bits are regarded as coefficients in polynomials. This allows the very highly developed and powerful mathematics of finite fields and polynomial fields to be applied to the theory of error control coding, both error detection and error correction.

   The expression as polynomials also allows a convenient representation for bit vectors which often have many zero elements. Only the terms corresponding to 1s appear in the polynomial and they very conveniently have the bit position shown explicitly as the exponent. Thus these two representations are equivalent

$$1\ 0\ 0\ 1\ 0\ 1 \Longleftrightarrow x^5 + x^2 + 1$$

$$
\begin{array}{rcl}
 & & 1\ 0\ 1\ 0\ 1\ 1\ 0 \\
1\ 0\ 1\ 1 &) & 1\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0 \\
 & - & 1\ 0\ 1\ 1 \\
\hline
 & & 0\ 1\ 0\ 0 \\
 & & \ \ 1\ 0\ 0\ 0 \\
 & - & 1\ 0\ 1\ 1 \\
\hline
 & & \ \ 0\ 1\ 1\ 1\ 0 \\
 & & \ \ - \ 1\ 0\ 1\ 1 \\
\hline
 & & \ \ \ \ 1\ 0\ 1\ 0 \\
 & & \ \ \ \ - \ 1\ 0\ 1\ 1 \\
\hline
 & & \ \ \ \ \ \ 0\ 1\ 0
\end{array}
$$

Figure 3: Polynomial division — $x^3(x^6 + x^3 + 1) \div x^3 + x + 1$

as are also

$$1\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1 \Longleftrightarrow x^8 + x^2 + x + 1$$

The polynomial variable is truly a dummy variable with little significance to most of the coding process. It could be regarded as a "carrier" for the exponents.

3. Numerical arithmetic in integers is replaced by logical operations on the bits in the $GF(2)$ finite field. Addition and subtraction are now *both* equivalent to an Exclusive-OR ($\oplus$) and multiplication is equivalent to a logical AND ($\wedge$). (In both cases, regard the bits as numerical values, do the numerical operations and take the result modulo 2.) There is no carry propagation between bits, which immediately removes one of the main impediments to fast addition.

   Practically, this change means that the arithmetic can be done very easily and quickly by simple logic. While not developed to any great extent here, this is a great incentive for using these methods in fast hardware.

The most visible operation for cyclic redundancy checks is polynomial division, as shown in Figure 3. Except for the slightly changed subtraction rules the overall method is precisely that of traditional long division. The divisor, which is always a constant, is normalised with its most-significant bit a 1. (In practise its least significant term is also a 1, giving a polynomial of the form $x^N \ldots 1$. Technically, $g(x)$ is *monic*.) Because there is no carry propagation in subtraction, the divisor can be subtracted from the partial remainder whenever the most significant bit of the remainder is a 1; there is no concept of a "trial subtraction" or compensation for overdraws as needed in integer division. The form of the dividend and the way it is written in this example are deliberately chosen to fit with the use of polynomial division in forming CRCs ($i(x) = x^6 + x^3 + 1$ and $g(x) = x^3 + x + 1$).

When we apply polynomials to checksum generation, the transmitted data forms a 1-dimensional bit stream, with earlier bits corresponding to higher-powers within the polynomial. There are several polynomials involved in transmitting data and checking for correct transmission –

information polynomial $i(x)$ The information polynomial is the transmitted data as provided by the user (usually including headers, addresses and

other transmission control information). The information polynomial is usually transmitted without modification.

generator polynomial $g(x)$ The information polynomial is divided by the generator polynomial and the remainder from that division is appended as the checksum. Usually zeros corresponding to the degree of $g(x)$ are appended to $i(x)$before the division.

codeword polynomial $c(x)$ Appending the checksum from the division to the information polynomial forms the codeword polynomial, which is what is actually transmitted.

error polynomial $e(x)$ During transmission one or more bits of $c(x)$ may be corrupted. The corrupted positions may be regarded as a polynomial, the *error polynomial* $e(x)$.

received codeword $v(x)$ This is what is received after corruption in transit. As $e(x)$ marks the corrupted bits in the transmitted data, then clearly $v = c \oplus e$, assuming a term-by-term exclusive-OR, or $v(x) = c(x) + e(x)$.

In more detail, if r is the degree of $g(x)$,

1. append r low-order zeros to $i(x)$, to form $x^r i(x)$.
2. calculate $(x^r i(x) \bmod g(x))$, the remainder on division by $g(x)$
3. append that remainder to $i(x)$ to form $c(x)$, the transmitted codeword. Thus the transmitted codeword

$$c(x) = x^r i(x) - (x^r i(x) \bmod g(x))$$

is always a multiple of $g(x)$. (Step 1 ensures that the whole of $i(x)$ is processed by $g(x)$ and also creates a space into which the remainder may be written.

4. On reception compute

$$\begin{aligned} r(x) &= v(x) \bmod g(x) \\ &= e(x) \bmod g(x) + c(x) \bmod g(x) \\ &= e(x) \bmod g(x) \text{ , as } c(x) \bmod g(x) \equiv 0 \text{ by construction} \end{aligned}$$

An error will be undetected if and only if $e(x)$ is a multiple of $g(x)$. The design of the generator polynomial $g(x)$ therefore determines the ability to detect errors and is in turn determined by its relationship to $e(x)$.

- If there is single-bit error, then the error polynomial is $e(x) = x^i$, where i determines the bit in error. If $g(x)$ contains two or more terms it will never divide $e(x)$ and all single-bit errors will be detected.
- If there are two single-bit isolated errors, then $e(x) = x^i + x^j$, or $e(x) = x^j(x^{i-j} + 1)$ if $i > j$. If $g(x)$ is not divisible by x, then all double errors will be detected if $g(x)$ does not divide $x^k + 1$ for all k up to the maximum message length. Suitable $g(x)$ may be found by computer search; for example $x^{15} + x^{14} + 1$ does not divide $x^k + 1$ for any $k < 32768$.

- If there is an odd number of bits in error, then $e(x)$ has an odd number of bits. As no polynomial with an odd number of terms has $(x+1)$ as factor, [2] we make $g(x)$ have $(x+1)$ as a factor to detect all odd numbers of errors.
- A polynomial code with r check bits will detect all burst errors of length $\leq r$. A burst error of length k can be represented as $x^i(x^{k-1}+\ldots+1)$. If $g(x)$ has a constant term it will not have x^i as a term, so if the degree of $(x^{k-1}+\ldots+1)$ is less than that of $g(x)$, the remainder cannot be zero.
- If the burst is of length $r+1$, the remainder $r(x)$ will be zero if and only if the burst is identical to $g(x)$. If all bit combinations are equally likely, the probability of the intervening $r-1$ bits all matching is $1/2^{r-1}$.
- For a longer error burst, the probability of an undetected error is $1/2^r$.

Many of the terms in this description of CRC codes actually arise from the use of polynomial techniques in *error correction* developing and improving on what Hamming Codes showed possible. The key difference in block error-correcting codes is the remainder from dividing $v(x)$ by $g(x)$ is known as the *syndrome* $s(x)$ and can be used to determine the error vector $e(x)$ and thereby correct any errors. Although the mechanics are similar, the design of $g(x)$ is quite different from what was described for *error detection*.

5.4 Examples of CRC polynomials

As a preliminary observation, the polynomial x^8+1 generates a simple longitudinal parity over a message of 8-bit characters, and similarly for other character lengths. [There are two bits in $g(x)$, which in the data stream correspond to similar bits of two adjacent data characters. The effect of this "window" is to Exclusive-OR bits of each data character into the corresponding bit of an overall parity character.]

Some standard error-checking polynomials are–

CRC-12	$x^{12}+x^{11}+x^3+x+1$
CRC-16	$x^{16}+x^{15}+x^2+1$
CRC-CCITT	$x^{16}+x^{12}+x^5+1$
IEEE 802	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+$ $x^7+x^5+x^4+x^2+x+1$
ATM HEC	x^8+x^2+x+1
ATM AAL3/4	$x^{10}+x^9+x^5+x^4+x+1$

CRC-12 is used for 6-bit character codes in some older banking and flight-reservation systems.

The 16-bit codes (CRC-16 used largely in North America, and CRC-CCITT in Europe) can detect all error bursts of 16 or fewer bits, all errors with an odd number of bits, and 99.998% of bursts of 18 or more bits.

The "ATM HEC" is the Header Error Control code used in ATM cells (Asynchronous Transfer Mode). It covers the 4 preceding octets and can correct all single errors and detect many multiple errors.

[2] Assume that $e(x)=(x+1)q(x)$. Then, because $e(x)$ has an odd number of terms, $e(1)$ must be equal to 1. But $e(x)=(x+1)q(x)=(1+1)q(x)=0.q(x)$ which is always 0. Therefore the assumption must be false.

The "ATM AAL3/4" is used to verify the user data of each ATM cell in the ATM Adaptation Layers 3 and 4.

The "IEEE 802" checksum has been adopted in many communications systems apart from the IEEE802.x standards, including Fibre Channel and ATM AAL-5.

In some cases the details of the checking are changed. For example, with X.25 frames, using the CRC-CCITT polynomial.

- The shift register is initially preset to all 1s,
- the check digits are inverted as they are shifted out after the information bits,
- the receiver includes the check field in its calculation, and
- the result must be 1111 0000 1011 1000.

Although the IEEE 802 generator polynomial is very widely used in many communications systems it is used with several variations.
In 802.3 Contention Bus (Ethernet) –

- the first 32 bits of the data are complemented,
- the entire frame including header and user data is divided by the generator polynomial,
- the FCS bits are inverted as they are shifted out after the information bits,
- the receiver checks that the FCS generated from the preceding received data matches the received FCS

and in 802.5 Token Bus –

- the 32-bit register for the checksum is initialised to all 1s,
- the entire frame including header and user data is divided by the generator polynomial,
- the check bits are inverted as they are shifted out after the information bits,
- the receiver includes the check field in its calculation, and
- the result, including the received checksum, must be
  $x^{31} + x^{30} + x^{26} + x^{25} + x^{24} + x^{18} + x^{15} + x^{14} + x^{12} + x^{11} + x^{10} + x^8 + x^6 + x^5 + x^4 + x^3 + x + 1$
  or, in binary, 1100 0111 0000 0100 1101 1101 0111 1011

6 Further developments

This chapter has just touched on some very large and important areas, which go far beyond what is appropriate in this book, in particular –

Error Correcting Codes These have been developed far beyond the simple Hamming Codes described in Section 3. As with the Cyclic Reduncancy checks, most codes are described by polynomial methods using the mathematics of finite fields. A good description of error correcting codes is given by Blahut [3], with a simpler introduction by Arazi [1]. Hamming [9] gives an excellent introduction to coding in conjunction with Information Theory, but without going far into error correcting codes.

The power of modern error correcting codes is demonstrated in a Compact Disk player [10]. Even minor surface scratches and dirt can cause some data loss and a 1mm disk blemish can cause a data loss of 1500 bits. Compact Disc players therefore need excellent error correction and use some of the most powerful error correcting codes. These codes (technically a cross-interleaved Reid-Solomon code) can completely correct an error burst (data drop-out etc) of 4000 data bits. At a Bit Error Rate (BER) of 10^{-3} the uncorrected error rate is less than one in 750 hours and is undetectable at a BER of 10^{-4}.

Message Authentication Some of the most complex checksums are the "message digests" and "message signatures" which are now used to authenticate messages whose integrity must be guaranteed. They must detect interference which is malevolent rather than accidental, requiring a quite different design process and performance analysis. They are really a development of cryptography and their whole discussion and theory comes from that area. In passing it should be mentioned that Cyclic Redundancy Codes are very weak cryptographically and should not be used for authentication or message security.

Data Scrambling At the physical level, where data bits are encoded on the physical medium, few data transmission techniques tolerate long sequences of 0s or 1s, or sometimes other regular repeated patterns. Information is "scrambled" or randomised to minimise regularities and ensure regular data transitions.

This is often done by *dividing* the data stream by a suitable *scrambling* polynomial and transmitting the *quotient* as the data. The receiver *multiplies* the data by the same polynomial to recover the original data. (The order of division and multiplication could be reversed, but division is much more prone to *catastrophic error propagation* in response to transmission errors and should be avoided in the receiver.) Some typical scrambler polynomials are –

$x^7 + x + 1$	V.27 4800 bps,
$x^{23} + x^5 + 1$	V.29 9600 bps,
$x^{20} + x^3 + 1$	V.35 48000 bps,
$x^{16} + x^{13} + 1$	Bell System 44Mb/s T3

The scrambler polynomials do not of themselves assure satisfactory operation; a long string of 1s or 0s may cause the shift register to freeze in an all-1 or all-0 or other repetitive state. In the V.35 standard for example, an *Adverse State* is recognised if, for 31 bits, each transmitted bit is identical to the bit 8 before it. An Adverse State causes line data to be complemented.

7 Historical Comments

The use of error-detecting and even error-correcting codes predates computers. As a very simple example, with telegrams minor errors in letters could be corrected from the context, but not digit errors. The digits were often all repeated, in order, at the end of the message as a form of redundancy check.

An even better example comes from commercial code books. In the days of manual transmission (Morse code, or teleprinters for very advanced work), transmissions were expensive, relatively open to public scrutiny, often commercially sensitive and liable to corruption. Many organisations used commercial codebooks which had 5-letter groups for frequent words or phrases. Using code groups shortened the data (reducing the cost) and also provided a measure of security by concealing the information. Transmissions were however prone to errors and the codes often included error-control mechanisms. Apart from the expected errors of transpositions and character reversals, other errors peculiar to Morse code included splitting and joining characters.

A	·−	⇒	· −	E T
B	−···	⇒	− ···	T S
	or	⇒	−· ··	N I
	or	⇒	−·· ·	D E

For example *Bentley's Second Phrase Code*[2] (1930), claimed the following properties for its codes[3]

1. A difference of two letters between any two codewords, including the spare codewords.

2. The reversal of any pair of consecutive letters in any codeword will not form a valid codeword.

3. The reversal of any three consecutive letters in any codeword will not form a valid codeword.

4. The mutilation of any pair of consecutive letters in any codeword by a pause-error in transmitting by Morse will not give another valid codeword.

Many of the early computers such as the Bell System relay machines used constant-parity codes (such as the 2-out-of-5 codes) and included extensive checking facilities. Apparently Richard Hamming was using one of these computers in the late 1940's on problems which could run over the weekend, but was continually frustrated by errors which froze the machine part way through the calculation. It was from considering that problem and trying to rectify that situation that he developed the Hamming codes which could not only detect errors (which the machine designers already knew about and handled) but could correct them. These were described in Section 3.

From that time designers of major computers have been well aware of the need to handle errors. Many errors arise from transmission or similar noise and are known as "soft" errors; they may be overcome just by retrying the operation. While other, "hard", errors may need an actual physical repair computers can be designed to recover from hard errors and reconstitute the correct results "on

[3]It is salutary to note that the book defines over 100,000 codes, all prepared without a computer.

the fly". Apparently on at least one occasion a major computer suffered a logic failure during an acceptance test, but the error correction allowed the test to continue satisfactorily.

Especially since the advent of semiconductor memories, smaller computers have tended to ignore errors, or at best pay lip service to error detection and recovery. While data buses and so on usually have parity checks, there seem to be few processors where checking is carried right through into the chip itself. This despite the fact that connectors are often the least reliable part of a system; it is all to easy for the deliberately temporary connection of a plug and socket to become accidentally temporary.

The author has heard two accounts from a major computer manufacturer in this regard. The first concerned a visit from a semiconductor manufacturer eager to show off their new semiconductor memory chips (this occurred in the mid 1970s). After a while it dawned on some attendees that the computer people could not understand why parity was *absent*, while the semiconductor people could not understand why parity was *expected*. The big systems and small systems expectations were quite different. In the second case, the same manufacturer had a big computer which was plagued by occasional errors, infrequent but enough to prevent commercial release. That problem was solved by changing to different, though apparently equivalent, backplane connectors.

Errors are a continuing problem in computing. This chapter touched on the enormous area of Error Correcting Codes, but emphasised Error Detecting Codes, as an area which is somewhat simpler and with much less coverage in the literature. The two are often inter-related under the general theme of "Error Control Codes".

References

[1] B. Arazi, *A Commonsense Approach to the Theory of Error Correcting Codes*, MIT Press, Cambridge MA, 1988

[2] E.L. Bentley, *Bentley's Second Phrase Code*, E.L. Bentley, London and Prentice-Hall New York, 1930

[3] R.E. Blahut, *Theory and Practice of Error Control Codes*, Addison-Wesley, 1983.

[4] F.P. Brooks and K.E. Iverson *Automatic Data Processing*, New York, NY : John Wiley, 1963

[5] P. Deutsch, J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", *RFC 1950*, Internet Engineering Task Force, May 1960

[6] Fletcher, J. G., "An Arithmetic Checksum for Serial Transmissions", *IEEE Trans. on Comm.*, Vol. COM-30, No. 1, January 1982, pp 247-252.

[7] Gumm, H.P. "A new class of check-digit methods for arbitrary number systems", *IEEE Trans Inf. Theory*, Vol 31, No 1, (Jan 1985, pp 102–105.

[8] R.W. Hamming, "Error Detecting and Correcting Codes", *Bell Sys. Tech. Journ.*, Vol 29, pp 147 – 160, 1950.

[9] R.W. Hamming, *Coding and Information Theory*, 2nd Ed, Prentice-Hall, Englewood Cliffs NJ 1986

[10] H. Hoeve, T. Timmermans and L.B. Vries, "Error correction and concealment in the Compact Disk system", *Philips Tech Rev.*, Vol 40, pp 166–172, 1982.

[11] ITU-T Recommendation X.224, Annex D, "Checksum Algorithms", November, 1993, pp 144, 145.

[12] L. Stone, M. Greenwald, C. Partridge and J Hughes, "Performance of Checksums and CRCs over Real Data", *IEEE/ACM Trans. on Networking*, Vol 6, No 5, pp529 – 543 Oct 1998.

[13] N.R. Wagner and P.S. Putter, "Error Detecting Decimal Digits", *Comm ACM*, Vol 32, No 1, Jan 1989, pp 106–110

[14] Verhoeff, J. "Error Detecting Decimal Codes", *Mathematical Centre Tract 29*, The Mathematical Centre, Amsterdam, 1969.